Deep Convolutional Neural Networks for Robotic Grasp Detection

CS39440 Major Project Report

Author: Oliver Thomas (olt13@aber.ac.uk)

Supervisor: Dr Patricia Shaw (phs@aber.ac.uk)

30th March 2021

Version 1.3 (Release)

This report is submitted as partial fulfilment of a MEng degree in Robotics and Embedded Systems Engineering (132C)

Department of Computer Science
Aberystwyth University
Aberystwyth
Ceredigion
SY23 3DB
Wales, UK

Declaration of originality

Name Oliver Thomas

Date 12/05/2021

Consent to share this work

Name Oliver Thomas

Date 12/05/2021

Acknowledgements

I am grateful to my supervisor, Patricia, for guiding and advising me while completing this interesting and challenging project.

I would also like to thank my family and my girlfriend Katie for their incredible ongoing support during this project.

Abstract

This project – ‘Deep Convolutional Neural Networks for Robotic Grasp Detection’ – has been designed to research and apply a potential solution to the challenging topic of grasp detection. This report details the process that was followed to: research and develop a hypothesis, design experiments, implement software to run these experiments, and analyse the results of the project. This was a complicated project that utilised a variety of different technological tools to achieve the desired functionality: from machine learning libraries to robotic middleware.

Initial research into the topic has shown that it is possible to build a network that will detect successful grasps. However, due to limitations on this project, the hypothesis was modified to state that the aim is to provide proof of concept, that could in future be refined to be more successful at detecting grasps for novel objects. By the end of this report, this proof of concept will be demonstrated and the areas to improve it further will be discussed.

Contents

Table of Figures

1 Project Background, Analysis & Process

1.1 Project Description

The aim of this project is to research and apply (in simulation) a deep convolutional neural network for the purpose of detecting successful grasp poses of objects. The project utilises many interesting technologies, from machine learning libraries to robotic middleware. This report's main focus is to explore these technologies in more depth, but it will also cover both the experimental and software engineering processes involved in completing the project.

1.2 Background & Motivation

Over the last five to ten years, a significant amount of work and research has gone into the field of robotics. Perhaps most significant is the work in the field of robotic manipulation and grasping. Despite these developments, robotic grasping is still a challenging topic, due to the many environmental inputs that effect how a grasp can actually be accomplished. One company at the forefront of robot research is Boston Dynamics [1], who have produced many new and exciting robots; including a dog-like robot that can open doors. However, although it may first appear like the robot has successfully tackled the grasping problem, it remains a very complex problem. Robots that are designed to perform grasp functions still typically use a database that contains information on how to grip specific objects that the robot will come across. However, this method does not work well in a constantly changing environment, in which a robot may find many novel objects.

This project was inspired by reading the Review of Deep Learning Methods in Robotics Grasp Detection [2]. This paper discusses the "current state-of-the-art in regard to the application of deep learning methods to generalised robotic grasping", which fascinated me. Further reading of the review led to the decision to research more into the topic of deep learning in connection with robotic grasp. This was followed by reading many other papers discussing the topic, such as Lenz et al. 'Deep Learning for Detecting Robotic Grasps' [3] (see other sources shown in the Annotated Bibliography [4]-[8]), to ascertain more details about what the project would entail. They also gave more detailed information about certain design issues that will be discussed in *Chapter 4*. For instance, Schmidt et al. paper on 'Grasping unknown objects' [9] provided details on a basic Convolutional Neural Network (CNN) structure to build off for this project.

1.3 Analysis

1.3.1 General Analysis

The nature of this project is research based; therefore, research questions that the project aims to answer need to be defined before the problem can be properly analysed. The main question to be researched is: **is it possible for a CNN to learn grasp patterns for specific objects?** It is clear, from the extensive research that has been done into this area, that this should be possible; however, there are technical limitations in place that could prevent this project from producing the most accurate model. The limitations to the implementation of this project are discussed further in *Chapters 4 and 6*. This question will be evaluated by using metrics during the model training pipeline, with results coming from the model loss function.

The next question to be answered is: **will the trained model produced be able to be applied to novel objects?** In other words, has the model been able to get a generalised view of grasping? The answer to this question will be found through simulation – the results of which will be used to evaluate the efficacy of the model and the success of the research. The design of the experiments will be shown in *Chapter 3* along with a statement of the project hypothesis.

Upon analysis of this project, it became clear from the beginning that there would be three major component elements to conduct the experiments. These components form the basis for the technical objectives and were as follows:

T1. Generating an image – grasp dataset.
T2. Training a CNN to predict grasps from images.
T3. Building a simulation environment that will be used to run experiments.

The next few sections will analyse each of these objectives, breaking down the possible technologies that could be used, plus a brief description of possible challenges that may arise (though this will be discussed in more detail in the *Design* and *Implementation* sections).

1.3.2 Grasp Dataset

By reading Caldera's review into the subject of deep learning for robotic grasping and the other papers discussed earlier, it became readily apparent that there was a main grasp dataset that was standard. This is known as the Cornell Grasp Dataset [10] and seemed to include everything that was required for the purposes of this project's research – a dataset of images with labelled grasps. However, when trying to obtain the data from the referenced links, unfortunately the data had been taken down from the public domain. Thus, more research was required to find data suitable for the project. During this research, two other datasets were found that seemed to meet the requirements – the Jacquard dataset [11], and the ACRONYM dataset [12], [13]. After more analysis of what was on offer from the two options, the ACRONYM dataset was decided upon, mainly due to the fact it is linked to the ShapeNetSem [14], [15] object database, which could easily be used within the simulation environment. In addition to this, the ACORNYM dataset is provided with access to Python scripts to manipulate the data. This task may be the most important part of the project, as the data used in machine learning is fundamental to the usefulness of the output.

1.3.3 Simulation Environment

Due to previous experience, the simulation side of the project is based in ROS [16]; however, research was required for which simulator was going to be used. The obvious choice was Gazebo [17], but other options included the OpenRave [18] and the GraspIt! [19] simulators. After analysing the strengths and weaknesses of each option, Gazebo was agreed on. This is mainly due to its ease of use, experience with the software, and compatibility with the ShapeNetSem models. Software for ROS Melodic can be developed in either C++ [20] or Python 2 [21] and it is likely that a combination of both will be used for this project.

The main decision to make regarding the simulation other than the simulator is which manipulator will be used. Upon initial analysis there are two options, the Franka Emika Panda arm [22] or the Fetch robot [23]. The Panda arm was chosen over the

Fetch robot for two reasons, the main being that the ACRONYM dataset was generated using the Panda arm and the continuity would be beneficial. The Panda arm would also be simpler to simulate due to it only being an arm, rather than an arm connected to a bigger robot.

To plan and move the arm to specific points, a library is needed to calculate the inverse kinematics and the relevant joint positions. The plan for this project is to use the MoveIt framework [24] for this purpose. Due to experience with working with this framework, no other library was considered for this project.

Another consideration to make when designing the environment, is to make the scene within the camera as similar to that produced by the grasp dataset. This will help when integrating the trained network into the simulation.

1.3.4 Convolutional Neural Network

Convolutional Neural Networks (CNNs) are tricky to implement efficiently; however, libraries and platforms such as, Tensorflow [25] and PyTorch [26], enable the implementation of machine learning algorithms such as CNNs easy. They also support customisation (such as with loss metrics or training loops) and optimisation. Tensorflow is the library of choice for this project alongside Python 3 [21], due to some previous experience and the interest in developing these skills further. Initial analysis of the problem shows that the project may need to implement a custom loss function to improve the network for this purpose; however, the standard libraries will likely be used to begin with, in order to create a working prototype. The CNN will likely be the biggest technological limitation of this project, as they require a lot of time to fully train.

1.4 Process

1.4.1 Feature-Driven Development

During the initial analysis of the project, several different software development methodologies were researched. It was decided that this project will use an agile Feature-Driven Development (FDD) methodology for planning and time management. The FDD process followed over the course of the project has been adapted to make it suitable for an individual developer. The main change is the removal of team roles and feature teams, as it is a solo project. The FDD process can be considered a plan-driven approach that incorporates agile elements. It was designed to follow five fundamental development steps that build around the features of the intended program over several one-to-two-week iterations. The FDD workflow steps are:

Step 1. Develop an overall model of the software and write the project outline. This model is the basis for the initial planning of the project onto which each iteration's designs are built upon. This can be done through updating existing diagrams or adding new diagrams that give more detail about specific features.

Step 2. Build a feature list of all the requirements by breaking up the target functionality into smaller features. Due to the agile nature of FDD, this list is not necessarily final; however, the small nature of the project makes it likely that the list shown in *Figure 1.1* will be final.

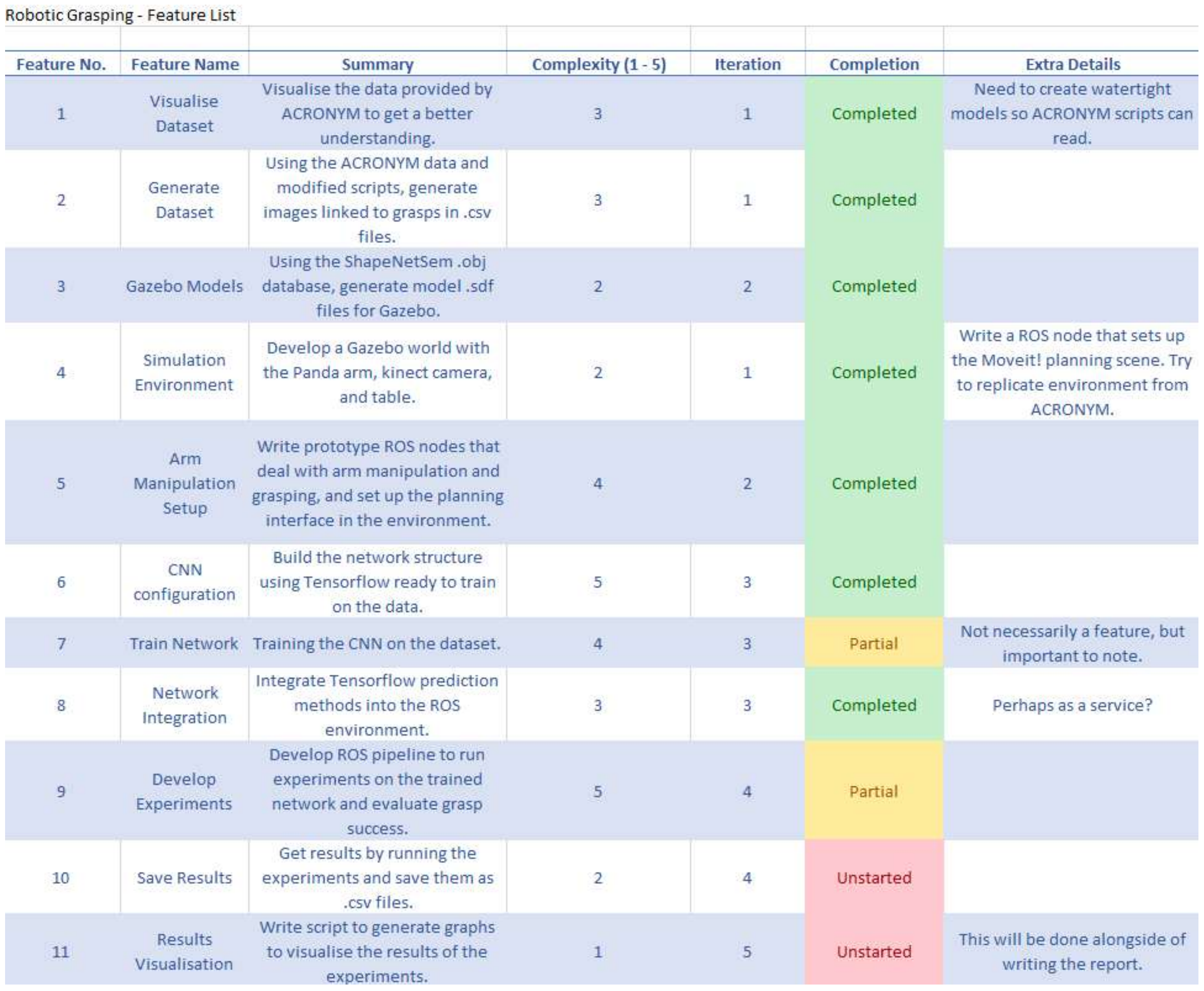
Step 3. Plan the development of the features – determine the order in which the features will be developed and who will oversee their development. A rough timeline for developing the features is shown in *Figure 1.2*.

Step 4. Design by feature – features are selected for the next iteration and assigned to different teams (though not in this case). The features are designed and added to the overall model of the system.

Step 5. Build by feature – once the designs are inspected, the feature is then implemented to that design. Usually, unit and acceptance testing are applied at this stage in the project process. However, due to the time constraints placed on the project, most of the testing was left to the end of the process.

Steps 4 and 5 are repeated until all the features have been designed, implemented, and tested. Once that is completed, the software produced should comprehensively fulfil the functional requirements set out as the features at the beginning.

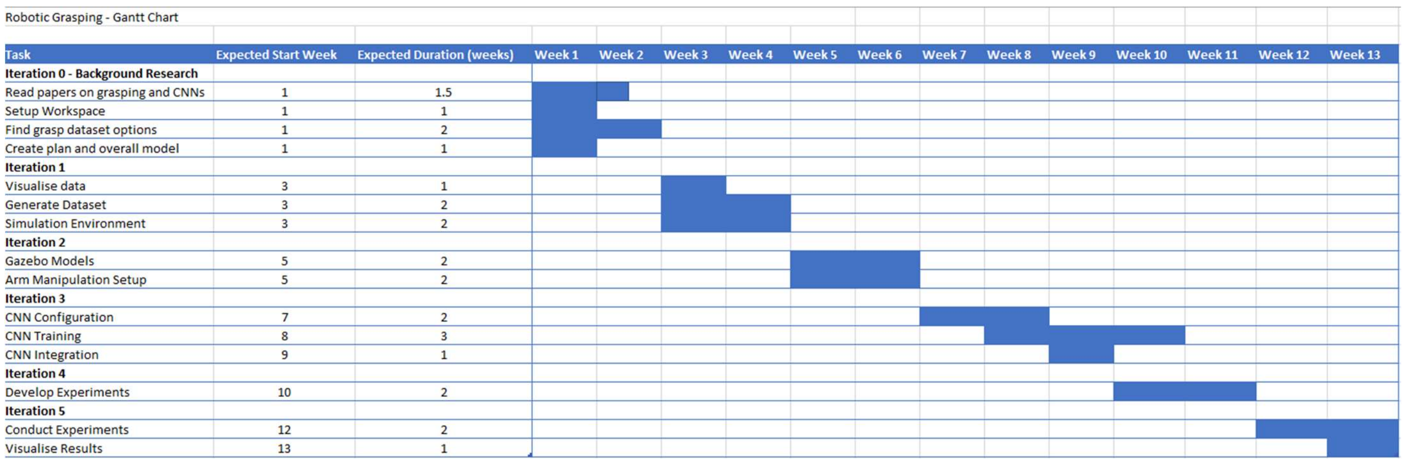
1.4.2 Feature Table and Initial Timeline



Robotic Grasping - Feature List

Feature No.	Feature Name	Summary	Complexity (1 - 5)	Iteration	Completion	Extra Details
1	Visualise Dataset	Visualise the data provided by ACRONYM to get a better understanding.	3	1	Completed	Need to create watertight models so ACRONYM scripts can read.
2	Generate Dataset	Using the ACRONYM data and modified scripts, generate images linked to grasps in .csv files.	3	1	Completed	
3	Gazebo Models	Using the ShapeNetSem .obj database, generate model .sdf files for Gazebo.	2	2	Completed	
4	Simulation Environment	Develop a Gazebo world with the Panda arm, kinect camera, and table.	2	1	Completed	Write a ROS node that sets up the MoveIt! planning scene. Try to replicate environment from ACRONYM.
5	Arm Manipulation Setup	Write prototype ROS nodes that deal with arm manipulation and grasping, and set up the planning interface in the environment.	4	2	Completed	
6	CNN configuration	Build the network structure using Tensorflow ready to train on the data.	5	3	Completed	
7	Train Network	Training the CNN on the dataset.	4	3	Partial	Not necessarily a feature, but important to note.
8	Network Integration	Integrate Tensorflow prediction methods into the ROS environment.	3	3	Completed	Perhaps as a service?
9	Develop Experiments	Develop ROS pipeline to run experiments on the trained network and evaluate grasp success.	5	4	Partial	
10	Save Results	Get results by running the experiments and save them as .csv files.	2	4	Unstarted	
11	Results Visualisation	Write script to generate graphs to visualise the results of the experiments.	1	5	Unstarted	This will be done alongside of writing the report.

Figure 1.1: The feature table for the project, as seen in week 11.



Robotic Grasping - Gantt Chart

Task	Expected Start Week	Expected Duration (weeks)	Week 1	Week 2	Week 3	Week 4	Week 5	Week 6	Week 7	Week 8	Week 9	Week 10	Week 11	Week 12	Week 13
Iteration 0 - Background Research															
Read papers on grasping and CNNs	1	1.5	■	■											
Setup Workspace	1	1	■												
Find grasp dataset options	1	2	■	■											
Create plan and overall model	1	1	■												
Iteration 1															
Visualise data	3	1			■										
Generate Dataset	3	2			■	■									
Simulation Environment	3	2			■	■									
Iteration 2															
Gazebo Models	5	2					■	■							
Arm Manipulation Setup	5	2					■	■							
Iteration 3															
CNN Configuration	7	2							■	■					
CNN Training	8	3								■	■	■			
CNN Integration	9	1									■				
Iteration 4															
Develop Experiments	10	2										■	■		
Iteration 5															
Conduct Experiments	12	2												■	■
Visualise Results	13	1													■

Figure 1.2: Initial Gantt chart to plan a rough timeline.

2 Convolutional Neural Networks

When implementing technologies, it is important to know where they come from. This helps you to understand how and why they work. This chapter's focus is on the origins and the inner workings of Convolutional Neural Networks. The aim of this research is to aid the design and implementation of a working CNN.

2.1 Background

In the 1950s, more research was done into the human visual cortex, attempting to answer two questions. First, what does the visual system do? Second, how does it do it? "In 1959, David Hubel and Torsten Wiesel described simple cells and complex cells in the human visual cortex" writes Rachael Draelos [27]. Hubel and Wiesel proposed that a combination of these two types of cell is used in pattern recognition. Each simple cell responds to bars and edges of a particular orientation in the scene as seen in *Figure 2.1*. Complex cells respond similarly to the bars and edges; however, they differ as they have a property known as 'spatial invariance'. This means the complex cells still respond when the bars and edges are shifted around the scene.

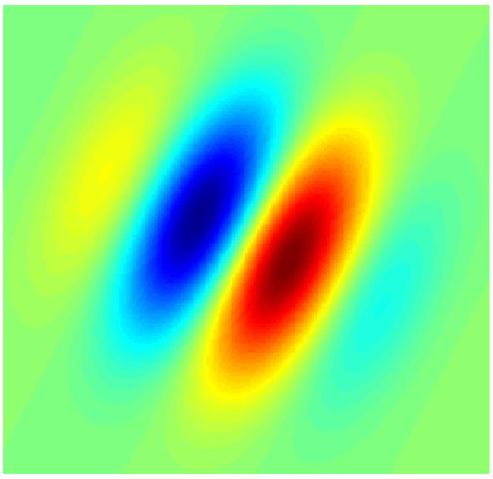


Figure 2.1: Shows a receptive field (Gabor filter-type) typical for a simple cell. Source: [28]

By 1962 Hubel and Wiesel's theory had developed. They proposed that spatial invariance of complex cells is achieved by 'summing' the output of several simple cells with similar orientations but different locations in the scene (or receptive field in the eye). This summing effect enables the complex cells to respond to similar stimuli that occur anywhere in the field.

Throughout the history of robotics, there are many examples where biology has inspired technical advancement. This is true also in the field of machine learning and convolutional neural networks. Draelos talks about how, in the 1980s, Dr Kunihiko Fukushima was inspired by the simple and complex cells and proposed a neural network model that applied this simple-to-complex concept through mathematical operations. This 'Neocognitron' [29], as it was named, was an attempt to create a computational model for visual pattern recognition and it successfully displayed the spatial variance properties desired. The neocognitron was the inspiration for more work in the 1990s that led to the modern CNNs.

More recently, CNNs have successfully been used for classifying images, such as AlexNet [30], which achieved state-of-the-art performance when trained on the ImageNet dataset in 2012. Furthermore, CNNs are now being used for a variety of purposes, from facial recognition to the analysis of medical images. This success, regarding detecting patterns within images, is what led to this project attempting to utilise this functionality in the context of robotic grasp detection.

2.2 Technical Workings

2.2.1 Convolutional Layers

Unlike a standard artificial neural network, a convolutional neural network takes an image as its input. Therefore, the network needs a way to deal with two, three, or even four-dimensional data. This is done through a convolutional layer, which is the core building block for these networks, hence the name. This layer convolves the input by using filters (or kernels) to create a feature map (the matrix on the right of Figure 2.2). This is done by calculating the dot product of the receptive field (highlighted in yellow in Figure 2.2) and the filter (the red numbers in Figure 2.2). The dot product is defined as the sum of the product of each corresponding element in the two matrices and it produces a scalar output. The filter's size is usually an odd number (3, 5, and 7 are common sizes) and determines the size of the area for features to be identified in the image. It is applied to the whole image by shifting the receptive field by a set number of pixels known as the stride, the default being one pixel. Increasing the stride leads to a decrease in the size of the feature map, due to less overlapping of the receptive fields.

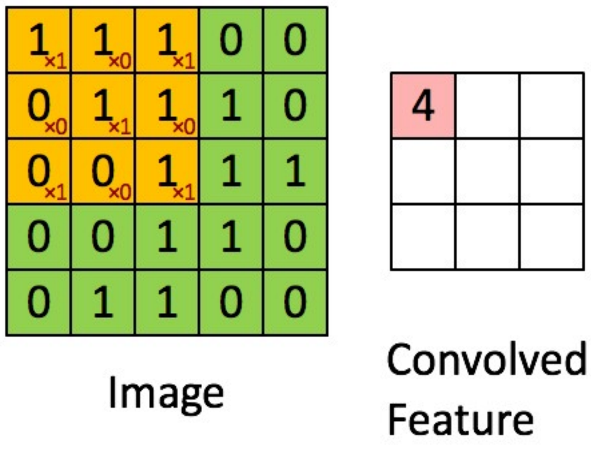


Figure 2.2: Shows the process of convoluting and input to create a feature map. Source: [31]

Once the feature map is calculated, a non-linear activation function is applied. The standard choice currently is the Rectified Linear Unit (ReLU) activation, as it is more computationally efficient than the other options. ReLU is a piecewise linear function that will output the input if positive, otherwise, it will output zero. The purpose of using an activation function such as this, is to introduce non-linearity into a network that has so far just been applying linear operations. The rectified feature map can either be the input for another convolutional layer, or it can be 'flattened' into a vector to be an input for a fully connected layer – leading to the output layer.

2.2.2 Pooling Layers

After every convolutional layer in a CNN, it is usual to have a pooling layer applied to the rectified feature map. The main purpose of this is to downsample the features; making the network more efficient and puts less strain on the available resources. There are different types of pooling, including max, sum, and average - with max pooling being the most used. Pooling layers work on a similar principle to convolutional layers. A filter, usually an even number this time, is applied to the feature pool with no overlap. So, in the common case of a filter size of 2x2, usually the stride would be set to two also, which leads to a feature map that is half the size. In the case of max pooling, the largest element in the receptive field is taken to represent that area (as shown in *Figure 2.3*). Sum and average pooling are calculated similarly but using the sum and mean average of the receptive field, respectively.

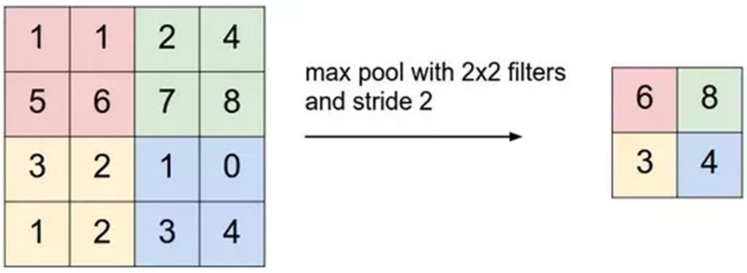


Figure 2.3: Max Pooling Diagram. Source: [32]

2.2.3 Overall Configuration

The small details of a CNN can vary greatly for different projects; however, the standard building blocks are usually used universally in a set order. This standard configuration is as follows:

- Input layer
- Convolutional and Pooling Layers alternated
- Flattening for input to Fully Connected Layers
- Output Layer with a softmax or linear activation depending on the task

This chapter has tried to explain the workings of a convolutional neural network in enough detail for the purposes of this project. However, if more information is required, there are many easily accessible papers and articles written on the subject, including [31] and [33].

2.3 SCW

This project has been given access to the Supercomputing Wales (SCW) [34] cluster, in order to train the network. The reason this has been granted is that CNNs require a significant amount of training and data, which means they are computationally intensive. This project therefore required more computing power than was initially available locally. The process to use this service will be discussed further in *Chapter 4.4*.

3 Experiment Methods

This project poses the question: can robotic grasps be successfully learnt by using a deep learning algorithm? *Chapter 3* will outline the original hypothesis made at the start of the process, and the experiments that were designed to evaluate the aforementioned hypothesis and the success of the project. The supporting software design will be covered in *Chapter 4*.

3.1 Experiment Hypothesis

The hypothesis being investigated in this project is that if a deep convolutional neural network can learn to predict grasps from images, then the trained network can be used to grasp novel objects in simulation. This functionality is trying to mimic the human intuition and experience that enables us, as a species, to pick up objects that we may have never encountered. This hypothesis has been tested in other papers, as mentioned in *Chapter 1*, with some reports quoting success rates of up to 89% in Lenz et al. paper [3] and between 70% and 92% in Schmidt et al. paper [9]. This project is not necessarily searching for the most accurate neural network or the most successful grasps; due to time and technical limitations imposed due to the nature of the project (discussed further in *Chapter 4*). However, the expected outcome is possibly a proof of concept, that could be further refined to produce better results.

3.2 Experiment Design

3.2.1 Experiment Outline

The aim of this project is to test this hypothesis through experimentation in simulation. This will involve writing software and handling different data, further discussed in *Chapter 4*. The software and data will be used to test the hypothesis by taking the following two measurements:

- Evaluation of the ground truth.
- Evaluation of the trained network.

Before describing the experiments to take these two measurements, I must define two things within the scope of this project:

- What is a successful grasp?
- What is a ground truth?

Perhaps most importantly, for the purpose of this project, I will be defining a successful grasp as a combination of two factors. Furthermore, I will also be recording semi-successful grasps that come close to picking up the object (e.g., if the object is thrown to the side). The two factors that are going to be considered for grasp success are: gripper closure and object movement within the camera frame. This will be further elaborated in the Iteration 4 section of *Chapter 4*.

A ground truth in the field of machine learning is the 'reality' that the model is attempting to predict; in this case the ground truths are successful grasp patterns for specific objects. The ground truth is dictated by the input data for a model; hence, the capability and efficacy of all machine learning models is directly linked to the quality of the ground truth. This project is making use of the ACRONYM dataset, as discussed in *Chapter 1.3*, which contains grasp and object information in relation to the ShapeNetSem object database. This grasp dataset is then considered to be the ground truth for these experiments.

As stated before, the training data for a neural network directly impacts the trained output; therefore, it is necessary to determine the quality of the dataset that is being used. The ACRONYM grasp dataset does contain information on the success of each grasp from their physics simulator; however, this does not necessarily mean that the labelled grasps will work in this project's simulation. Therefore, it is essential to conduct an experiment to ascertain the accuracy of the grasps stated in the dataset within the project simulation environment, evaluating the ground truth of the scenario, thus allowing a comparison to the accuracy of the neural network.

3.2.2 Ground Truth Evaluation

The ground truth experiment will be simple. A subset of the objects named in the dataset (a description of how this is produced is in *Chapter 4*) will be spawned into the Gazebo environment and each grasp labelled for that object will be attempted three times. Repeating the test enables an average success rate to be taken for both the different objects and the dataset as a whole. A subset is used to be more time efficient and the result of this can be extrapolated to create a picture of the success of the whole dataset.

3.2.3 Trained Neural Network Evaluation

Firstly, while training the model it is possible to assess the accuracy of the network by viewing the loss metrics. This will evaluate how close the predictions of the model are to their ground truth labels. It will also be possible to conduct tests on unseen data to check for overfitting during the training. However, this value will not necessarily indicate whether the network will work well in the simulation. Therefore, this loss metric will not be used to assess the network in this report.

The main experiments to evaluate the efficacy of the trained neural network model will be similar to that of the ground truth data. The pipeline for this evaluation is shown in F*igure 3.1*. Firstly, a random model from the ShapeNetSem database is spawned into the Gazebo simulation. An image of the object will be generated from the camera in the environment and fed into the trained model, outputting the predicted grasp. This prediction will define a transform from the object, which will then be used to attempt the grasp. The grasp success will be monitored and recorded, looking for partial gripper closure and the object to be moving up in the camera frame for a fully successful grasp. This pipeline, as in the ground truth evaluation, will be repeated three times to enable the taking of an average, thus decreasing the likelihood of outliers in the results.

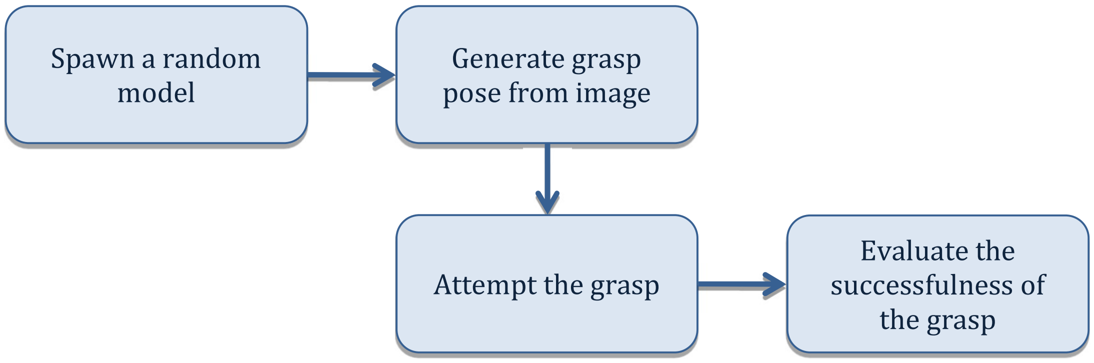

Spawn a random model
Generate grasp pose from image
Attempt the grasp
Evaluate the successfulness of the grasp

Figure 3.1: The pipeline for evaluating the trained neural network in simulation.

3.2.4 Experiment Results

There will be two ways to evaluate the results of the project’s experiments:

- Looking at the average success rate of all objects.
- Looking at the category of object where the model performs best.

The first of these two options best evaluate the initial hypothesis laid out originally, as it focuses on the generalisation of the model and its ability to predict grasps for all kinds of objects. A high average success rate would show that the model (or a similar one) could be robust enough to be applied in many scenarios. On the other hand, a low success rate could be the result of several factors, such as: a low-quality ground truth, an incorrectly setup simulation environment, or a poorly built convolutional neural network.

The second option could possibly be more interesting in the situation where the grasp success rate is mid-ranged (i.e. 40% - 60%). This is because in this situation the model appears to be working well to some degree; however, we might assume that some types of objects are holding it back. Looking at the success rate of the different category of objects could enable this to be fixed in future iterations of the project. There could be many reasons why some objects have a higher grasping rate than others, for example, the object is bigger or more complicated; therefore, will naturally be harder and more awkward to pick up. The object could even be too large for the gripper on the robot, though this can be mostly mitigated during this project due to its simulated nature.

By using both of these evaluation methods, we can generate a good assessment of how successful the model is performing, how it can possibly be improved, and how correct the original experiment hypothesis was.

4 Software Design and Implementation

This chapter merges the discussion of the design and implementation of the project, due to the decision to follow a feature-driven-development-like methodology in *Chapter 1.4*. A subsection in this chapter has been dedicated to the design and implementation of each iteration in the project.

4.1 Iteration 0

As part of the FDD methodology, the first iteration always includes creating an initial model that encompasses the overall design. This model is then updated during each following iteration. Iteration 0 will also include the development of a feature list, the setup of the workspace environment, and the development of a rough plan to help manage time.

So, the work for Iteration 0 is as follows:

- Initial workspace setup
- Feature list
- Create plan (rough Gantt chart to manage time)
- Overall model of the project

4.1.1 Feature List

As stated in the project analysis (*Chapter 1.3*), the project has been split into three fundamental tasks:

T1. Generating an image – grasp dataset.
T2. Training a CNN to predict grasps from images.
T3. Building a simulation environment that will be used to run experiments.

By analysing these tasks and the experiments outlined in *Chapter 3*, a list of basic requirements or features can be established. These features can be viewed in more detail as a table in *Figure 1.1*.

4.1.2 Overall Model

The nature of this project does not necessarily lend itself to one overall model for all software – there are many unconnected parts of software to design and implement the designs for which will be left to later iterations. However, it is still possible to create an overview of how the data will be transferred between the sections. In other words, how the output of one task becomes the input to the next.

This initial design can be seen in *Figure 4.1* as a UML case diagram. The tasks listed above form the cases in the diagram, which also shows how the tasks link together.

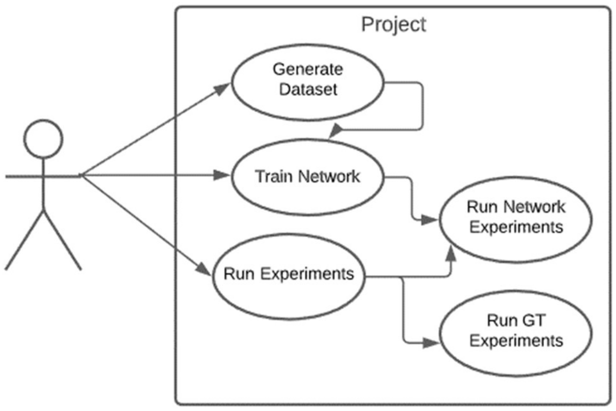


Figure 4.1: UML Case diagram showing initial design.

4.1.2.1 Initial ROS Graph

As part of the initial design, a basic plan for the structure of the ROS nodes was created. This simple overview of the ROS structure is shown in *Figure 4.2*.

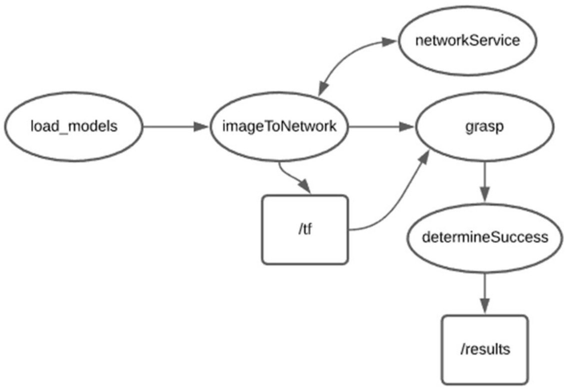


Figure 4.2: Initial ROS Graph.

It demonstrates the possible connections between the nodes to create the functionality for running the experiments defined in *Chapter 3*. The details of this structure will be further designed in future iterations.

4.1.3 Workspace Initialisation

At the beginning of the project, time was spent to setup the three main elements of the workspace. The tools setup during this Iteration were:

- GitHub
- Catkin workspace / ROS
- PyCharm [35] and VSCode [36]

4.1.3.1 GitHub

GitHub is used to create a private online git repository. This repository was used as a project management and version control tool. Two branches were used in the repository, a master and development branch. At the end of each iteration, the development branch will be merged with the master branch so that the next iteration can be developed.

4.1.3.2 Catkin Workspace

A catkin workspace is required for ROS development, as it enables the building of the C++ source code. As part of the catkin setup, the requirements for the project are established, though more libraries can be added later in the process if required. This process also creates a tidy filesystem, making working with the files easy.

4.1.3.3 PyCharm and VSCode

Most features within this project require programming. It was decided during the initial analysis that this would be split between two languages, C++ and Python. Therefore, it is necessary to setup an IDE for each of these languages. PyCharm will be used to develop the Python scripts for generating the dataset and building the Tensorflow CNN. VSCode will be used entirely for the ROS development, predominantly in C++ with some Python code written as well.

4.2 Iteration 1

Iteration 1 focuses mainly on the understanding of the ACRONYM dataset in order to generate usable data to train the network on. The simulation environment is also designed in this iteration.

4.2.1 Feature 1 – Visualise Dataset

Visualise the data provided by ACRONYM to get a better understanding.

Feature 1 did not require any major design, due to the fact that the ACRONYM dataset comes with tools [37] that enable the visualisation of the data. However, to enable the reading of the object files by these tools, it was required to use the Manifold [38] library to create watertight models. By integrating the Manifold code into a Python script ('ACRONYM OBJ WATERTIGHT FILES.PY'), each of the ShapeNetSem object files was able to be formatted.

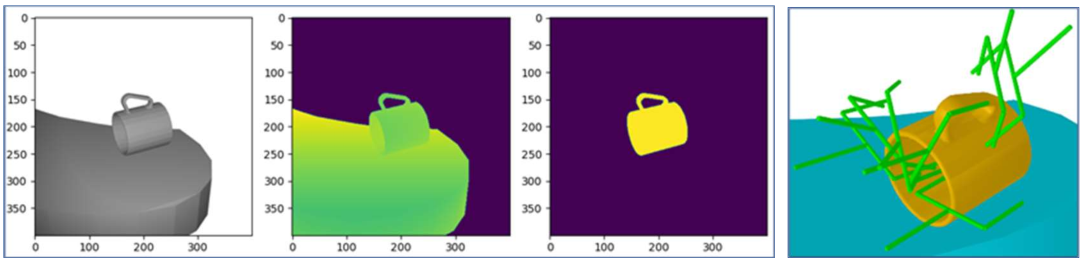


Figure 4.3: The visualisation of the data using ACRONYM scripts. Left to right: greyscale, depth, segmentation, grasp visualization.

The ACRONYM tools provide three command-line interfaces to interact with the models: visualising the model, visualising the grasps, and generating a random scene with grasps that are not in collision. The image rendering and grasp visualisation tools are demonstrated in *Figure 4.3*. The implementation of these visualisations enables an analysis of the information provided in the dataset. It will also be of use when designing Feature 2.

4.2.2 Feature 2 – Generate Dataset

Using the ACRONYM data and modified scripts, generate images linked to grasps in .csv files.

Feature 2 required some thought for the initial design, as it is vital in any deep learning task that there is enough data to train on. The ACRONYM dataset certainly provides enough data (17.7M grasps on 8,872 objects generated in a physics simulator), but how it is presented to the machine learning model is important. For this project, it was decided that each object would be randomly placed into a scene fifty times, with an image of the scene generated and with ten of the available successful grasps being stored. The dataset provides each grasp as a 4x4 transformation matrix from what is estimated to be the centre of the object. Using this as a label for the object images would be computationally inefficient, requiring the data to be simplified into a six-dimensional format (x, y, z, roll, pitch, yaw). This was accomplished by using the transformations library provided by tf. It was decided that the dataset would have the following structure:

X	Y	Z	ROLL	PITCH	YAW	IMAGE

Once that was determined, the scripts to generate the dataset could be designed. This design is shown in *Figure 4.4* and includes a combination of modified ACRONYM tool scripts and new scripts.

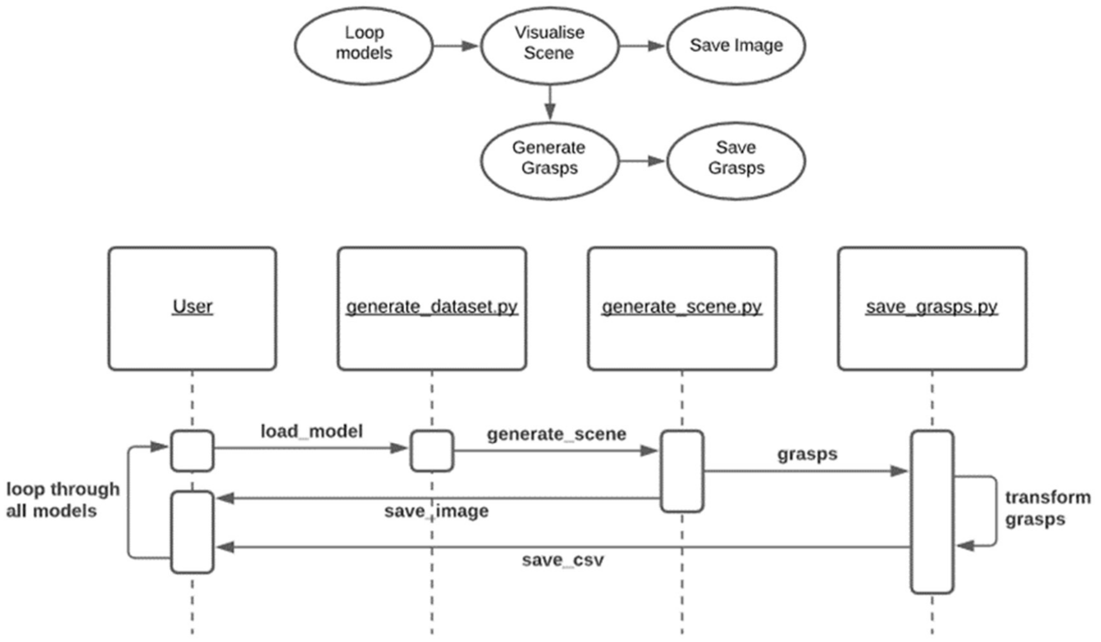


Figure 4.4: Two diagrams - first a simple break down of the feature - second a UML Sequence diagram showing the interaction between the scripts.

This series of scripts produces both a .png image and a .csv file containing the selected grasps. The image and corresponding grasp files share a name, meaning another Python script ('load_dataset.py') can match the two files to create one large .csv file with the structure mentioned above. The image column in the resulting table is a string that corresponds to the image file name.

The implementation of this caused some issues due to different Python versions being required by the transformations library and the ACRONYM tools. This problem was solved by using the python subprocess module that enabled a Python 2 to be called from Python 3. While this is probably not the most computationally efficient way to do this, it was the quickest solution enabling the feature to be produced as quickly as possible. It also meant that no existing libraries had to be re-written for the purpose of this project.

4.2.3 Feature 4 – Simulation Environment

Develop a Gazebo world with the Panda arm, kinect camera, and table.

When designing the ROS simulation environment, it was important to mimic the setup from the ACRONYM scenes. This is because the images that the neural network will be trained on will expect a similar scene when predicting in the simulation. The initial design for the environment utilises the Franka Emika Panda arm, a Kinect depth camera, and a table, as seen in the feature description. This would have worked well; however, for an unknown reason the Panda gripper was not working in the Gazebo simulation. Unfortunately, due to this complication, the environment feature was delayed while trying to fix the gripper controllers. After spending a considerable amount of time trying to fix this, it was decided to change from the Panda arm to use the Fetch robot instead (due to previous experience of this working). The fetch robot also has an inbuilt head camera which makes the environment simpler to build.

The design of the environment is purely observational; therefore, does not require any design diagrams. The implementation of this feature utilises ROS launch and world files to successfully load the same environment each time.

4.2.4 Reflection

Upon reflection, the main challenge during this iteration was surprisingly setting up the Gazebo environment. These problems could have been avoided if the fetch robot had been used initially. Another issue faced was the time it took to generate the watertight models. In future this could be sped up by multithreading the task; however, this process would still take a long period of time, due to the large volume of files required.

4.3 Iteration 2

Iteration 2's focus is on the simulation environment; generating the models that can be spawned; creating the nodes that will control the arm and gripper.

4.3.1 Feature 3 - Gazebo Models

Using the ShapeNetSem .obj database, generate model .sdf files for Gazebo.

Spawning objects into a Gazebo world requires a specific file format: SDF. SDF files are formatted as xml, with tags to represent specific properties of a model. Gazebo needs these properties in order to simulate the object successfully. In SDF there is a tag <uri>, which takes a mesh input provided by the ShapeNetSem database (.obj or .dae) and enables the model to be displayed in Gazebo. The physical information needed for the SDF file is provided by the ACRONYM dataset. Alongside the SDF files, each model needs a .config file in its folder to initialise basic information about the model, such as its name.

To generate all the models for simulation in Gazebo, a Python script was designed as show in the sequence diagram in *Figure 4.5*.

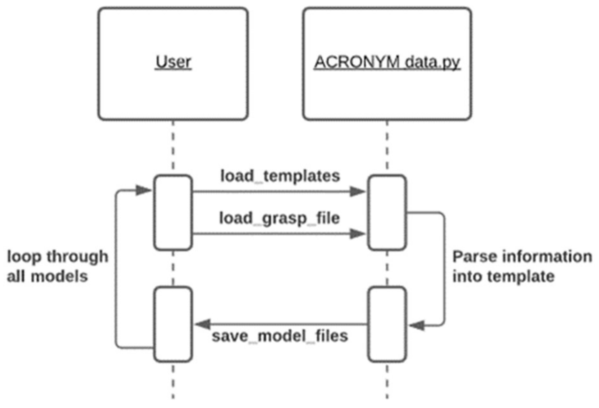


Figure 4.5: Design of the Gazebo model generation script.

This script was implemented using the loading of template text files (for both SDF and config files) and using the Python string format function to parse the object physical information. This made the script neater, more maintainable, and more understandable. The script also implements a multithreading pool that enables the script to run faster.

4.3.2 Feature 5 - Arm Manipulation Setup

Write prototype ROS nodes that deal with arm manipulation and grasping and set up the planning interface in the environment.

This feature was designed and developed using a mixture of spike work and experience. The initial plan for the grasping functionality was to use the MoveIt library, as discussed in the project analysis section. The system makes use of MoveIt for inverse kinematics and path planning to reach a goal position. The goal position is set by a static transform broadcaster from the base link of the fetch robot to the grasp pose. The transform tree will initially be hard coded within the grasping node; however, in later iterations, the grasp frame will be determined by another node. The grasp poses will either be from the trained model or the grasp dataset. Once the location has been reached, the node uses a gripper controller action to close the gripper around the target object and then retreat. In future iterations, this is when the success of the grasp will be measured.

4.3.3 Reflection

This iteration made very clear progress in regard to the completion of the project, as two key components were put in place. Furthermore, there were not many issues that arose, due to experience with most of the technologies used during this iteration. However, there was an issue with some of the gazebo models that do not spawn in the pose defined. This is due to the defined origin in the ShapeNetSem model file and cannot be changed easily. As this only seems to occur for a few of the models, the future code can be written to bear this in mind.

4.4 Iteration 3

Iteration 3 focuses mainly on implementing and training the Convolutional Neural Network (CNN). The following features form a major part of the project but rely heavily upon the data generated in the previous iterations.

4.4.1 Feature 6 – CNN Configuration

Build the network structure using Tensorflow ready to train on the data.

The design for the CNN was inspired by that of a CNN from Schmidt's paper, as discussed in *Chapter 1*. The configuration follows the standard structure of convolutional layers followed by fully connected layers discussed in *Chapter 2* and can be seen in *Figure 4.6*.

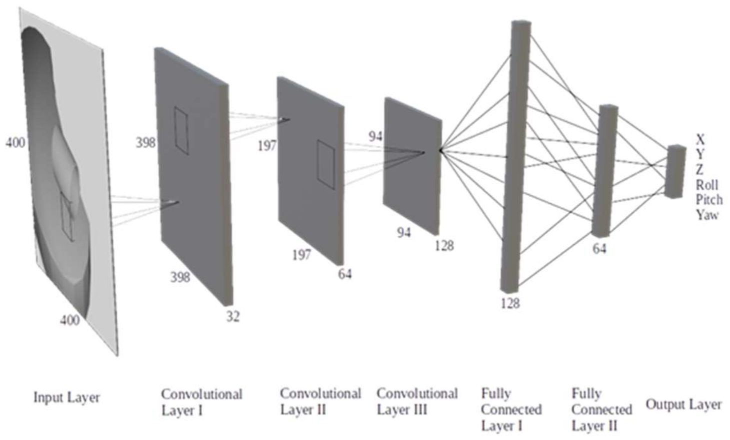


Figure 4.6: The deigned configuration for the CNN.

This network is seven layers in total, with four sub-sections:

Section 1. The input layer – the network takes a 400 x 400 x 1 greyscale image converted to a tensor as input.
Section 2. The three convolutional layers uses a filter of size 3 x 3, a stride of 1, and an increasing kernel size of 32, 64, and 128, respectively. After layer two and three there is a max pooling layer, with a 2 x 2 filter and a stride of 2.
Section 3. There are two fully connected layers following the flattening of the output of the third convolutional layer. The first layer has 128 nodes, decreasing to 64 in the second.
Section 4. The output layer consists of 6 nodes to represent the grasp coordinates.

The implementation of this configuration was made simple by using a Tensorflow Keras Functional model. This also allowed batch normalisation and dropout to be applied to the layers, to help reduce overfitting and to increase the accuracy. An example of a Tensorflow functional model can be seen in *Appendix C*.

Later in the project's development, it was decided to implement a new design of CNN, keeping the structure, but creating new training and loss functions. This was done to try and fine tune the network for the specifics of the project. The loss function was updated to minimise the mean absolute error from the closest grasp for that object. Updating the loss function in this way meant the training loop also had to be rewritten to allow access to the current input data. These new functions can be viewed in *Appendix D*.

4.4.2 Feature 7 – Train Network

Training the CNN on the dataset.

In *Chapter 2*, it was briefly discussed that access to the Supercomputing Wales cluster (SCW) has been granted for this project. This was used to train the CNN on the generated images. While this helps lift certain technical limitations with training the CNN, to fully train a large network still takes a lot of time.

In order to use SCW, certain things needed to be initially setup. Setting up the Python 3 environment on SCW was done using Anaconda 3 [39], which gave access to the most updated Python libraries – particularly Tensorflow 2, which was not available by default on SCW. Furthermore, to run jobs on SCW, a batch file containing information about the job and commands to run the job was required (an example is in *Appendix E*). This enables the automatic queueing of all jobs. This does mean that occasionally a job must wait a while before resources are available to run it. Due to the slow speed, large amount of data, and limited resources, the network was trained for ten epochs at a time. At the end of each set of epochs, the weights would be saved so that the training could restart in the same place.

Part of the process of training a neural network is testing different hyperparameters, such as learning rate, batch size, and the number of epochs. It was while testing these different options that it was decided to write the new training functions previously mentioned to see if these would increase the performance of the network. The loss results of training the custom model are shown in *Figure 4.7*.

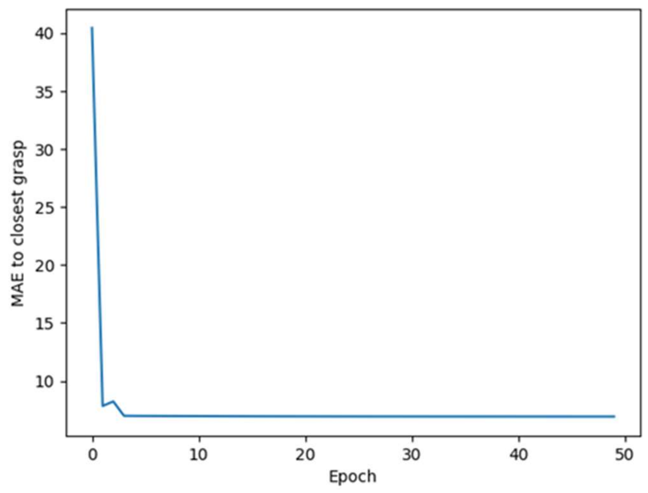


Figure 4.7: A plot of the loss metric decreasing after each epoch.

It is clear that the loss converges very soon after starting the training process. This also occurred when training the standard model, which achieved similar loss performance.

4.4.3 Feature 8 – Network Integration

Integrate Tensorflow prediction methods into the ROS environment.

This feature deals with the utilisation of the trained CNN model within the ROS environment. Since the main structure for ROS was designed in Iteration 0 (see *Figure 4.2*), including how the network would be integrated, there was no need for further design. It was, however, decided to integrate the network into the environment using a ROS service, meaning it could be called from any node within the system if required.

A ROS service requires a definition for a pair of messages: one to define the request type, and the other for the response. In this case, the service will be called with a sensor_msgs Image. The response will be floating point array of size six, which will be the predicted grasp pose. Within the service an OpenCv [40] bridge will be used to format the data from the Image message, so that it is readable by the Tensorflow model.

4.4.4 Reflection

Looking back, this iteration may have produced the first signs that the project would not be as successful as initially first hoped. This can be seen through *Figure 4.7* as the loss metric converged after very few epochs to a value higher than expected from both network models. However, as stated in *Chapter 2*, this project's success will not be determined by the network loss metric, but by conducting the experiments in the simulation environment.

Another issue faced during this iteration was during the process of integrating the network into the ROS structure as a service. The issue was caused by a mismatch of Python versions. ROS melodic is limited to using Python 2, which due to a recent depreciation, only has access to older versions of libraries, specifically Tensorflow. This meant that errors occurred when trying to load files saved by a newer version. This was solved by saving the files in a format that the older library could read. However, this did make the initial process more complex.

4.5 Iteration 4

Iteration 4's main goal was to develop the software for running the experiments in the simulation environment. The experiments to be implemented have already been designed in *Chapter 3*, also briefly defining a successful grasp in the scope of this project, but this will be elaborated next.

4.5.1 Grasp Criteria

The definition for a successful grasp in this project was briefly described in *Chapter 3*. It was stated that there will be two factors to determine a successful grasp, these being gripper closure and object movement within the camera frame.

Firstly, gripper closure can be used to determine the success of a grasp because, if the fingers of the gripper are together after grasping, then the object is not between them. Therefore, the grasp was not successful. On the other hand, if the fingers are partially separated, then the object is likely to be between them; thus, the grasp was successful.

Secondly, object movement can also be watched to assess the grasp success. If the object moves up within the camera frame it is likely that the grasp was successful, as it is following the direction of the retreat of the gripper. A semi-successful grasp can also be measured using this measurement. For example, if the object is thrown from the gripper to either side, then this can be seen using the camera.

By using a combination of these two factors, a numerical value can be assigned to each attempted grasp to determine how successful it is. The logic for this is as follows:

- A grasp is fully successful if the gripper is not closed, and the object moves up within the camera frame. This grasp would be assigned a 1.
- A grasp is semi-successful if the gripper is closed, and the object has moved to either side in the frame. This grasp would be labelled with the value 0.5.
- A worse grasp than this would be if the object had been dropped off the table. This can be seen if the object moves down in the camera frame and would be assigned the value 0.25.
- Finally, if the object does not move at all, then the grasp is completely unsuccessful. Therefore, it is given the value 0.

The numerical label for each of these grasps enables the visualisation of the success data; helping to evaluate the results of the experiments in future iterations.

4.5.2 Feature 9 - Develop Experiments

Develop ROS pipeline to run experiments on the trained network and evaluate grasp success.

The initial design for the ROS pipeline to conduct the experiments is shown in *Figure 4.2* and does not need much alteration, except for a node to position the object. By using this design, a state machine configuration combining these nodes can be written. This state machine was implemented by using both topics with messages and the ROS parameter server. In both the depth detection and determine success nodes OpenCv moments were utilised to locate the object within the camera frame. At the beginning of each loop, the location of the object is stored in the success node to enable a comparison after the attempted grasp.

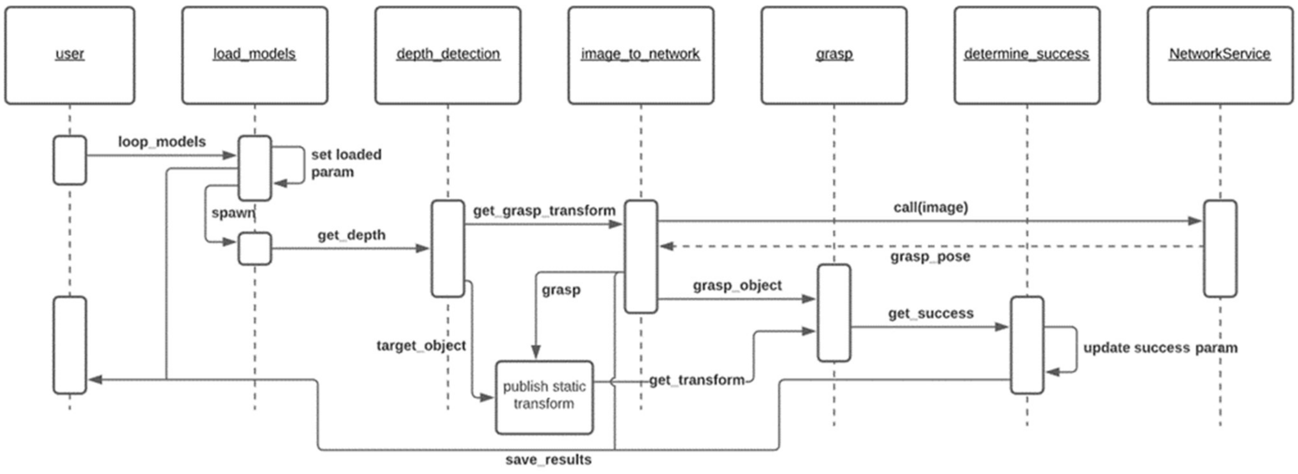


Figure 4.8: UML Sequence diagram of design of the neural network model experiment.

Figure 4.8 shows the design of the model experiment pipeline in more detail. The design uses five parameters on the ROS server to achieve the required functionality. These are:

- Num_models – used to represent how many models will be assessed during the current experiment. This is set in the experiment launch file.
- Loaded – used to notify all the nodes that a new model has been loaded.
- Grasp – used as part of the grasping process to keep track of which stage of grasping the robot is in. Also used by the success node to know when to judge the success of the grasp (i.e., when the object is meant to be held up to the camera).
- Initial – used to ensure the values in each node initialises its values when a new model is spawned into the simulator.
- Successful_grasps – used to keep track of the ongoing total, representing the grasp success. This is incremented in the determine_success node, depending on the successful grasp criteria.

The experiment for determining the ground truth success in the simulation follows the same pipeline as shown in *Figure 4.8*. However, instead of getting the grasp poses from the model, they are retrieved from the dataset from within another node that replaces ‘load_models’.

4.5.3 Feature 10 - Save Results

Get results by running the experiments and save them as .csv files.

The design for this feature has been included in *Figure 4.8*. It can be seen in the sequence diagram that information from three of the ROS nodes are saved to the results file. This file will be a .csv file following the structure bellow:

Object Name	Grasp Pose (as a list)	Grasp Success

This feature involves the running of the experiments that were designed and implemented as part of Feature 9 (the ROS Graph of this can be seen in *Appendix G*). This involves running each experiment three times, to enable the taking of an average as described in *Chapter 3*. The results from the repeated experiments will be saved in different files and combined as part of Feature 11. They will also be visualised during that iteration and conclusions will be drawn – presented as part of *Chapter 6*.

4.5.4 Reflection

On reflection, this iteration was the hardest to implement, particularly Feature 9 which was expected. This was mainly due to the nature of simulating with ROS, which can be a tricky, tedious process, in which it can be hard to find bugs. However, this did not stop the feature being successfully developed to meet the requirements of the project.

An issue that was encountered during this process was that the weights of the first CNN took a considerable amount of time to load into the software. In comparison, the slightly lighter weight model used by the custom model loaded very quickly. It was decided to just use the custom model as the performance of both models was very similar anyway.

4.6 Iteration 5

4.6.1 Feature 11 – Results Visualisation

Write script to generate graphs to visualise the results of the experiments.

The output from running the experiments is three separate .csv files. The initial part of this feature is to combine these files into one usable table of results. The data also includes different types of each object, mugs for example, and these will also be combined to make larger categories containing the average success of the individual objects. This ensures the visualisation of the data is less crowded and more understandable. This functionality will be achieved through a Python script using the Pandas [41] library to build the datasets with the following design:

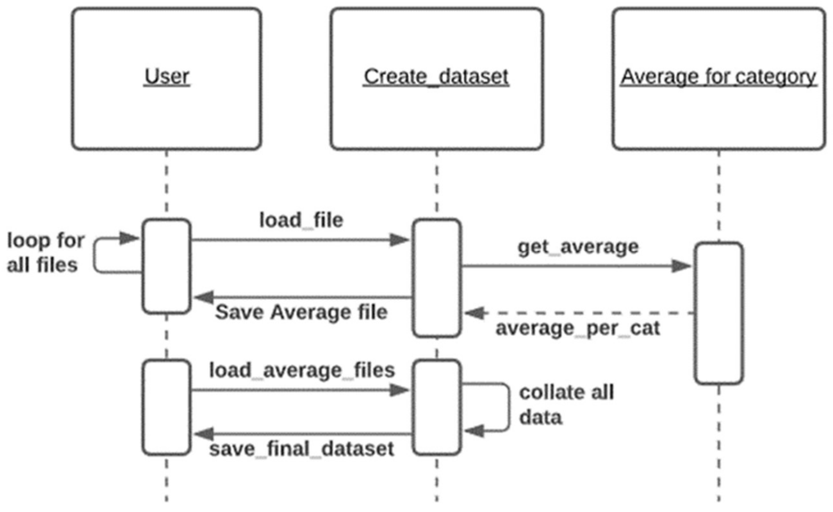


Figure 4.9: Design for collating results into one dataset.

The collated dataset will then be visualised using Python and Matplotlib to produce three plots: a word-cloud based on the highest success rate objects, a 'top fifteen' bar chart, and a 'bottom fifteen' bar chart. These charts should give a comprehensive view of the success of specific objects in the simulation. The average success of all the grasps will also be available from the dataset.

4.6.2 Reflection

There was not much to go wrong during this iteration. This meant there was more time to analyse the different results that were received from running the experiments. This analysis is discussed in *Chapter 6*.

5 Testing

5.1 Overall Approach

The overall approach to testing for this project is a combination of two methods: automated testing (including unit testing) and visual testing. Due to the nature of the project, there is no need for user, interface, or stress testing, as there are no intended users. The following sections will discuss both aspects of the testing plan; considering what testing has been achieved and what would be planned if there were no time constraints.

A short list of the main tests has been created to keep track of testing the fundamental features. These broader tests can be seen in *Appendix F*. It is worth mentioning that while the different elements and programs were being developed, smaller manual tests were also conducted during each iteration to make sure the code ran as it should and did not break other features of the project. This involved a lot of code debugging and fine tuning.

5.2 Automated Testing

Automated testing has been and would be utilised to ensure the reliability of much of the code. Including the ROS nodes for running the experiments and the pre-processing scripts used to generate the dataset. There are two ways that these automated tests have been implemented, through standard checking within the code and through separate unit tests. However, the nature of the project limits the number of useful unit tests that can be applied to the source code.

Due to time constraints placed on the project and the fact that automated tests were left to the end of the process, only a few automated tests were implemented, not including any unit testing. The decision to leave automated testing to the end was made early on in the process. Due to the nature of feature-driven development, it is not necessarily clear what the final structure of all the features will be; therefore, it could be a waste of time to develop unit tests that would later be discarded. This would have been unadvisable, especially with the tight timeline for the project. The implemented tests are seen in the test table in *Appendix F* along with an example and are applied from within the Python scripts, using a mixture of if statements, for loops, and assert statements. Combined, these methods ensure the correct broad functionality. Unit tests would in the future be included to test more of the lower level logic and would be written using the PyTest [43] library.

5.3 Visual Testing

As this project is based mainly in the Gazebo simulator, a lot of the testing was done visually. This included tests for the setting up of the environment and tests to ensure the manipulator was behaving how it should, before and during the experiments. These sorts of tests were completed during each iterations' development to ensure that the functionality of each feature was improving. Visual testing may not be as reliable as computational based testing; however, with features such as these, it is hard to test them in any other way – especially within the scope of this project.

6 Results and Conclusions

This chapter is dedicated to analysing the results from the experiments conducted as part of this project. The experiments are detailed in *Chapter 3*, one for determining the ground truth of the project, the other for the success of the trained model in simulation. The results of the experiments will be compared against the original hypothesis:

If a deep convolutional neural network can learn to predict grasps from images, then the trained network can be used to grasp novel objects in simulation.

These results will prove this hypothesis to be correct by demonstrating a proof of concept that, in future, can be refined to become more successful.

6.1 Ground Truth Success

As part of any machine learning project, it is important to evaluate how good the training data is, as this will heavily influence the outcome of the model. The ground truth in this case is the grasp dataset and it was tested using the same environment and pipeline as the trained network. The results of these experiments are shown in *Figure 6.1* and the average result for the whole dataset was **51%**. This result was lower than anticipated and could be due to bad data in the dataset. It could also be caused by the environment not being set up in a way that matched how the grasps were generated. It is difficult to say for certain which of these factors it could be, but it is most likely a combination of both. When watching the experiments run, it was noticed that some of the grasp transforms were severely out of line with the object they were intending to pick up.

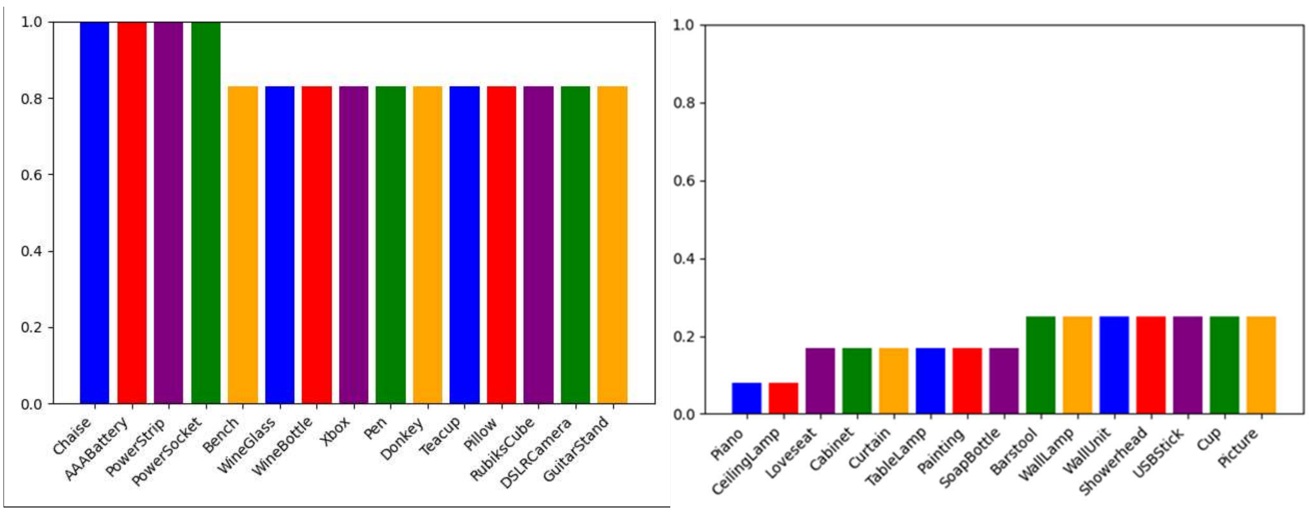


Figure 6.1: Bar chart showing results of the ground truth success in simulation. Left: most successful objects. Right: Least successful objects.

6.2 Model Success in Simulation

As stated in *Chapter 4*, only the custom model was utilised to run the experiments. Therefore, only the results from this model will be analysed in this section. Running the experiments in the designed simulation environment did not produce as good results as was originally hoped. The average success rate of all the objects was only **35%**. There could be a few reasons why this average is lower:

- The CNN was not trained for long enough.
- The ground truth on which the CNN was trained was not good enough.
- The simulation environment was not set up correctly.

As can be seen in *Figure 6.2*, some objects have worked well and can be grasped with 60-80% accuracy, but this success rate decreases quite quickly. This suggests that the model has learnt to grasp some objects better than others and considering that this pattern is also observed in the results of the ground truth, this is a definite possibility. In addition, the ground truth success rate was also lower than expected and this would have had a knock-on effect on the success of the CNN model. These factors combined could lead this project to believe that the issue lies in the original data.

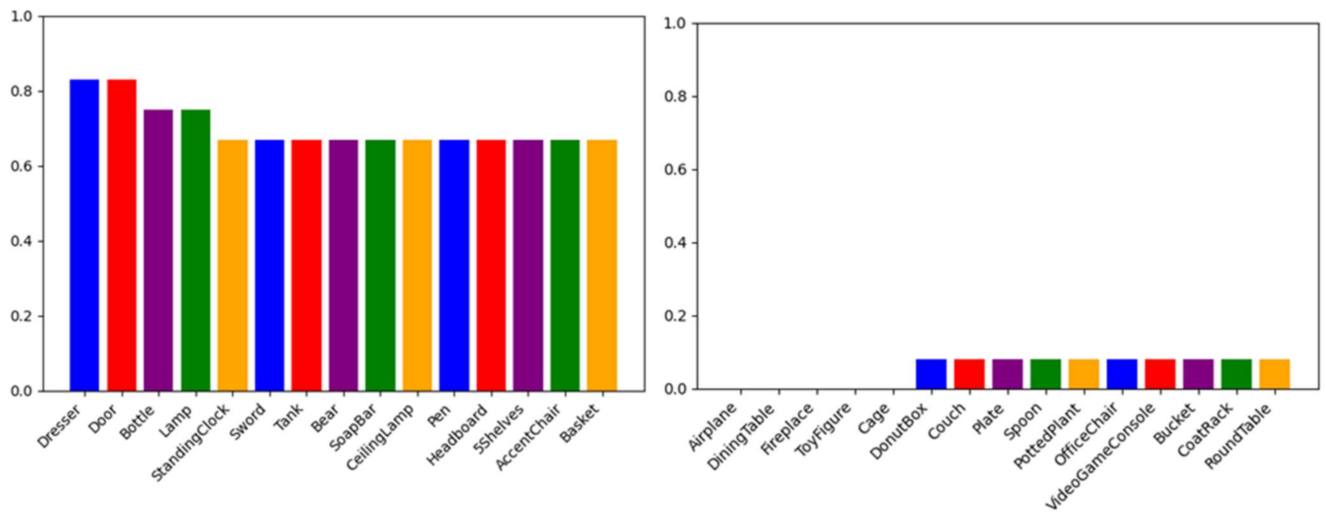
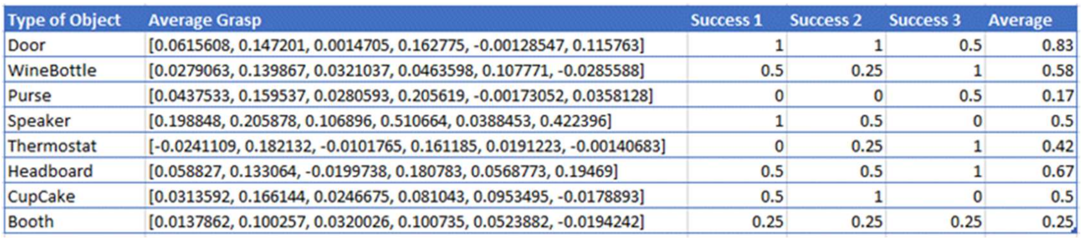


Figure 6.2: Bar chart showing results of the model success in simulation. Left: most successful objects. Right: Least successful objects.

However, when observing the experiments running, the grasps that fail are not far from being successful (examples of the model generated grasps are shown in *Figure 6.3*). This suggests that this result could be improved by further training and fine tuning of the CNN. Therefore, it is likely that a combination of the three factors mentioned above will have caused the lower-than-expected average.

Looking at the generated word-clouds in *Figure 6.4* and the most successful objects in the bar charts, suggests that the easier objects to grasp in this project's simulation environment are either box shaped, or objects with a thin section such as a 'Wine Glass' or a 'Bottle'.



Type of Object	Average Grasp	Success 1	Success 2	Success 3	Average
Door	[0.0615608, 0.147201, 0.0014705, 0.162775, -0.00128547, 0.115763]	1	1	0.5	0.83
WineBottle	[0.0279063, 0.139867, 0.0321037, 0.0463598, 0.107771, -0.0285588]	0.5	0.25	1	0.58
Purse	[0.0437533, 0.159537, 0.0280593, 0.205619, -0.00173052, 0.0358128]	0	0	0.5	0.17
Speaker	[0.198848, 0.205878, 0.106896, 0.510664, 0.0388453, 0.422396]	1	0.5	0	0.5
Thermostat	[-0.0241109, 0.182132, -0.0101765, 0.161185, 0.0191223, -0.00140683]	0	0.25	1	0.42
Headboard	[0.058827, 0.133064, -0.0199738, 0.180783, 0.0568773, 0.19469]	0.5	0.5	1	0.67
CupCake	[0.0313592, 0.166144, 0.0246675, 0.081043, 0.0953495, -0.0178893]	0.5	1	0	0.5
Booth	[0.0137862, 0.100257, 0.0320026, 0.100735, 0.0523882, -0.0194242]	0.25	0.25	0.25	0.25

Figure 6.3: An extract from the results of the CNN model experiments.

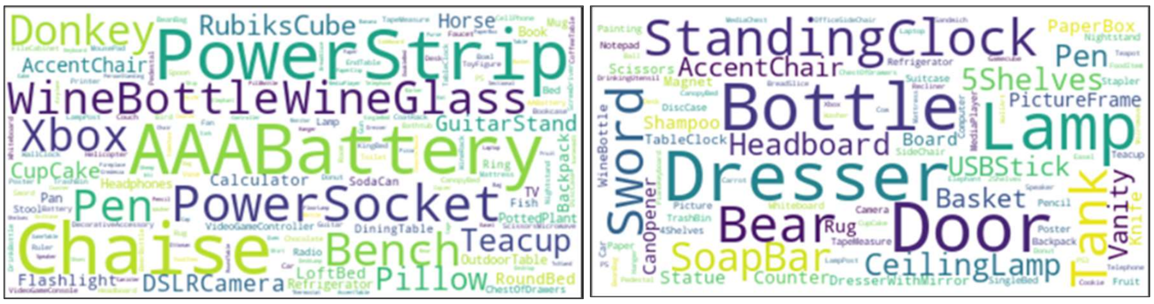


Figure 6.4: Word-clouds that show the objects grasped in simulation. The bigger the word, the more successful the grasp. Left: Ground truth experiment. Right: CNN model experiment.

6.3 Conclusions

In conclusion, this project has successfully run experiments to ascertain the validity of the hypothesis. These experiments have returned results that are less successful than were anticipated; however, this lack of success can be explained by a few key factors and given more time could be effectively minimised to produce better results in the future. Therefore, this project has demonstrated a refinable proof of concept by the way that it has utilised a deep convolutional neural network to successfully grasp objects in simulation. Future versions of this project could make further progress when it comes to the accuracy of the model.

7 Critical Evaluation

7.1 Project Requirements and Design Decisions

7.1.1 Project Features

This project's purpose was to utilise a deep learning approach for the task of detecting grasps for objects. It was designed to test the hypothesis stated in *Chapter 3*. Therefore, the requirements were built around the software and data for running the experiments.

The requirements that were outlined in *Figure 1.1* were mostly identified at the beginning of the project. This was done through research into the topic to identify the key features that would be required to accomplish the desired functionality. This research, in conjunction with discussions with the supervisor, initially enabled a broad overview of the features. This led to the creation of a more detailed feature list, that allowed flexibility for design during the iterations. It also provided a good vision of the work required for the desired system.

Overall, the features of this project were well planned and structured, enabling the fulfilment of the project aims. This list could also become a springboard upon which the possibility of new features could be developed and expanded.

7.1.2 Design Decisions

As this project was following the FDD methodology, design was a key focus point. This involved designing an overall model of the project at the start of the process to shape the direction of the project. Furthermore, at the beginning of each iteration there was a time in which decisions were taken about the design of the features. This included creating new diagrams and modifying the original model to include new features. These diagrams aided the development and implementation of key algorithms and scripts within the system.

Right at the beginning of the project, two key decisions were made because of the initial analysis of the problem. The first being to use ROS as the environment to develop the 'hypothesis experiments'. The second decision was to utilise the Tensorflow library to build the CNN. On reflection, it was definitely a good idea to get these decisions agreed as soon as possible, especially as previous experience of these technologies was available. This enabled the project to proceed more quickly onto decisions that required more research to get right.

The most complicated design decision during the project was most likely the way the dataset was generated. This is the case because there were many different variables involved in the process, such as how many grasps per image, or how many random rotations of the objects to render. Furthermore, the success of the entire project would rely heavily on the quality of this data (as discussed in *Chapter 3*). On reflection, it may have been advisable to initially use less repeated data (i.e., less images of each object) to allow more training to be done in the time available.
Overall, the decisions taken during the design of the project have produced good individual results that when integrated together have built up the required functionality.

7.2 Methodology and Time Management

This project is utilising the feature-driven development (FDD) methodology with a few slight alterations that tailored the process to suit a solo project. This process was chosen as the upfront design plus the iterations seemed to suit the nature of the project well. Looking back, this methodology worked well for this project and helped keep track of the development of all the features, enabling the fulfilment of each of them on time. Time management was an essential part of the project, as there was a lot to complete; the methodology definitely helped to keep all activity on track.

As part of FDD, it is standard to include testing during every iteration of the process. However, due to time constraints it was decided to leave the testing to the end. This meant that there were not many automated tests developed for the project and if this project were restarted, then a different approach to testing would be taken. This ended up being the only issue with the process as part of this project.

7.3 Tools and Technologies

This project makes use of a large range of different tools and technologies, chiefly the Robot Operating System (ROS) and Tensorflow 2. These two technologies form the main part of the project's functionality, ROS for the simulation and Tensorflow for the CNN. They were selected at the beginning of the project and proved to be effective at fulfilling the needs of the project. ROS in particular was utilised well, making use of many different features of the tool, such as services, actions, and topics, to produce the desired functionality.

7.4 Final Product and Results

The final product of this project is a combination of different things. It includes the pre-processing scripts written to generate the dataset, the ROS scripts to run the experiments, and the scripts to train the Tensorflow CNN. Overall, the design and implementation of all these components was successful and have worked well together. However, unfortunately as discussed in *Chapter 6*, the results of the project were not as good as hoped, which was due to a combination of reasons. Looking back, it would have been good to have been able to evaluate the ACRONYM dataset before committing to using it as part of the project. This may have aided in improving the results. However, this was not feasible due to time constraints. In addition, with more time, it would have been possible to fine tune the CNN and fix any issues within the environment and the models. Hence why it was concluded earlier that this project has successfully shown a proof of concept that can be improved in the future.

7.5 Future Work

Given more time to work on this project, there would be several areas upon which more work would be focussed. These are as follows:

- Training and refining the current CNN for longer, to see if that improves the results.
- Improving the simulation, including fixing the models that do not spawn in the correct location or fall over when spawned.

- Training the CNN on different datasets, such as the Jacquard dataset discussed in the initial analysis, to see if this improves the results.
- The ACRONYM also comes with bad grasp data, this could be integrated into the CNN, which could help the algorithm distinguish between 'good' and 'bad' grasps.

Developing these areas would take the project from being a proof of concept, to being a fully functional example of a successful system.

7.6 Summary

When looking back over the project as a whole, it is clear that there are both positives and negatives to draw from the process and there are areas which would be done differently if repeated. Furthermore, the project was definitely more complicated than originally anticipated.

The strength of the project was the use of the FDD methodology and iterative development. This structure really helped manage the time and functionality, enabling more concentration on the actual design and implementation of the project. However, more testing should have been done throughout the course of the process.

The biggest weakness of this project is the final results produced by the experiments; ways to improve this have already been discussed. Overall, I feel that the project was a success and delivered what it set out to prove. The software produced was also to a high quality and made good use of available libraries. The project as a whole has also enabled me to develop many skills; including: the use of an agile methodology to manage a project, and the use of Tensorflow to build and train a custom CNN.

8 Annotated Bibliography

[1] 'Spot | Boston Dynamics'. https://www.bostondynamics.com/spot (accessed May 05, 2021).

This reference is to provide an example of the current use of grasping in robotics.

[2] S. Caldera, A. Rassau, and D. Chai, 'Review of Deep Learning Methods in Robotic Grasp Detection', *Multimodal Technol. Interact.*, vol. 2, no. 3, Art. no. 3, Sep. 2018, doi: 10.3390/mti2030057.

[3] I. Lenz, H. Lee, and A. Saxena, 'Deep Learning for Detecting Robotic Grasps', *ArXiv13013592 Cs*, Aug. 2014, Accessed: Feb. 04, 2021. [Online]. Available: http://arxiv.org/abs/1301.3592

[4] S. Sapora, E. Johns, and A. Cully, 'Grasp Quality Deep Neural Networks for Robotic Object Grasping', p. 116.

[5] U. Asif, J. Tang, and S. Harrer, 'GraspNet: An Efficient Convolutional Neural Network for Real-time Grasp Detection for Low-powered Devices'.

[6] H. Cao, G. Chen, Z. Li, J. Lin, and A. Knoll, 'Lightweight Convolutional Neural Network with Gaussian-based Grasping Representation for Robotic Grasping Detection', *ArXiv210110226 Cs*, Jan. 2021, Accessed: Feb. 04, 2021. [Online]. Available: http://arxiv.org/abs/2101.10226

[7] J. Watson, J. Hughes, and F. Iida, 'Real-World, Real-Time Robotic Grasping with Convolutional Neural Networks', Jul. 2017, pp. 617–626. doi: 10.1007/978-3-319-64107-2_50.

[8] S. Kumra and C. Kanan, 'Robotic Grasp Detection using Deep Convolutional Neural Networks', *ArXiv161108036 Cs*, Jul. 2017, Accessed: Feb. 04, 2021. [Online]. Available: http://arxiv.org/abs/1611.08036

[9] P. Schmidt, N. Vahrenkamp, M. Wächter, and T. Asfour, 'Grasping of Unknown Objects Using Deep Convolutional Neural Networks Based on Depth Images', in *2018 IEEE International Conference on Robotics and Automation (ICRA)*, May 2018, pp. 6831–6838. doi: 10.1109/ICRA.2018.8463204.

[2]-[9] are papers that were read as background research into grasping and using CNNs for grasp detection. They helped get a picture of what was required for the project.

[10] 'Cornell University. Robot Learning Lab: Learning to Grasp.' http://pr.cs.cornell.edu/grasping/rect_data/data.php (accessed Feb. 06, 2021).

[11] A. Depierre, E. Dellandréa, and L. Chen, 'Jacquard: A Large Scale Dataset for Robotic Grasp Detection', *ArXiv180311469 Cs*, Sep. 2018, Accessed: Feb. 04, 2021. [Online]. Available: http://arxiv.org/abs/1803.11469

[10] and [11] are examples of grasp datasets that were researched as part of initial analysis.

[12] C. Eppner, A. Mousavian, and D. Fox, ‘ACRONYM: A Large-Scale Grasp Dataset Based on Simulation’, *ArXiv201109584 Cs*, Nov. 2020, Accessed: Feb. 04, 2021. [Online]. Available: http://arxiv.org/abs/2011.09584

[13] ‘ACRONYM: A Large-Scale Grasp Dataset Based on Simulation’. https://sites.google.com/nvidia.com/graspdataset (accessed Feb. 04, 2021).

[12] and [13] are references to the ACRONYM grasp dataset, which was utilised for this project.

[14] A. X. Chang *et al.*, ‘ShapeNet: An Information-Rich 3D Model Repository’, *ArXiv151203012 Cs*, Dec. 2015, Accessed: May 12, 2021. [Online]. Available: http://arxiv.org/abs/1512.03012

[15] M. Savva, A. X. Chang, and P. Hanrahan, ‘Semantically-Enriched 3D Models for Common-sense Knowledge’, *CVPR 2015 Workshop Funct. Phys. Intentionality Causality*, 2015.

[14] and [15] are the references to the ShapeNet and ShapeNetSem papers, from which the project’s model database was downloaded.

[16] ‘ROS.org | Powering the world’s robots’. https://www.ros.org/ (accessed Feb. 06, 2021).

ROS was used to implement the experiments on the robot conducted as part of this project.

[17] ‘Gazebo’. http://gazebosim.org/ (accessed Feb. 07, 2021).

Gazebo is the simulator that was used in this project.

[18] ‘OpenRAVE | Welcome to Open Robotics Automation Virtual Environment | OpenRAVE Documentation’. http://openrave.org/docs/latest_stable/ (accessed Feb. 04, 2021).

[19] ‘GraspIt!’ https://graspit-simulator.github.io/ (accessed Feb. 04, 2021).

[18] and [19] were other simulators that were looked at in the initial analysis.

[20] ‘C++ documentation — DevDocs’. https://devdocs.io/cpp/ (accessed May 05, 2021).

The C++ documentation. This was used to aid the development of the C++ software.

[21] ‘Python’, *Python.org*. https://www.python.org/ (accessed Feb. 07, 2021).

The Python documentation was used to aid the development of the Python scripts.

[22] ‘Franka Panda Arm’, *Franka Emika*. https://www.franka.de/technology (accessed Feb. 07, 2021).

The Franka Panda Arm was the initial arm that was going to be used to simulate the grasping.

[23] ‘Autonomous Mobile Robots That Improve Productivity | Fetch Robotics’. https://fetchrobotics.com/ (accessed May 05, 2021).

The Fetch robot was used to simulate the grasping in the environment.

[24] ‘MoveIt Motion Planning Framework’. https://moveit.ros.org/ (accessed Feb. 07, 2021).

MoveIt is the framework that is used to move the robot arm to different positions and grasp objects.

[25] ‘TensorFlow’, *TensorFlow*. https://www.tensorflow.org/ (accessed Feb. 04, 2021).

Tensorflow was used to build the convolutional neural network.

[26] ‘PyTorch’. https://www.pytorch.org (accessed May 05, 2021).

PyTorch was another option for developing the CNN.

[27] R. Draelos, ‘The History of Convolutional Neural Networks’, *Glass Box*, Apr. 13, 2019. https://glassboxmedicine.com/2019/04/13/a-short-history-of-convolutional-neural-networks/ (accessed May 02, 2021).

This article was utilised to research about the history of CNNs.

[28] ‘Simple cell’, *Wikipedia*. Apr. 30, 2021. Accessed: May 06, 2021. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Simple_cell&oldid=1020699956

Image of a simple cell, used to illustrate the text about the visual cortex.

[29] K. Fukushima, ‘Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position’, *Biol. Cybern.*, vol. 36, no. 4, pp. 193–202, Apr. 1980, doi: 10.1007/BF00344251.

One of the first examples of what is classified as a CNN.

[30] A. Krizhevsky, I. Sutskever, and G. E. Hinton, ‘ImageNet classification with deep convolutional neural networks’, *Commun. ACM*, vol. 60, no. 6, pp. 84–90, May 2017, doi: 10.1145/3065386.

ImageNet/AlexNet is an example of a more modern CNN used to classify images.

[31] S. Bansari, ‘Introduction to how CNNs Work’, *Medium*, Apr. 30, 2019. https://medium.datadriveninvestor.com/introduction-to-how-cnns-work-77e0e4cde99b (accessed May 05, 2021).

[32] Prabhu, 'Understanding of Convolutional Neural Network (CNN) — Deep Learning', *Medium*, Nov. 21, 2019. https://medium.com/@RaghavPrabhu/understanding-of-convolutional-neural-network-cnn-deep-learning-99760835f148 (accessed May 05, 2021).

[31] and [32] were articles used to better understand the inner workings of CNNs. The images were used to illustrate points mentioned in the body of the report.

[33] J. Wu, 'Introduction to Convolutional Neural Networks', [Online]. Available: https://cs.nju.edu.cn/wujx/paper/CNN.pdf (accessed May 05, 2021)

This paper is here in case more information is required on CNNs.

[34] 'SCW', *Supercomputing Wales*. https://www.supercomputing.wales/ (accessed May 05, 2021).

SCW was utilised in this project as a means to train the network faster.

[35] 'PyCharm: the Python IDE for Professional Developers by JetBrains', *JetBrains*. https://www.jetbrains.com/pycharm/ (accessed May 08, 2021).

PyCharm was the IDE used to develop the Python 3 scripts for pre-processing and Tensorflow.

[36] 'Visual Studio Code - Code Editing. Redefined'. https://code.visualstudio.com/ (accessed May 08, 2021).

Visual Studio Code was the IDE used to develop the ROS C++ and Python 2 code.

[37] *NVlabs/acronym*. NVIDIA Research Projects, 2021. Accessed: May 05, 2021. [Online]. Available: https://github.com/NVlabs/acronym

The GitHub repository that holds the ACRONYM dataset scripts that were modified and utilised to generate the dataset in this project.

[38] J. Huang, H. Su, and L. Guibas, 'Robust Watertight Manifold Surface Generation Method for ShapeNet Models', *ArXiv180201698 Cs*, Feb. 2018, Accessed: May 05, 2021. [Online]. Available: http://arxiv.org/abs/1802.01698

This is the paper in connection to the library that was used to create watertight ShapeNet models so they could be utilised by the ACRONYM scripts.

[39] 'Anaconda | Individual Edition', *Anaconda*. https://www.anaconda.com/products/individual (accessed May 05, 2021).

Anaconda was used to get the most up to date libraries on SCW.

[40] 'OpenCV', *OpenCV*. https://opencv.org/ (accessed May 08, 2021).

Some of the OpenCv libraries were used as part of the software to run the experiments.

[41] ‘pandas - Python Data Analysis Library’. https://pandas.pydata.org/ (accessed May 08, 2021).

Pandas was the library used to organise the datasets in Python for this project.

[42] ‘NumPy’. https://numpy.org/ (accessed May 05, 2021).

Numpy was used to manipulate some of the data in the project, particularly for use with Tensorflow.

[43] ‘Full pytest documentation — pytest documentation’. https://docs.pytest.org/en/6.2.x/contents.html (accessed May 08, 2021).

PyTest would have been used if unit tests were implemented.

9 Appendices

A. Third-Party Code and Libraries

ROS – This project is going to be centred around the Robot Operating System (ROS) [16]. It will be used to simulate the robot and run the grasping experiments in the Gazebo simulator [17].

Tensorflow – One of the main components of this project is the Convolutional Neural Network. This will be built, trained, and saved using the Tensorflow [25] library.

MoveIt – The main aim of the project is to grasp objects in the simulator. The MoveIt framework [24] enable the control of the simulated manipulator within ROS.

Fetch libraries – ROS comes with many libraries that deal with the simulation and interaction with specific robots. One such library is provided for the Fetch robot [23] which is used in this project.

Manifold [38] – This library provides the service of creating watertight versions of the ShapeNetSem [14-15] model database. This is done so that the ACRONYM scripts can read the files to generate the dataset.

ACRONYM – This project is utilising the ACRONYM [12-13] grasp dataset to train the CNN to learn grasp patterns. It contains 17.7M grasps on 8,872 objects generated in a physics simulator.

Pandas and Numpy – These libraries were used to manipulate the data in the project. Pandas [41] was used to build the initial dataset and the save the results. Numpy [42] was used to manipulate the data to and from the Tensorflow model.

OpenCv – An OpenCv [40] Bridge was used in the ROS pipeline to take the ROS Image message from the camera and pass it through the trained Tensorflow model to predict a grasp for the current object.

B. Ethics Submission - Application Number: 18838

AU Status
Undergraduate or PG Taught
Your aber.ac.uk email address
olt13@aber.ac.uk
Full Name
Oliver Thomas
Please enter the name of the person responsible for reviewing your assessment.
Neil Taylor
Please enter the aber.ac.uk email address of the person responsible for reviewing your assessment
nst@aber.ac.uk
Supervisor or Institute Director of Research Department
cs
Module code (Only enter if you have been asked to do so)
CS39440
Proposed Study Title
Deep learning for robotic grasp detection
Proposed Start Date
25 January 2021
Proposed Completion Date
1 June 2021
Are you conducting a quantitative or qualitative research project?
Mixed Methods
Does your research require external ethical approval under the Health Research?
Authority?
No
Does your research involve animals?
No
Does your research involve human participants?
No
Are you completing this form for your own research?
Yes
Does your research involve human participants?
No
Institute
IMPACS
Please provide a brief summary of your project (150 word max)
The project aims to apply deep learning to robotic grasp detection; using a deep convolution neural network fed with RGB-D (depth camera) images in order to predict successful grasps for novel objects. This will then be applied in simulation.
Where appropriate, do you have consent for the publication, reproduction or use of any unpublished material?
Yes
Will appropriate measures be put in place for the secure and confidential storage of data?

Yes
Does the research pose more than minimal and predictable risk to the researcher?
Not applicable
Will you be travelling, as a foreign national, in to any areas that the UK Foreign and Commonwealth Office advise against travel to?
No
Please include any further relevant information for this section here:
Is your research study related to COVID-19?
No
If you are to be working alone with vulnerable people or children, you may need a DBS (CRB) check. Tick to confirm that you will ensure you comply with this requirement should you identify that you require one.
Yes
Declaration: Please tick to confirm that you have completed this form to the best of your knowledge and that you will inform your department should the proposal significantly change.
Yes
Please include any further relevant information for this section here:

C. Tensorflow Model

An example of building a Tensorflow CNN model in Python using the functional API.

```
def build_model():
    inputs = keras.Input(shape=(400, 400, 3))
    x = layers.Conv2D(32, 1)(inputs)
    x = layers.BatchNormalization()(x)
    x = keras.activations.relu(x)
    x = layers.MaxPooling2D()(x)
    x = layers.Conv2D(64, 1)(x)
    x = layers.BatchNormalization()(x)
    x = keras.activations.relu(x)
    x = layers.MaxPooling2D()(x)
    x = layers.Conv2D(128, 3)(x)
    x = layers.BatchNormalization()(x)
    x = keras.activations.relu(x)
    x = layers.MaxPooling2D()(x)
    x = layers.Conv2D(256, 3)(x)
    x = layers.BatchNormalization()(x)
    x = keras.activations.relu(x)
    x = layers.Flatten()(x)
    x = layers.Dropout(0.35)(x)
    x = layers.Dense(128, activation='relu')(x)
    x = layers.Dense(64, activation='relu')(x)
    x = layers.Dense(32, activation='relu')(x)
    outputs = layers.Dense(6, activation='linear')(x)
    model = keras.Model(inputs=inputs, outputs=outputs)
    return model
```

D. Custom Training and Loss Functions

The custom training loop that was designed so that the custom model loss metric could be applied in training.

```
def training_loop(model, dataset):
    optimizer = keras.optimizers.RMSprop(learning_rate=0.001)
    loss_obj = keras.losses.MeanAbsoluteError()
    train_loss_results = []

    epochs = 10 + 1
    widgets = [progressbar.SimpleProgress(), '   ', progressbar.Bar(), '   ', 
progressbar.ETA()]
    b = progressbar.ProgressBar(len(dataset))
    for epoch in range(1, epochs):
        epoch_loss_avg = tf.keras.metrics.Mean()

        print("Epoch: {e}/{n}".format(e=epoch, n=epochs-1))
        b.start()
        for index, (x, y) in enumerate(dataset):
            with tf.GradientTape() as tape:
                inp = tf.expand_dims(x[0], axis=0)
                loss = grasp_loss(loss_obj, y, model(x))
                # print(loss)
            grads = tape.gradient(loss, model.trainable_variables)

            optimizer.apply_gradients(zip(grads, model.trainable_variables))
            epoch_loss_avg.update_state(loss)
            b.update(index)

        print("Epoch loss: {l}".format(l=epoch_loss_avg.result()))
        train_loss_results.append(epoch_loss_avg.result())
        b.finish()

    model.save_weights("custom_model_weights.h5")
    plt.plot(train_loss_results)
    plt.xlabel('Epoch')
    plt.ylabel('MAE to closest grasp')
    plt.show()
```

The custom loss function that is utilised in the training algorithm.

```
def grasp_loss(loss_obj, labels, y_pred):
    losses = []
    for y in labels[:1]:
        loss = y - y_pred
        losses.append(sum(loss))

    closest = min(losses)
    return loss_obj(closest, y_pred)
```

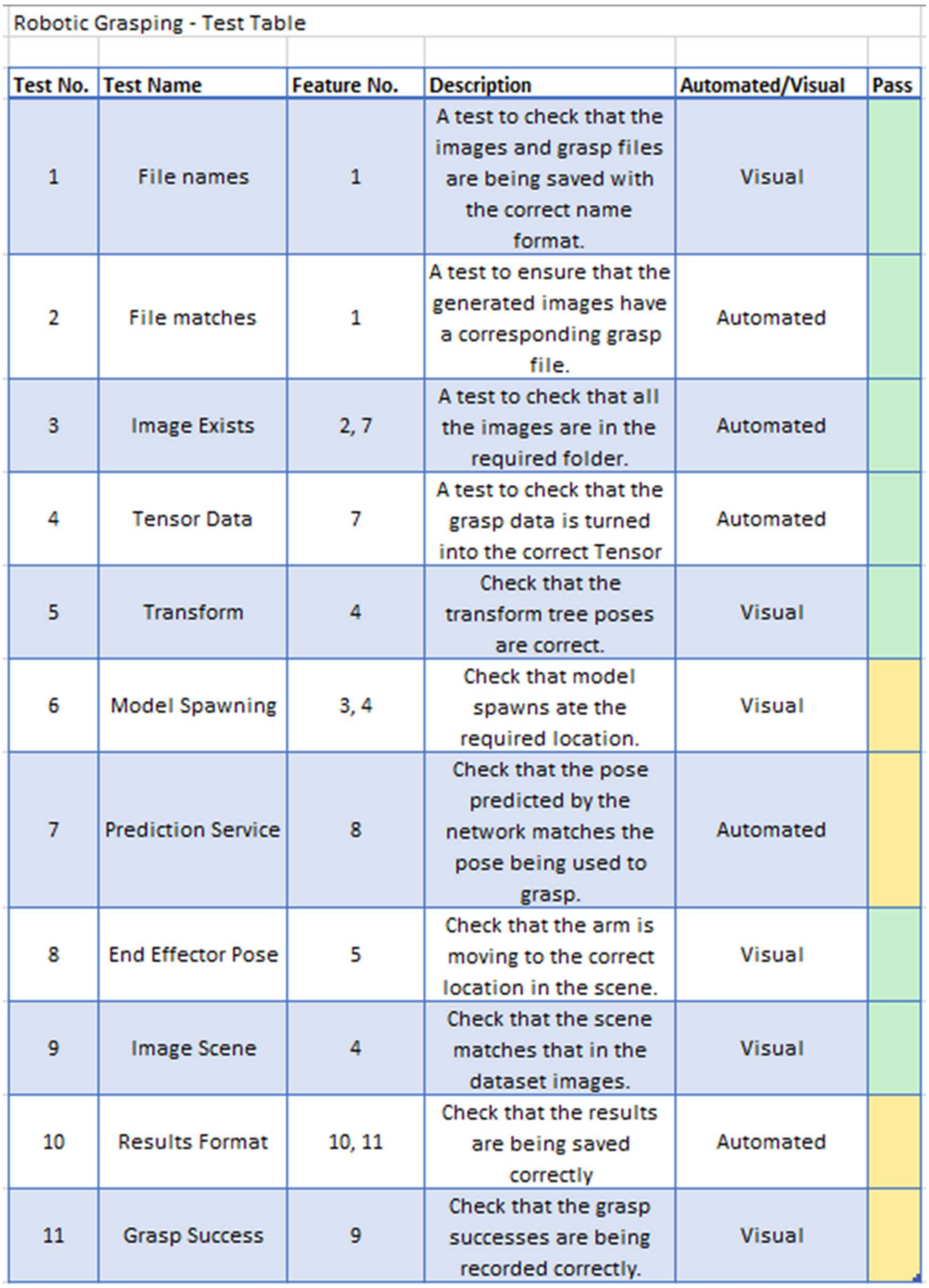
E. SCW SBatch File

The batch file that is required to run programs on the SCW server. This example defines all the necessary SCW variables, sets up the running environment using Anaconda, and then runs the Python script containing the code to train the CNN.

```
#!/bin/bash --login

###

#SBATCH --partition=gpu
#SBATCH --gres=gpu:2

#SBATCH --job-name=TRAINING

#SBATCH --output=grasp.out.%J

#SBATCH --error=grasp.err.%J

#maximum job time in D-HH:MM
#SBATCH --time=1-00:00

#SBATCH --ntasks=1

#SBATCH --mem-per-cpu=8000

#SBATCH --ntasks-per-node=1

#SBATCH -A scw1780

###

#now run normal batch commands

module load anaconda/3
conda activate ../.conda/envs/cenv

python3 model_training.py
```

F. Testing Table and Example

The ongoing testing table for the project, as of week 12. It can be seen that some tests are passing, but other features still need work.



Robotic Grasping - Test Table

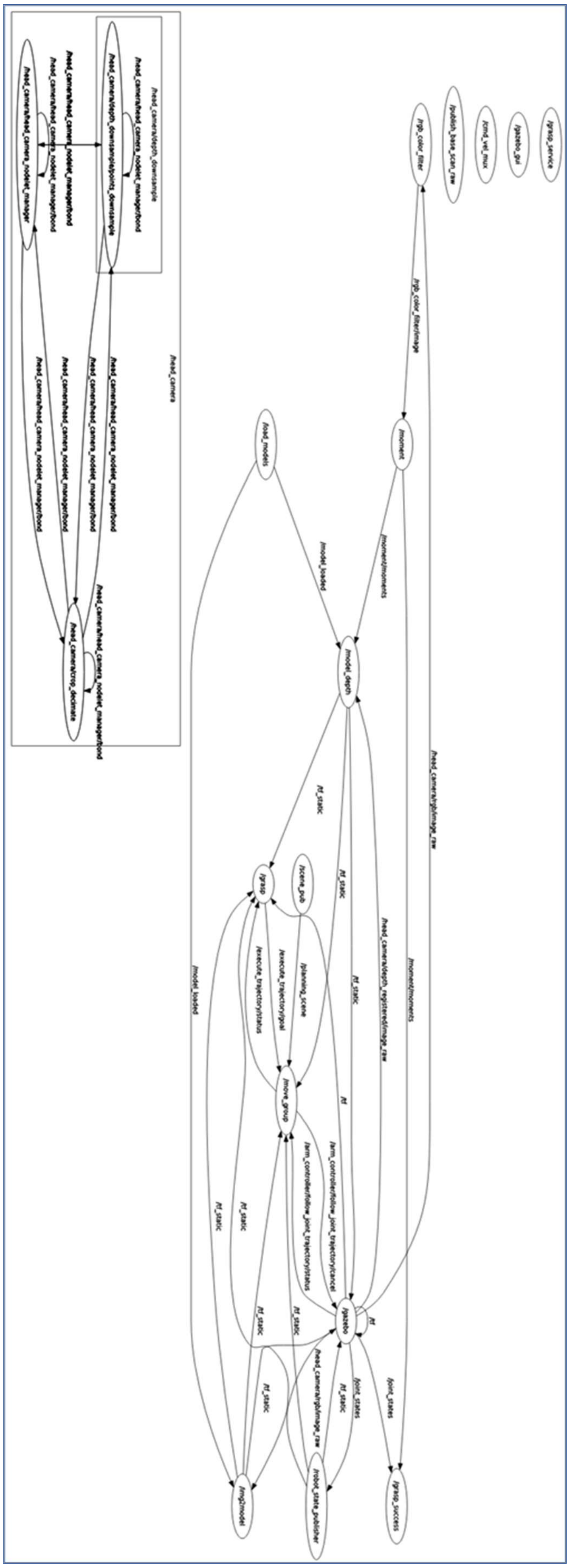
Test No.	Test Name	Feature No.	Description	Automated/Visual	Pass
1	File names	1	A test to check that the images and grasp files are being saved with the correct name format.	Visual	🟩
2	File matches	1	A test to ensure that the generated images have a corresponding grasp file.	Automated	🟩
3	Image Exists	2, 7	A test to check that all the images are in the required folder.	Automated	🟩
4	Tensor Data	7	A test to check that the grasp data is turned into the correct Tensor	Automated	🟩
5	Transform	4	Check that the transform tree poses are correct.	Visual	🟩
6	Model Spawning	3, 4	Check that model spawns ate the required location.	Visual	🟨
7	Prediction Service	8	Check that the pose predicted by the network matches the pose being used to grasp.	Automated	🟨
8	End Effector Pose	5	Check that the arm is moving to the correct location in the scene.	Visual	🟩
9	Image Scene	4	Check that the scene matches that in the dataset images.	Visual	🟩
10	Results Format	10, 11	Check that the results are being saved correctly	Automated	🟨
11	Grasp Success	9	Check that the grasp successes are being recorded correctly.	Visual	🟨

This is a section from the script that generates the dataset to be uploaded to SCW. It shows the test done on the files to make sure all the images are in place before running the training to avoid any errors that would stop the process.

```
df_images = df['image'].values
df_images = set(df_images)
....
# Test for missing images
print(len(df['image']))
bar = progressbar.ProgressBar(max_value=len(df_images))
done_images = [f.name for f in os.scandir(new_loc)]
done_images = set(done_images)
index = 0
for img in df_images:
    bar.update(index + 1)
    index += 1
    if img in done_images:
        continue
```

G. ROS Graph

This is the ROS rqt graph while the CNN model experiments are running. It shows the connections between each of the nodes and some of the external libraries/programs in use.


/grasp_service
/gazebo_gui
/cmd_vel_mux
/pblish_base_scan_raw
/rgb_color_filter
/rgb_color_filter/image
/moment
/moment/moments
/load_models
/model_loaded
/model_depth
/head_camera
/head_camera/depth_downsample
/head_camera/head_camera_nodelet_manager
/head_camera/depth_downsample/points_downsample
/head_camera/head_camera_nodelet_manager/bond
/head_camera/crop_decimate
/tf_static
/tf
/grasp
/scene_pub
/planning_scene
/execute_trajectory/goal
/execute_trajectory/status
/move_group
/arm_controller/follow_joint_trajectory/cancel
/arm_controller/follow_joint_trajectory/status
/head_camera/depth_registered/image_raw
/head_camera/rgb/image_raw
/gazebo
/joint_states
/grasp_success
/robot_state_publisher
/img2model