LONDON'S GLOBAL UNIVERSITY

Robust Robotic Grasping Utilising Touch Sensing

COMP0029_ZZJN7[1]

BSc Computer Science

Supervisors: Prof. Marc Deisenroth, Dr. Yasemin Bekiroglu

Submission date: May 11, 2023

[1]**Disclaimer:** This report is submitted as part requirement for the BSc degree in Computer Science at UCL. It is substantially the result of my own work except where explicitly indicated in the text. The report will be distributed to the internal and external examiners, but thereafter may not be copied or distributed except with permission from the author.

Abstract

Robotic grasp synthesis has been studied extensively as robotic grasping skills have a significant impact on the success of subsequent manipulation tasks. Various approaches have been proposed for robotic grasp planning, with different assumptions regarding the available information about the type of objects in question (known, unknown, familiar). These approaches range from heuristic rules, and designing simplifying hand models, to complete end-to-end systems inferring grasp parameters from raw data.

However, most of these approaches do not address robustness in grasping, which refers to the ability of a robot to perform a grasping task consistently and accurately even in the case of unexpected disturbances or large degrees of errors in perception. There are several fundamental problems that need to be addressed to achieve better robustness in grasping tasks. These include mainly dealing with uncertainties in sensing, actuation and perceptual data.

In this project, we study how to build a robust learning framework that can be employed to construct robust grasp configurations using multi-modal data, e.g. tactile, and visual. The project addresses the following main issues with the robotic grasping systems: a) balancing the trade-off between data representation and data dimensionality; b) analysing the modelling effects of different modalities, e.g. tactile and visual, and features to capture the underlying characteristics of the overall grasping process; c) a correction policy that relies on assessing grasp success before further manipulation using perceptual data, to choose the right grasping configuration.

Acknowledgements

I would like to extend my sincere gratitude to all those who have contributed to the successful completion of this project. I am sincerely grateful to all those who have contributed to this project, and I am honoured to have had the opportunity to work with such incredible individuals. Without their support and guidance, this accomplishment would not have been possible.

First and foremost, I would like to express my deepest appreciation to my supervisors, Prof. Marc Deisenroth and Dr. Yasemin Bekiroglu, for their invaluable guidance, expert knowledge, and unwavering support throughout this research journey. Despite facing a steep learning curve in familiarising myself with the field, their mentorship has greatly expanded my understanding and revealed the immense potential of robotic grasping in real-life applications.

I would also like to acknowledge the assistance of Sicelukwanda Zwane from the Statistical Machine Learning Group, who provided valuable guidance on practical aspects of the project. Our stimulating exchanges allowed me to benefit from his expertise in simulation environments and integrating different physical robot components. Additionally, I am grateful to Jeffery Wei, a Master's project student also working with the SML group, who collaborated intensively with me during the early stages of the project, helping to develop a Pybullet simulation that forms the foundation of my work.

Furthermore, I would like to thank all my friends and colleagues who have provided me with moral support and engaging discussions during this intense research period. Your presence has made the journey more enjoyable and has created lasting memories.

Lastly, I want to express my heartfelt appreciation to my parents for their unwavering love, belief in my abilities, and endless encouragement. Their presence and the opportunities they have provided me have been instrumental in my academic journey.

Contents

Chapter 1

Introduction

1.1 Motivation

Humans are able to grasp objects seemingly intuitively. We can efficiently identify and reach for objects, adjust our finger placements and optimally balance contact forces by coordinating our wrists, arms, and shoulders when we approach and lift objects. Our sense of touch plays a major role, providing us with information about the object's size, shape, texture, weight, and other physical properties. This information is processed by sensorimotor and cognitive functions [8] in our brains, giving us instructions on how to adjust our grasp pose and apply the appropriate amount of force needed to pick up the object. Without the sense of touch, grasping objects would be much more difficult and less efficient.

For several decades, researchers have delved into the domain of robotic grasping to equip real-life robots with the ability to grasp objects. Various techniques have been introduced in this field, such as the incorporation of touch sensing [16] using tactile sensors [29, 31] and/or visual perception [7]. These methods enable robots to determine grasp configurations that meet a set of criteria that are relevant to the grasping task, which is known as grasp synthesis [5]. However, obtaining appropriate grasp configurations has proven to be a challenging task since it requires identifying an infinite set of grasp candidates given a n-dimensional pose of a robotic arm and gripper joints, where n is the number of movable joints. Thus, most approaches to grasp synthesis are categorised into two main groups [5, 36]: analytical and data-driven methods. Analytical methods treat grasp synthesis as constrained optimisation problems that measure dexterous, stable, and dynamic behaviour properties using heuristics. In contrast, data-driven methods rely on sampling grasp candidates for an object and ranking them based on a pre-determined metric [5].

In recent years, the combination of physical simulators and machine learning has led

to significant advancements in the field of robotic grasping. One of the key challenges in robotic grasping is developing algorithms that can handle the complexity and variability of the real world. Physical simulators such as Pybullet [13] provide a valuable tool for generating large amounts of training data, allowing machine learning algorithms to learn from simulated experiences before being deployed on physical robots with minimal additional tuning [4]. Researchers have also utilised physical simulators to train robotic grasping policies [9] and transfer them to the real world, and have proven that data-driven approaches outperform classical methods [43].

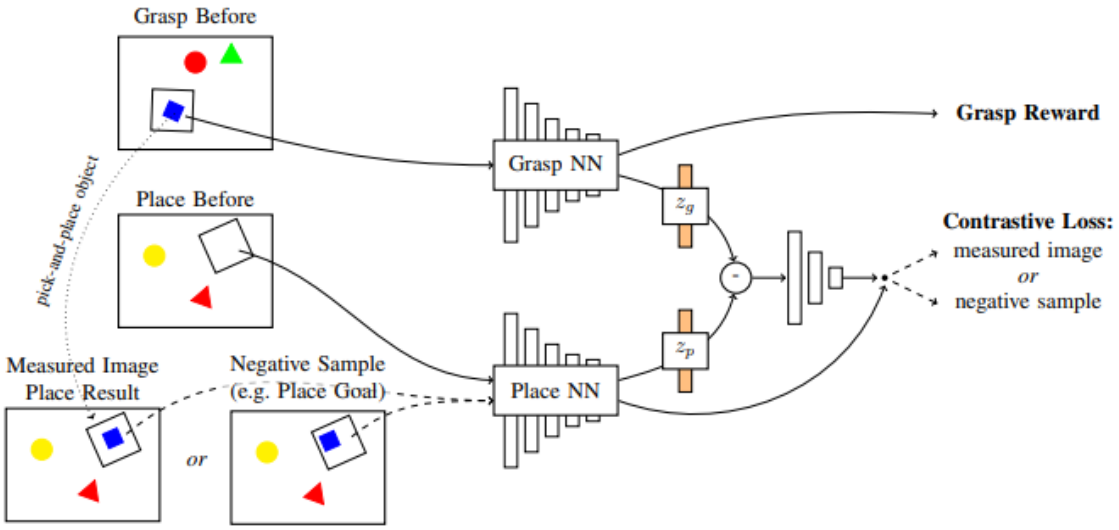


Figure 1.1: Berscheid et al.: Self-supervised learning for Precise Pick-and-place without Object Model [4]: A purely data-driven approach to generating data for self-supervised learning for robotic grasping using neural networks and object recognition

However, little work has been done on increasing robustness in grasping systems that utilise multi-modal (tactile and visual) data. Robustness in robotic grasping is a crucial property for ensuring effective and reliable manipulation of objects in a wide range of scenarios. It refers to the ability of a robotic grasping system to perform successful grasps despite uncertainties and variations in the environment and object properties (geometric, physical). In addition, ideally our robotic grasping system should be trained on relatively simple models and datasets while preserving as much performance as possible.

One of the primary challenges in robotic grasping is the wide variety of shapes and sizes that objects can come in. This can make it difficult for a robotic grasping system to determine the best approach for grasping, as well as to adjust its grasp strategy based on the object's properties. A robust system should be able to handle these variations and adjust its grasp strategy accordingly to ensure that the object is firmly held during manipulation. Another challenge is posed by uncertainties in the object's pose. Objects can be placed in various orientations and positions, making it challenging for a robot to determine the optimal grasping location. A robust system should be able to estimate

the object's pose accurately and choose an appropriate grasp location to ensure that the object is grasped correctly. Furthermore, the environment in which a robot operates can change unpredictably, with variations in lighting conditions, surface textures, and other factors affecting the ability of a robotic grasping system to perform well. A robust system should be able to adapt to these changes and adjust its grasping strategy accordingly.

We aim to take the best of both analytic [23] and data-driven methods [4, 34] to develop robust and scalable grasping methods. Robustness in a robotic grasping system refers to its versatility of accurately recognising and grasping objects with varying geometric features (width, depth, curvature, etc) and its ability to successfully grasp the objects. Robustness is increasingly becoming a vital factor in the study of grasp stability and quality, however, incorporating robustness makes it practically difficult to design a single gripper that can effectively grasp all objects. As a result, researchers and engineers have developed dedicated grippers tailored to specific grasping tasks or types of objects.

Developing a robust and efficient grasping system presents several challenges. One major hurdle is the incorporation of touch sensing into robotic grasping, which is hindered by hardware limitations such as sensor sensitivity and cost, as well as difficulties in integrating tactile inputs into standard control schemes [7]. Consequently, most robotic grasping research has focused on vision and depth as the primary input modalities [7, 16].

However, vision-based grasping approaches have limitations in measuring and reacting to ongoing contact forces, which hinders the full potential benefits of interaction. As a result, these approaches mostly rely on pre-selecting grasp configurations, such as location, orientation, and forces, before making contact with the object. Addressing these limitations and integrating tactile feedback could significantly improve robotic grasping, leading to more efficient and effective interactions between robots and the physical world. Despite the advantages of pre-selecting grasp configurations, it is important to note that this approach may not fully address object variations and results in limited generalisations.

Another challenge is developing a concrete strategy for improving grasp quality and planning while conserving the efficiency of the overall system. Such strategies can be summarised into three types: (1) choice of object representation, where coarse approximations of the underlying true shape of an object can simplify the generation of new grasps [16] [19]; (2) use of local symmetry properties to capture key geometric features of the object to generate promising grasp candidates [16]; (3) optimising shape modelling where target objects are parameterised using smooth, differentiable functions from point clouds using spectral analysis [16].

1.2 Project Aims

Having gained a comprehensive understanding of the previously discussed optimisations and strategies, our research project focuses on the development and testing of a robust robotic system that can learn to pick up an object with simple geometry using a two-finger hand. The project will take a learning-based approach to grasping through, for example, Bayesian optimisation [33, 17]. The learning-based approach should be compared with a baseline approach from the related literature (e.g. [33, 14, 6]) for evaluation, for example, a simple regression classifier to differentiate between good and bad grasps.

1.3 Project Objectives

This project is concerned with the development of a complete simulation with supporting features for data collection and analysis. Therefore, the aims of this project are split according to our system requirements, which we describe in detail below. These objectives are expected to be completed throughout the academic year:

1. Setting up a simulation environment, e.g. PyBullet [13] or NVIDIA Isaac.
2. Creating a data collection pipeline for sensory data (e.g. visual and force/torque readings) via the simulator.
3. Applying basic simulation functionalities: position control and vision sensing on the robot.
4. Implement and test baseline (e.g. [6], and a basic approach such as executing predefined grasps per object model given object pose).
5. Build the learning framework:
   (a) Learning a grasp model encoding relevant grasp parameters given train and test object sets and available sensory data
   (b) Grasping with two fingers based on the learned model

1.4 Project Approach

This section discusses the important phases of the project and the approach taken at each phase. Technologies required for completing a particular phase will be documented and justified.

1.4.1 Planning Phase - Design and Research

We first conduct a literature survey to explore and categorise research projects related to robotic grasping by data input, grasping method and the problem these projects ad-

dress. Simultaneously, background research was conducted on grasp learning [35] and robot manipulation to obtain a general understanding of the project's task.

1.4.2 Early Development Phase - Pybullet Simulation

This project is completely conducted via a simulation environment, including data collection, robot manipulation, experimentation and testing of our approaches. Therefore, the development of a Pybullet simulation was considered a top priority in the early phases of the project. The robot we used for our simulation consists of the following components:

1. Arm: UR5 Robot Arm Manipulator
2. Gripper: Robotiq 2-Finger Adaptive Robot Gripper 85
3. Tactile sensors: DIGIT tactile sensors [29] mounted onto each finger of the gripper

The Pybullet simulation should support the following features for our project:

1. grasping and manipulation of objects using a two-finger gripper
2. Real-time display of tactile data (depth and colour cameras)
3. A grasp planner and grasp execution implementation based on inverse kinematics given a grasp configuration
4. A random end effector pose generator for sampling varying grasp poses from manually-selected seed poses
5. A tactile data collection pipeline using the randomly-sampled end effector grasp poses

This will be further discussed in Section 3.1.

1.4.3 Baseline Development Phase

This phase focuses on determining the optimal dataset representation for our collected multi-modal data, consisting of tactile sensor readings, visual end effector poses relative to the arm, and the respective grasp outcomes of each end effector pose. In addition, we experiment with several feature engineering techniques, for example, dimensionality reduction using ConvNets and Principal Component Analysis (PCA).

1.4.4 Execution Phase - Learning Approach Implementation

Following the outcomes of our baseline approach where we identified the optimal feature representation of our dataset, we move on to develop our learning-based approach which is trained on three primitive object types. In addition to tactile sensor readings and visual data (randomly-generated end effector poses), we include the object features for each

object class in our dataset, and train this dataset on a multi-layer perceptron (MLP). Finally, we conduct several experiments, including (1) training our MLP using various dataset segmentations; and (2) validating our MLP on unseen objects that comprise similar geometric features to that of the primitive objects that our MLP was trained on.

1.4.5 Documentation Phase

The final phase of the project involves the completion of the project report, writing documentation on setting up and using the simulation, as well as including a brief description of the methodology behind our baseline and learning-based approaches which are implemented in Python.

1.5 Report Structure

Chapter 2 gives a brief overview of the background of robotic grasping and relevant literature on the project. We investigate common grasp representations and grasp quality evaluation techniques, and analyse various data representations of multi-modal data (tactile, sensory, temporal) and policy-learning approaches.

Chapter 3 documents the system architecture for the project and the development of a Pybullet simulation, including input components to enable data collection and simulation of our robot setup.

Chapter 4 documents the development and testing of our baseline approach using a binary logistic regression classifier.

Chapter 5 documents the development and testing of our proposed approach using a multi-layer perceptron (MLP) model that attempts to infer good grasp configurations on three primitive object classes from random hand poses, trained on data collected from the Pybullet simulation. The dataset includes tactile sensor readings, random hand poses and geometric features of the target objects.

Chapter 6 summarises the results and performances of the project, as well as makes suggestions for future work and improvement based on these findings.

Chapter 2

Background and Literature Review

2.1 Robotic Grasping

In the field of robotics, grasping is a fundamental yet challenging skill of robots which refers to the autonomous ability of a robot to grasp and move objects with its mechanical grippers or other end effectors [43]. It demands precise coordination between the visual perception of the surrounding environment and efficient grasp planning, as well as uncertainties in the perceptual data (robustness).

Robotic grasping is far away from being as intuitive, instinctive and successful as human grasping skills. There are many unknown factors that enable humans to flawlessly perform grasps when provided with visual and tactile information. However, with the advancements of deep learning and computer vision, semantic grasping is becoming the main focus and basis for autonomous robotic grasping systems [43], where a robot is trained on monocular images of a user-specified object type [25].

To successfully grasp an object, a robot needs to detect the object's location (through, for example, a wrist camera), orientation, and size accurately, estimate the best grasp point and orientation, and then execute the grasp with appropriate force and control to ensure a secure hold. The grasping process can be performed using various types of end effectors, including grippers, suction cups, or specialised tools designed for specific tasks.

Therefore, accurate and efficient grasp representation and planning are paramount to ensure a smooth grasping process when performing grasping tasks.

2.1.1 Object Representation

A grasp in robotic grasping refers to a specific configuration of a robotic hand or gripper that enables a firm grasp on an object. These representations are utilised in grasp planning algorithms and can be combined to create more complex grasp representations.

2D Object Images

Object detection through the use of 2D object images as visual input is a common first step in robotic grasping to identify relevant grasp parameters given a known object. Specifically, object recognition involves the input of object images into a machine learning model such as a Convolutional Neural Network (CNN), which learns to identify and classify the object depicted in the image. For instance, Chiu [11] applied this principle in developing a method for reconstructing 3D models of objects depicted in 2D images:

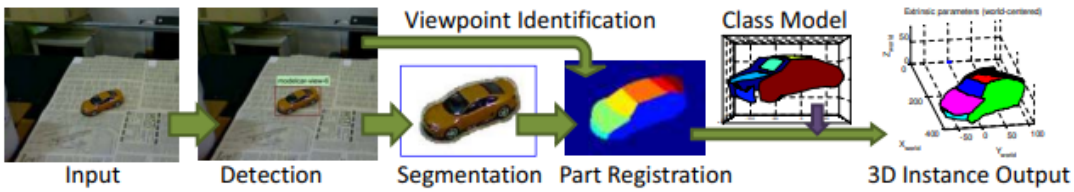


Figure 2.1: Chiu et al. [11]: Class-specific grasping of 3D objects from a single 2D image: In this work, object detection is applied to 2D images for the reconstruction of a 3D object model of the object in the images

In contrast, grasping unknown objects is a much more challenging task. Shannon et al. [38] proposed a scalable approach that efficiently generalises grasping policies on novel objects into real-world scenarios, without requiring any type of prior object awareness or task-specific training data.

Meshes

In computer graphics and object modelling, a (polygon) mesh is a 3-dimensional representation or rigid reconstruction of the surface of an object. Typically, the representation is a collection of vertices, edges and faces that define the object's shape. Faces usually consist of triangles an object's surface that is composed of vertices, edges, and faces. Grasps can be represented as a set of vertices on the mesh that corresponds to the location of the gripper's fingertips and palm.

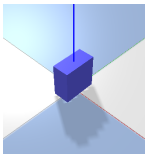


Figure 2.2: Example of an object mesh in our Pybullet simulation

Point Clouds

In the context of robotics, a point cloud is a discrete set of 3-dimensional Cartesian coordinates which collectively represents the surface of an object. The location of a gripper's fingertips can be represented as sets of points on the point cloud in order to annotate a grasp. In addition to object representations, point clouds can be merged for surface reconstruction (using, for example, Poisson surface reconstruction), which creates a mesh (or set of surfaces) that approximate the true shape of the target object.

2.1.2 Grasp Representations

In this section we will discuss about two widely-used grasp representations: grasp rectangles and 6-degrees-of-freedom end effector poses.

Grasp Rectangles

Grasp rectangles are used in robotic grasping to define the orientation and position of a robotic gripper when performing a grasp on an object. It is typically defined as a rectangle centred on a 2D image or point cloud of the target object and is aligned with the object's major axis. The grasp rectangle is used to represent the area on the object that the robotic gripper should contact when grasping the object.

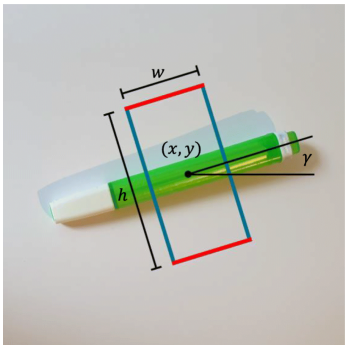


Figure 2.3: Kleeberger et al. [27]: A Survey on Learning-Based Robotic Grasping: Parameterisation for robotic grasp detection: Two values for the position, two for the size, and one for the orientation of the oriented rectangle. Red sides indicate the jaws of the gripper and blue the opening width

Figure 2.3 is an example of a grasp rectangle with width w, height w, centre of mass at coordinates (x, y). In addition, the term γ refers to the angle between the grasp rectangle's major axis and the horizontal axis of the image or coordinate system. The major axis of the grasp rectangle is the longer side of the rectangle, and it is aligned with the object's principal axis. The gamma angle specifies the amount of rotation that is required to align the grasp rectangle's major axis with the horizontal axis.

The grasp rectangle is used to guide the robotic gripper towards the object and ensures that the gripper can grasp the object securely without dropping it. Once the grasp rectangle is defined, the robotic system can use it to plan a grasping trajectory that will bring the gripper into contact with the object in the desired position and orientation. The system can then execute the grasping motion, picking up the object and moving it to a desired location.

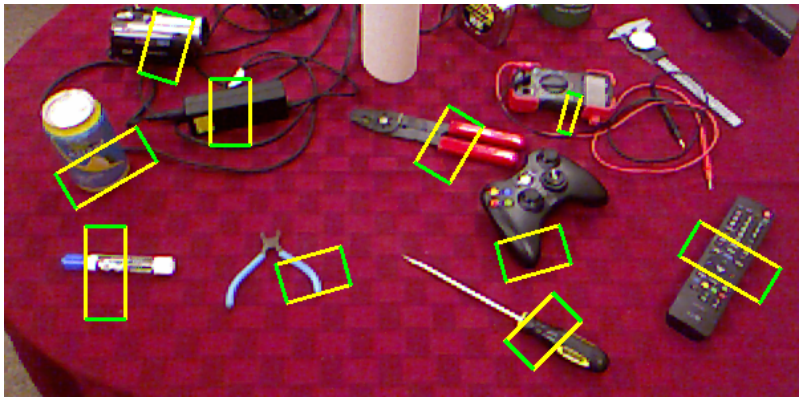


Figure 2.4: Lenz et al. [30]: Deep Learning for Detecting Robotic Grasps: Application of grasp rectangles corresponding to robotic grasps for objects in the scene

Figure 2.4 illustrates an example of applying grasp rectangles to objects in a scene. The objects are first detected using cameras or depth sensors, then a grasp rectangle is defined around the object based on its shape, size, and centre of mass.

6-Dimensional End Effector Pose

An end effector pose refers to the position and orientation of the end effector of a robot relative to the object being grasped. It is typically represented using a 6-degree-of-freedom (6-DOF) pose as follows:

$$S = (x, y, z, r_x, r_y, r_z) \tag{2.1}$$

where x, y, z are Cartesian coordinates describing the position of the end effector in a physical 3-dimensional plane, and r_x, r_y, r_z represents the orientation of the end effector as roll, pitch and yaw angles.

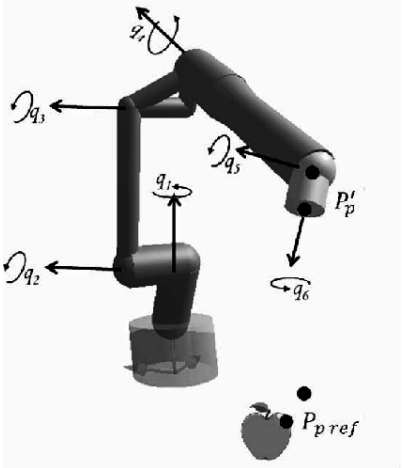


Figure 2.5: Jiménez et al. [20]: Robust Pose Control of Robot Manipulators Using Conformal Geometric Algebra: A visualisation of a 6-dimensional end effector pose

An end effector pose describes the joint angles or positions of a robotic hand or gripper relative to the robot itself. This is a simple and straightforward approach to representing grasps. In this work, we use the terms "hand configurations" and "end effector poses" interchangeably.

In this work, we will utilise object meshes in our Pybullet simulation as our object representation and adopt end effector poses for our grasp representation.

2.2 Role of Tactile Sensing in Robotic Grasping

Tactile sensing provides robots with the ability to sense and respond to physical contact with target objects in their environment in a similar manner to human beings. It has been widely used to provide robots with information about target objects, allowing them to grasp and manipulate said objects with greater precision. Incorporating tactile sensors into their end-effectors enables robots to better grasp and manipulate objects of varying shapes, sizes, and textures.

A common use of tactile sensing in robotic grasping is to determine object properties [10, 24, 26]. Tactile sensing can be used to determine the properties of objects by detecting and analysing various physical cues, such as force, pressure, and vibration. By measuring these cues at different points on an object's surface, tactile sensors can provide information about the object's size, shape, texture, and hardness, among other properties. [10] discusses a tactile perception strategy that enables mobile robots with tactile sensors to measure the properties of objects that are inaccessible through visual perception using a two-finger gripper. [24] applies tactile sensing to a robot hand which localises the object and selects the best trajectory for achieving the target placement through a

decision-making process. [26] uses a neural network (learning vector quantisation (LVQ)) for object classification using pattern variations in position, orientation and size.

Tactile sensing is also used for object recognition by comparing the sensory data to a database of known objects, which helps the robot identify the target object they are grasping [11, 21, 37]. This allows the robot to execute appropriate grasping and manipulation strategies. [21] uses modified tactile sensors to introduce passive degrees of freedom (DOF) to improve the acquisition of sensor information. The tactile data collected is then used to classify an object without building a 3D model of the object. [37] collects low-resolution images of objects obtained from grasping them and applies unsupervised clustering to learn a vocabulary of tactile observations. The study demonstrates that the approach is able to classify a large set of objects and distinguish between visually similar objects with different elasticity properties by solely relying on tactile sensory data.

Tactile sensing is also widely-used for grasp optimisation and grasp stability prediction [3, 12, 16, 29, 31]. [3] uses tactile imprints as perceptions and a kernel logistic regression model to determine the probability of grasp success. [12] summarises how high-resolution tactile sensors can improve the prediction of grasp stability when used in combination with vision and depth sensing for training neural networks. Monitoring grasp stability [2, 9, 18, 22] is achieved through tactile sensing by measuring the forces applied to the target object during grasping, where tactile sensors detect changes in the grasp or slipping of the object, hence providing information that can be utilised to correct the robot's grasp and maintain stability.

2.3 Common Grasping Approaches

Grasping approaches are used to equip a robot with the ability to effectively grasp and manipulate objects in its environment. These approaches can be broadly categorised into two types: analytical approaches [15, 23], which involve developing mathematical models based on theoretical principles to compute grasping configurations, and numerical (empirical) approaches [9, 7, 16] which are learning-based approaches that adopt data-driven techniques to enable robotic grasping.

A common approach is to train regrasping policies for robotic grasping. Chebotar et al. [9] aims to learn a general regrasping behaviour using a reinforcement learning policy. During the initial phases of grasp learning in this work, a grasp stability predictor (using Support Vector Machines) was learned for the early detection of grasp failures based on spatio-temporal tactile features, where Spatio-Temporal Hierarchical Matching Pursuit (ST-HMP) was used for temporal tactile data classification tasks. Next, grasp prediction was utilised to provide feedback for reinforcement learning of low-dimensional regrasping

policies for single projects. Principal Component Analysis (PCA) was used to reduce the dimensionality of the ST-HMP descriptors to a single principal component, and Relative Entropy Policy Search (REPS) was used to optimise linear policies for individual objects. Finally, a high-dimensional policy was trained via supervised learning using the outputs of the aforementioned individual policies. This is performed using a neural network that employs the Levenberg-Marquardt (LM) optimisation algorithm, which is based on the Gauss-Netwon algorithm that is famous for converging quickly. Hyperbolic tangent sigmoid functions are used as activation layers in the network, and a linear transfer function was used as the activation function for the output layer. This approach was able to learn complex high-dimensional policies using small amounts of data by using reinforcement learning to mimic the behaviour of various simple policies. The method achieved an 80% grasp success rate after a single regrasp on an unseen object during training, and demonstrated that the network was able to generalise the learned policies to more complex objects.

Calandra et al. [7] presented an action-conditional model (a multi-modal Convolutional Neural Network) to learn regrasping policies from tactile and visual data. The model predicts the grasp outcome of a candidate grasp adjustment, then executes a grasp by iteratively selecting the most promising actions.

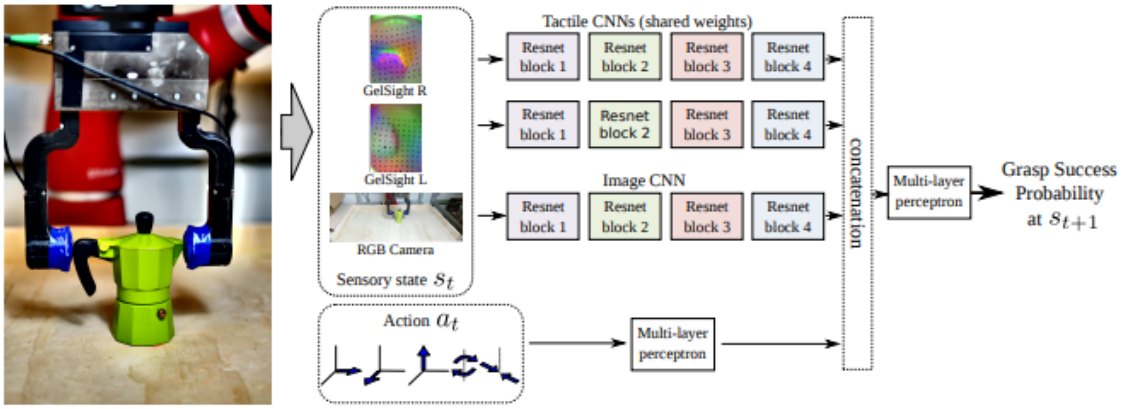


Figure 2.6: Calandra et al. [7]: More Than a Feeling: Learning to Grasp and Regrasp using Vision and Touch: A study on learning regrasping policies from multi-modal (tactile and visual) data by predicting the grasp stability outcome of a candidate grasp adjustment, then executing a grasp by iteratively selecting the most promising actions

The above figure is an overview of the study's model. It consists of two branches of tactile CNNs, one each for tactile sensory readings collected on each finger of its two-finger gripper robot, one branch of visual CNNs for extracting features from the object scene captured by a camera, and a multilayer perceptron (MLP) to preprocess possible actions. The outputs of these networks are concatenated and passed into another MLP, which outputs a single grasp success probability for a candidate grasp adjustment. This approach

does not require the calibration of the tactile sensors or any analytical modelling of the contact forces between its sensors and the target object.

An alternative grasping approach to learning regrasping policies is object exploration and finger placement optimisation. De Farias et al. [16] presents an approach to tackle to problem of a robot only having a partial camera view that looks at a single side of an object.

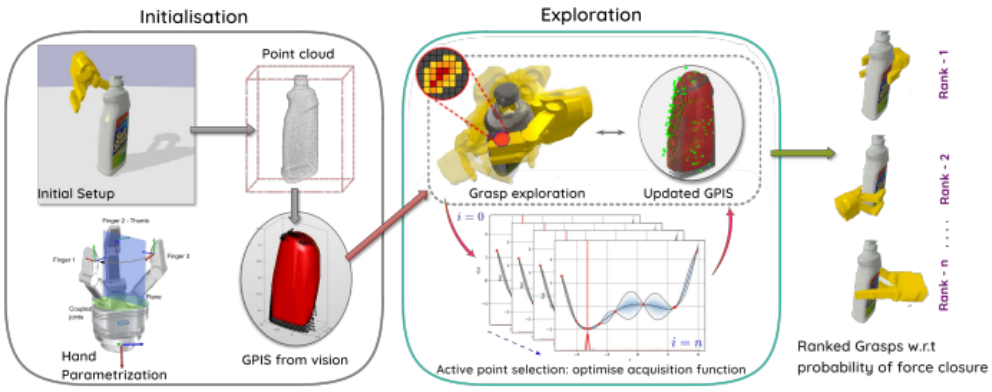


Figure 2.7: De Farias et al. [16]: Simultaneous Tactile Exploration and Grasp Refinement for Unknown Objects: A Bayesian Optimisation-based exploration pipeline for shape reconstruction and grasp search. This approach explores an unknown object to model its shape using tactile sensors, while also improving finger placement to optimise grasp stability simultaneously.

The presented approach in this study uses an optimisation framework that optimises two objectives, where successive finger placements are planned to improve grasp stability and shape representation. Optimisation is done simultaneously for grasp stability and model shaping without exhaustive exploration. Emphasis is put on improving the placement of fingers of a multi-finger hand in terms of optimising grasp stability, while also simultaneously exploring the shape of a partially unknown object. A free-floating hand was deployed in Pybullet for proof of concept, and a hand and arm setup was used for a concrete implementation.

An overview of the pipeline of this approach can be found in Figure 2.7. To summarise, the pipeline is split into three phases: 1) determining a good shape for grasping. This phase yields sufficient surface reconstructions to identify and categorise objects; 2) addressing errors and uncertainty since no object can be modelled perfectly. This is done by using GPIS to construct a probabilistic object shape representation; 3) grasp planning via Bayesian Optimisation, which handles complexity and uncertainty in various parameters to create optimal grasps.

2.4 Conclusion

To summarise, this section has explored several strategies that could potentially improve grasping:

1. Choice of object representation: Coarse approximations of the underlying true shape can simplify grasp generation and reduce computability costs.

2. Using local symmetry properties: This captures key object features, especially for objects with more symmetry such as rectangular boxes and cylinders.

3. Shape modelling: Representing objects and grasps in a common space enables transferable grasps to various objects. However, this may deteriorate with partial point cloud data, therefore the shape space must accommodate missing data while avoiding unrealistic shape reconstructions.

4. Use of tactile sensing: Tactile sensing can improve grasp planners, for example, through reinforcement learning to support open-loop systems. It provides an in-depth local regrasping procedure to improve stability.

Most studies we have investigated incorporates some combination of tactile and visual data for learning grasping policies or predicting grasp stability, but very few have been optimised for high and robust performance. Therefore, our work primarily addresses the impact of various feature representations on the performance of our supervised-learning models to reduce the dimensionality of our datasets, while attempting to achieve good robustness across all known objects in our project.

Chapter 3

Preliminaries: Simulation and Data

In this chapter, we present the practical aspects of the project. We first provide an overview of the Pybullet simulation and its functionalities specific to the project.

3.1 Pybullet Simulation Overview and Functionalities

Pybullet is a Python module designed for robotics simulation and machine learning purposes, specifically emphasising sim-to-real transfer, as indicated by Coumans and Bai [13]. In this project, Pybullet's functionalities play a crucial role, such as:

1. Allows the loading of articulated bodies from various file formats, such as Unified Robotics Description Format (URDF) files, enabling the development of complex robotic systems;

urdf > objects > mustard_bottle > mustard_bottle1.urdf > ...

```
<?xml version="1.0" ?>
<robot name="mustard_bottle">
  <link name="baseLink">
    <visual>
      <origin rpy="0 0 0" xyz="0 0 0"/>
      <geometry>
      <mesh filename="poisson/textured.obj" scale="0.6 0.8 0.8"/>
      </geometry>
       <material name="purple">
        <color rgba="1.0 1.0 0.0 1.0"/>
      </material>
    </visual>
    <collision>
      <origin rpy="0 0 0" xyz= "0 0 0"/>
      <geometry>
      <mesh filename="poisson/textured.obj" scale="0.6 0.8 0.8"/>
      </geometry>
    </collision>
  </link>
</robot>
```

Figure 3.1: Example of a URDF files that describes a bottle. The file can be loaded into Pybullet to generate an object mesh of the bottle

2. Provides forward dynamics simulation and inverse dynamics computation, enabling accurate physical modelling of robot movements and interactions with the environment;
3. Offers collision detection queries, which play a vital role in robot safety and efficiency.

A system architecture diagram for our Pybullet simulation can be found in the Appendix.

3.1.1 Role of the Pybullet Simulation

In the scope of this project, the Pybullet simulation plays a fundamental role in the acquisition of both visual and tactile data. By simulating the robot's dynamics and its interactions with the environment, Pybullet generates data that can be exploited for the purpose of training machine learning algorithms or assessing control strategies. Furthermore, the collected data may also be utilised for analytical purposes, such as the evaluation of the efficacy of distinct robotic designs or the appraisal of the performance of a specific algorithm. Figure 3.1 is a screenshot of our Pybullet simulation with core user-defined parameters on the right panel. Some important parameters include:

1. Manually manipulate of the end effector position of the robot gripper (via the sliders)
2. Data collection pipelines for the baseline and proposed approaches (via the "Collect Data" buttons)

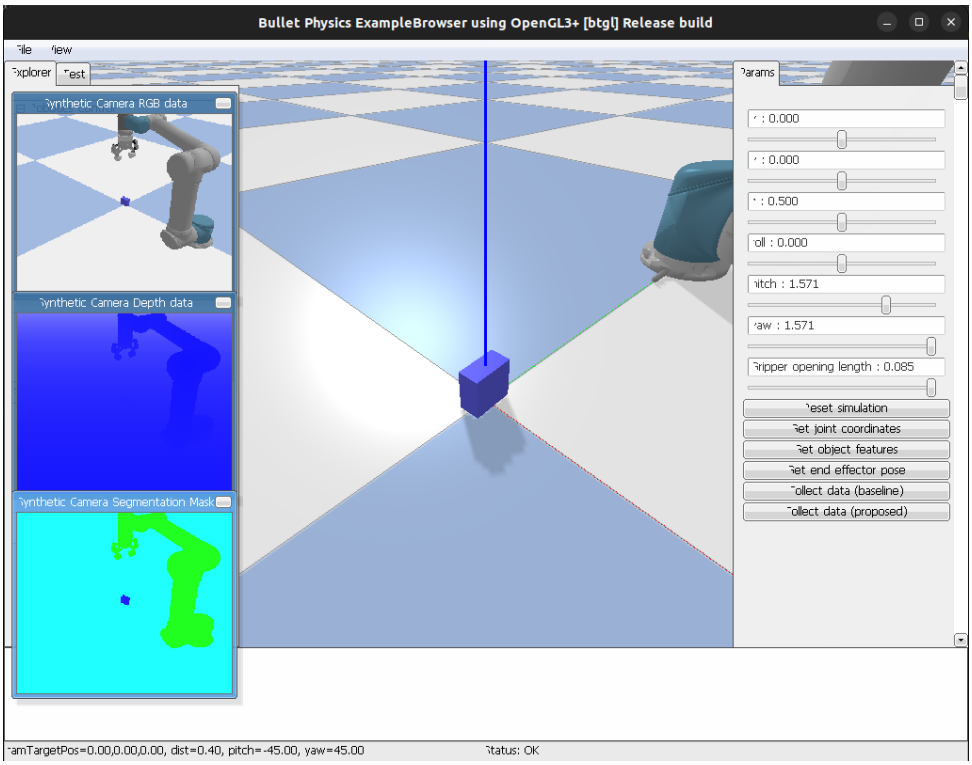


Figure 3.2: A screenshot of the Pybullet simulation for this project. The left panel displays real-time visualisations of colour (RGB) and depth data. The right panel includes parameters that can be manually controlled to manipulate the robot setup to perform specific tasks.

3.1.2 Configuring the Simulation Environment

This project allows users to configure the simulation via pre-defined parameters, including Pybullet camera positions and orientations, object variations, and robot setup parameters (e.g., Cartesian position, speed of grippers, etc.), by loading them through the `parameters.yaml` file. YAML is a widely-used format for configuration files that is both human-readable and machine-parsable. In a YAML file, data is structured in a hierarchical system of key-value pairs, which are expressed in a plain-text format.

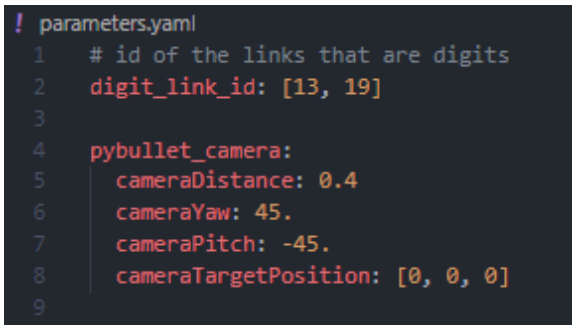


Figure 3.3: Snippet of the parameters.yaml configuration file for the Pybullet simulation

3.1.3 Robot Control and Data Collection

Our Pybullet simulation consists of two data collection pipelines for the baseline and proposed approaches respectively (see Section 4.2.2). These pipelines are activated through the `Collect Data` buttons on the right panel in the simulation. The data collection pipelines generate randomised end effector poses from manually-selected seed poses around the vicinity of the target object. Every time a new pose is generated, the robot's position and orientation are reset, and we move the robot to the generated pose via inverse kinematics provided by Pybullet.

3.2 Description of Data

This work employs the terms "tactile" and "visual" data with frequent recurrence. A comprehensive elucidation of these terminologies shall be presented in the forthcoming sections. To acquire the majority of the data, we utilise "cameras", which are devices that capture sensory data by means of imaging objects being grasped as well as their surrounding environment. In the domain of robotic grasping, cameras are typically installed on a robot gripper and are capable of capturing diverse image types contingent on the given task. In Section 3.2.4, we shall delve further into how the real-time visualisation of depth and colour image representations of tactile data is achieved within our simulation.

3.2.1 Tactile data

In robotic grasping, tactile data refers to rich information collected from contact with a robotic hand or gripper during the manipulation of objects. It reflects the amount of force and pressure applied by the gripper on an object in the grasping process. Tactile data can be extracted from tactile sensors (cameras) mounted into robotic end effectors or grippers, which can be used to provide feedback to the control system about the grasp quality, such as whether the object is slipping or not, and to adjust the gripper's position and force to improve the grasp. In our simulation, we mount DIGIT tactile sensors [29] to each finger of a 2-finger gripper, thus generating a pair of tactile (depth and colour) readings per grasp.

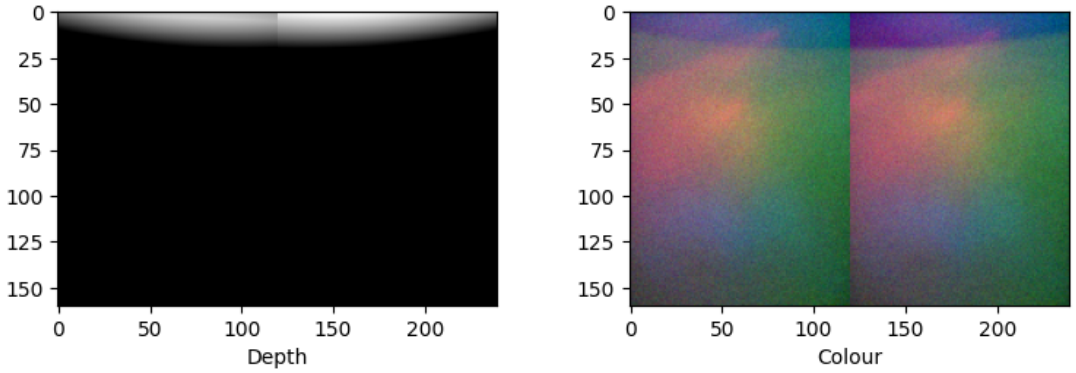


Figure 3.4: Example pair of depth and colour images extracted from the DIGIT cameras

3.2.2 Visual data

Visual sensing is an intuitive way for a robot to perceive its environment and is widely used in robotic grasping [22]. In this work, visual data refers to information collected by cameras in our simulation. Our visual data will consist of any of the following data sources:

1. 6-Dimensional end effector poses relative to the robotic arm/gripper. We refer readers to Section 4.2.1 for a detailed description of this data.

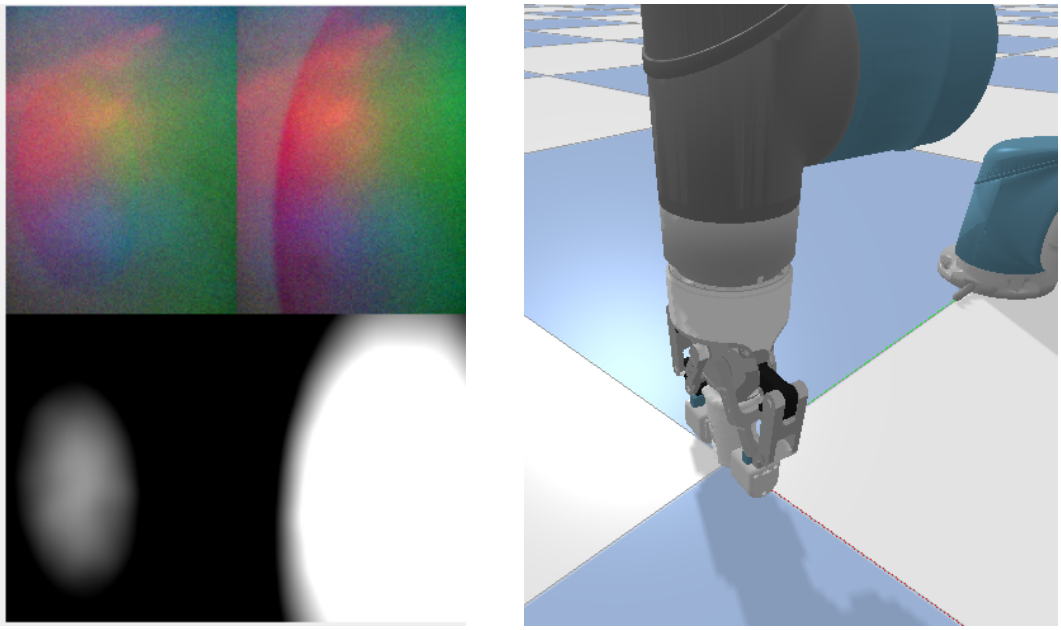
2. Geometric features of target objects, which we utilise in Chapter 5.

3.2.3 DIGIT Camera for Live Tactile Data Visualisation

The PyBullet simulation visualisation tool is based on OpenGL and provides a 3D rendering of the simulated environment. Users can manipulate the camera view, zoom in and out, and move around the simulation environment using the mouse and keyboard. The GUI also provides tools for controlling the simulation, such as pausing and resetting, and for editing the simulated objects.

In addition to the basic visualisation capabilities, the PyBullet GUI also supports advanced features such as rendering depth and segmentation maps, displaying sensor data from cameras and other sensors, and recording and playing back simulations. It also provides a Python scripting interface that allows users to automate tasks and customise the visualisation and simulation behaviour.

DIGIT cameras in PyBullet [40] are often used in conjunction with the "Camera Sensor" feature, which allows users to specify the position, orientation, and other parameters of a virtual camera within a simulated environment. Once configured, the camera can be used to capture RGB images, depth maps, or other types of sensor data from the environment. The captured images can then be used as input to machine learning models, or as visual feedback for controlling robotic agents within the simulation.



(a) Real-time colour and depth readings from DIGIT camera

(b) Pybullet robot gripper and object state

Figure 3.5: Demonstration of real-time visualisation of tactile sensory readings via DIGIT tactile sensors with Pybullet

To generate live visualisations of colour and depth sensory data, we configure the DIGIT camera sensor within our Pybullet simulation by specifying its position and orientation relative to the sensors. Next, we capture the sensor data from the simulation and feed it into the DIGIT camera sensor, and finally use Pybullet's built-in visualisation tool which provides real-time visualisation of Pybullet physics simulations to display live visualisations of the tactile sensor data.

Chapter 4

Learning Feature Representations

This chapter introduces a baseline approach to grasp stability prediction using a binary logistic regression model. In a previous study conducted by Alkhatib et al. [1], a regularised logistic regression model was used to determine the robustness and quality of grasps and predict their stability. The model's inputs were derived from a set of features measured at each joint of a robotic hand, which were subsequently utilised for training the logistic regression model. In our approach, we use tactile and visual data instead of joint data to classify binary grasp stability labels.

In addition, we address a pressing issue of the project - high dimensionality data collected from our simulations. In the previous chapter, we highlighted that a significant portion of the data collected from our simulation is high-dimensional. High-dimensional data poses a significant challenge in terms of computational processing and feature identification. The risk of overfitting also increases as the dimensionality of the data grows, leading to sub-optimal performance of the grasping policy. To mitigate these challenges and effectively train the grasping policy, it is imperative to reduce the data's dimensionality and represent it more concisely.

Dimensionality reduction and feature representation techniques can be employed to reduce the data's dimensionality and represent it in a more meaningful way. By doing so, we can identify the most relevant features for the grasping task and reduce the risk of overfitting. Techniques such as Principal Component Analysis (PCA) can be used to determine the most important features in the data. Alternatively, Convolutional Neural Networks (CNNs) can automatically learn features from images or depth maps. A previous study [7] has demonstrated the effectiveness of CNNs in extracting features from high-dimensional data for grasping tasks.

To determine an appropriate feature representation, we adopt a binary logistic regression model (commonly known as binary logit) for the classification of grasps and evaluation

of the accuracy of the model on various feature representations. Logistic regression is a widely used binary classification algorithm that can be effectively applied to datasets with non-linear relationships, as may exist between the multi-modal features and grasp stability label in our project. It serves as a strong baseline for our investigation, as it provides interpretable results that facilitate the understanding of the effectiveness of the chosen feature representations.

4.1 Logit Model for Grasp Classification

The fundamental assumption of this approach is that grasps can be classified as either successful or unsuccessful outcomes. In this study, we classify successful hand configurations as those that result in a stable grasp of the object in question. However, there are more sophisticated methods for categorising grasps, which we refer the reader to [2, 39].

Assume that we have selected some representation of tactile and/or visual features for each grasp candidate, denoted x_i. Our logit model attempts to classify the grasp with a binary label y_i. To achieve this, the weights w of the model are fitted through a maximum likelihood estimation where the distribution of the features x_i sits. Assuming the grasp outcomes as a Bernoulli random variable, we can construct and minimise the following cross-entropy loss function and treat it as an optimisation problem:

$$\ln(L(w)) = \sum_{i=1}^{n} y_i \cdot ln(\rho_y(y_i = 1|x_i)) + (1 - y_i) \cdot \ln(\rho_y(y_i = 0|x_i)) \tag{4.1}$$

where $\rho_y(y_i = 1|x_i)$ and $\rho_y(y_0 = 1|x_i)$ represent the probability of a grasp x_i being successful ($y_i = 1$) or unsuccessful ($y_i = 0$) respectively.

Therefore using our learned logit model (with an optimised weights vector), we can determine whether the outcome (successful or unsuccessful) of an unseen grasp is ascertainable based on its feature representation.

4.2 Dataset Generation

To generate a dataset for training our proposed logit model, we collect tactile and visual data by randomly generating grasps on a smooth rectangular box of dimensions (depth: 0.025, width: 0.05, height: 0.05) in our Pybullet simulation. These dimensions are re-scaled to be compatible with the dimensions of the robot setup, which are defined in its URDF (Unified Robot Description Format) file.

4.2.1 Randomised Grasp Sampling

To generate random grasps on the box, we first manipulate the robot's end effector manually into diverse positions within the box's vicinity and collect three distinct end effector poses $S_i : i = 1, 2, 3$ to ensure some variability in grasping orientation, position and efficacy. The selected end effector poses are subsequently referred to as seed poses, denoted as 6-tuples S_i:

$$S_i = (x, y, z, r_x, r_y, r_z) \tag{4.2}$$

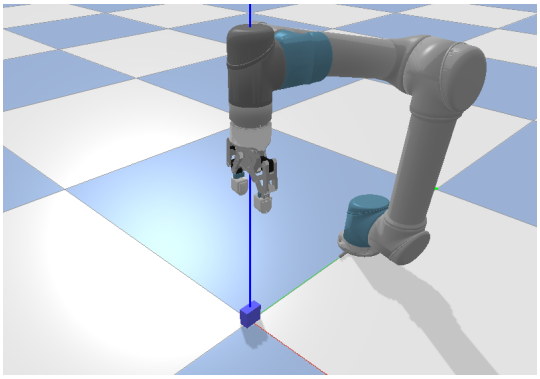
A fixed number n of random poses are generated from each seed pose. This is accomplished by adding a small amount of Gaussian noise K_n, with a mean of zero and a unit variance, to each dimension of S_i. Taking into consideration the range of variation in Cartesian coordinates, object and robot dimensions of the Pybullet simulation, the applied noise is scaled down by a factor of 0.01. The resulting grasp poses are denoted as X_v as follows:

$$X_{v,i} = \Big\{ S_i + 0.01K_i : K_i \sim \mathcal{N}(0, 1),\ i = 1, 2, 3, ..., n \Big\} \tag{4.3}$$

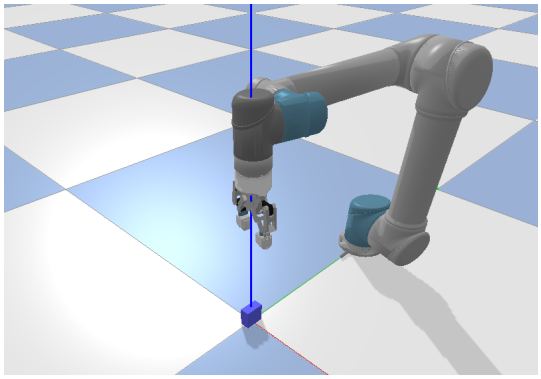
The variance of the Gaussian noise can be adjusted for a more/less dispersed distribution of the random grasp poses. It is essential to balance the trade-off between generating diverse grasp poses (and their corresponding data) and ensuring that the poses are likely to succeed. Notably, the impact of the variance of the Gaussian noise will be contingent on the characteristics of the object being grasped, which may necessitate modifying the variance according to the object's properties. This will be discussed in Section 5 where we introduce geometric features of objects into our classifier approach.

4.2.2 Data Collection Pipeline

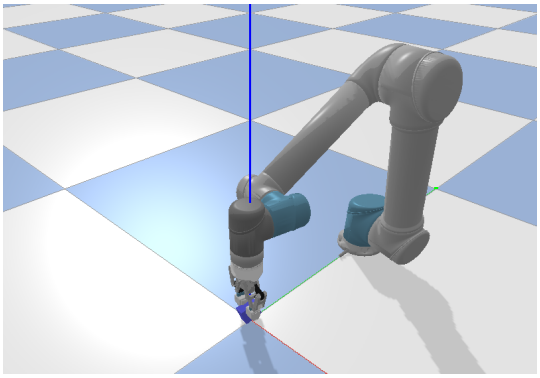
Prior to executing each randomly generated pose $X_{v,i}$ as a pick-and-place task, we first reset the position and orientation of our robot to an initial, predefined configuration (Figure 4.1a). Next, we utilise Pybullet's built-in inverse kinematics method to manipulate the robot's end effector to $X_{v,i}$. To avoid collision with the object, we include a vertical padding distance p_z above the object (Figure 4.1b). Then, the robot lowers the end effector by removing the padding p_z to reach the target object and closes its gripper (Figure 4.1c).



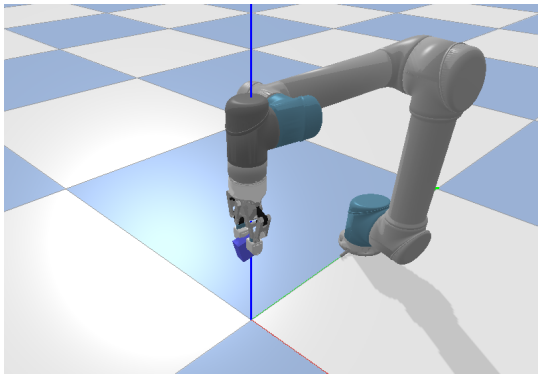
(a) Reset robot to initial configuration



(b) Move end effector to specific pose (with vertical padding)



(c) Remove vertical padding and close gripper



(d) Lift object

Figure 4.1: Tactile and visual data collection procedure on a single generated hand pose

Before lifting the object, the robot measures the number of contact points between the two fingers and the object to ensure the grasp is stable. We define a grasp to be stable if there are a non-zero number of contact points, in which the robot records depth and colour tactile information using the DIGIT [29] tactile sensors on each finger. We concatenate each pair of depth and colour tactile readings into a (160×240) grey-scale image and a $(160 \times 240 \times 3)$ RGB image respectively.

Upon acquiring tactile readings, the robot executes a sanity check on the recorded tactile data from DIGIT cameras. The DIGIT cameras use a combination of depth sensing and colour imaging technologies to produce tactile readings. These cameras have a structured light system that projects a pattern of infrared light onto the object being touched. The camera then captures the reflected light and uses it to calculate the depth of the object at each pixel location. To produce colour tactile readings, the camera also includes a traditional RGB colour camera. This camera captures the colour information of the object at each pixel location. The depth and colour information is then combined to produce a textured 3D representation of the object's surface, which can be used to extract tactile features such as roughness, curvature, and shape. The following figure describes a set of depth-colour tactile readings collected from the DIGIT cameras via our simulation:

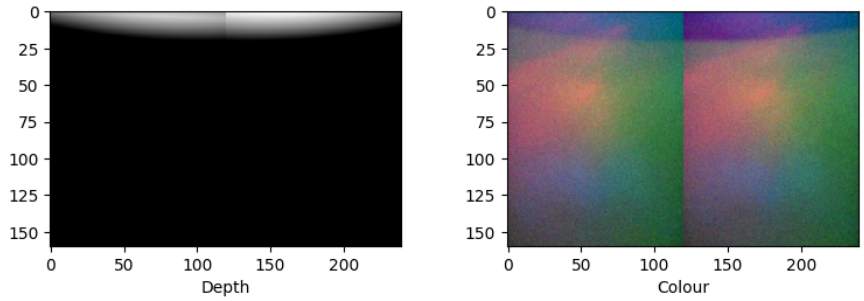
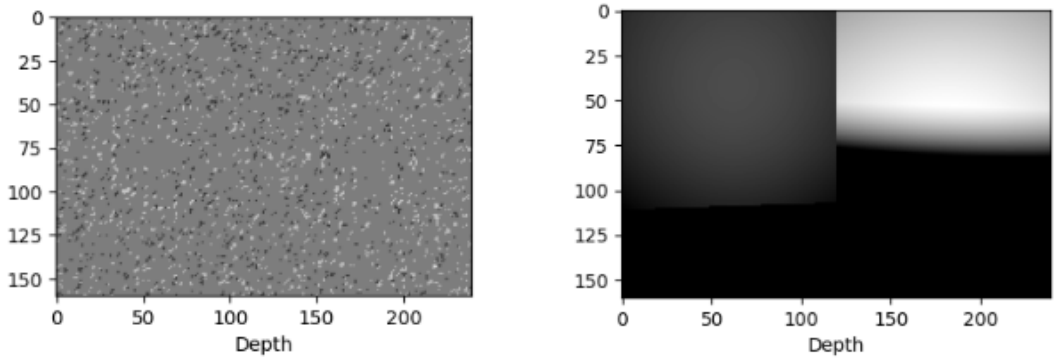


Figure 4.2: Example depth and colour tactile readings collected from a DIGIT camera

After running the sanity check, we analyse the depth data exclusively, as the colour data is a mere RGB representation of the depth data, and thus does not require any additional scrutiny. This process involves assessing whether the average pixel value surpasses a predetermined threshold, thereby enabling the elimination of spurious signals (Figure 4.3a) and preserving the integrity of the precise depth data (Figure 4.3b). By implementing this approach, the quality of the data is significantly improved, thus ensuring the accurate lifting of the object.



(a) Invalid depth data

(b) Valid depth data

Figure 4.3: Concatenated depth data for each finger

Once the robot verifies that the object is stably held (i.e. the tactile data is valid), it lifts the object vertically upwards by an offset of p_z and holds the object for 750 steps (Figure 4.1d). Finally, the robot records a binary outcome of the grasp by determining if the change in the object's z-position Δz is greater or equal to the vertical padding p_z:

$$outcome = \begin{cases} 1 & \Delta z \geq p_z \\ 0 & \Delta z < p_z \end{cases} \tag{4.4}$$

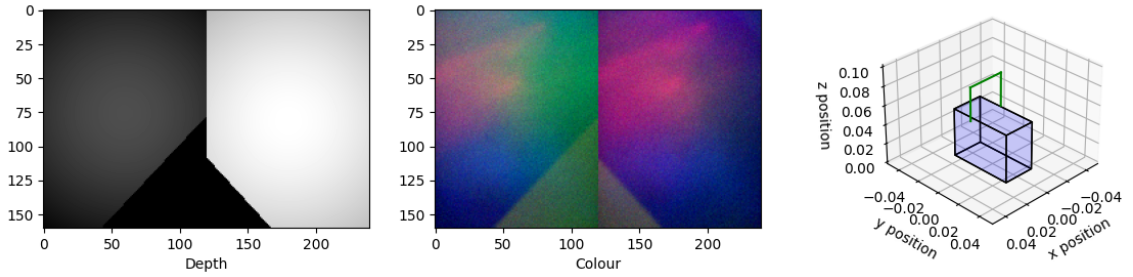
After executing each random pose, we append the collected tactile data, end effector pose and grasp outcome to individual arrays. Once the required data for our n random end effector poses has been collected, these arrays are saved to separate files which can be loaded for further analysis. Specifically, we create one .npy file each for the depth, colour, grasp outcomes, and random pose data.

4.3 Dataset Validation and Visualisation

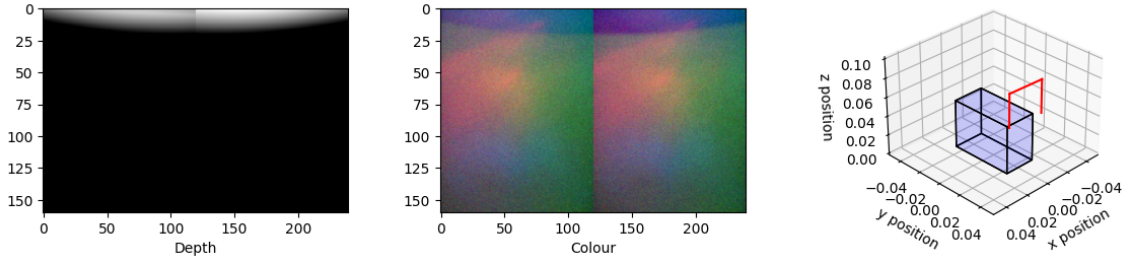
Before using the generated tactile and visual data for random hand configurations for analysis or modelling, it is important to validate and visualise it to ensure its quality and reliability. In this context, this article will explore the process of visualising and validating a dataset of tactile sensor readings and end-effector poses. We will discuss various techniques and tools that can be used to explore the data, identify patterns and anomalies, and ensure that it meets the requirements of the analysis or model being built. In this section, we explore our data to understand any underlying patterns or characteristics and prevent the generation of abnormal grasp data (as in Figure 4.3).

4.3.1 Visualising Tactile Data

The tactile readings collected from our simulation are represented in the depth and colour images, as depicted in the figures below. These figures display the data for two grasp attempts - one successful (Figure 4.4a) and one unsuccessful (Figure 4.4b) - which were based on end effector poses randomly generated by the system. Furthermore, we present a 3D visualisation of the hand pose and the object, with the target object shown as a blue box and the gripper represented as a staple-like object consisting of two fingers.



(a) Successful grasp data



(b) Unsuccessful grasp data

Figure 4.4: Left to right: Depth readings, Colour (RGB) readings, Skeleton 2-finger model to represent a hand pose

4.3.2 Visualising Hand Poses

We create a visualisation of each hand pose X_n in a 3-dimensional space, using its Cartesian coordinates and the "pitch-roll-yaw" rotational matrix representation R of its Euler angles (α, β, γ) [41]:

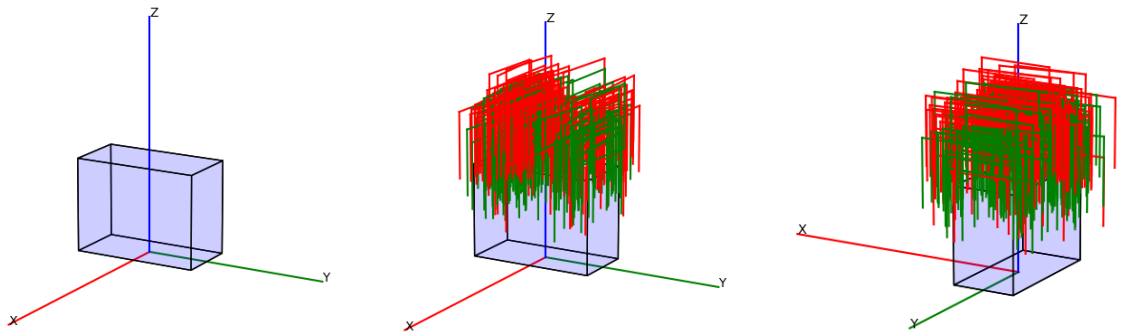
$$R = r_x \cdot (r_y \cdot r_z) \tag{4.5}$$

where r_x, r_y and r_z are the rotational matrices about the x, y and z-axis respectively:

$$r_x = \begin{pmatrix} 1 & 0 & 0 \\ 0 & cos(\alpha) & sin(\alpha) \\ 0 & -sin(\alpha) & cos(\alpha) \end{pmatrix} \quad r_y = \begin{pmatrix} cos(\beta) & 0 & -sin(\beta) \\ 0 & 1 & 0 \\ sin(\beta) & 0 & cos(\beta) \end{pmatrix} \quad r_z = \begin{pmatrix} cos(\gamma) & sin(\gamma) & 0 \\ -sin(\gamma) & cos(\gamma) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Figure 4.5: Rotational matrices for each axis in the rotational matrix representation of Euler angles

Using this information, we annotate all randomly-generated hand poses from our dataset in 3 dimensions using `matplotlib`. Each 6D hand pose is represented using a skeleton 2-finger model in a similar manner to [2], where green and red poses denote successful and unsuccessful grasps respectively:



(a) Box model. Grasp poses are not shown.

(b) Orientation 1

(c) Orientation 2

Figure 4.6: Visualisation of 200 random end effector poses

4.4 Feature Engineering & Data Pre-processing

We aim to assess the efficacy of utilising our multi-modal dataset for our grasping task by experimenting with various combinations of the tactile and visual (random end effector poses) data.

4.4.1 Multi-modal Data Combinations

Based on the tactile and visual data collected in section 4.2, we created different combinations of the dataset to determine the best representation for our analysis. The table below presents a summary of the various data combinations we tested and their corresponding creation methods.

Data Combination	Dimensions	Creation
Tactile only	$N \times 160 \times 240 \times 4$	Concatenating the depth ($N \times 160 \times 240$) and colour ($N \times 160 \times 240 \times 3$) data
Visual only	$N \times 6$	Random 6D end effector poses
Tactile + Visual	$N \times (160 \times 240 \times 4 + 6)$	Concatenating the flattened tactile-only and visual-only datasets

Table 4.1: Dataset combinations

The first dataset consists of tactile sensor readings collected from a robotic gripper during grasping tasks. The dataset has a shape of $N \times 160 \times 240 \times 4$, with N distinct concatenated colour and depth tactile sensory readings per finger collected from the simulation data collection pipeline (4.2.2). The following flowchart illustrates how the tactile dataset was created:

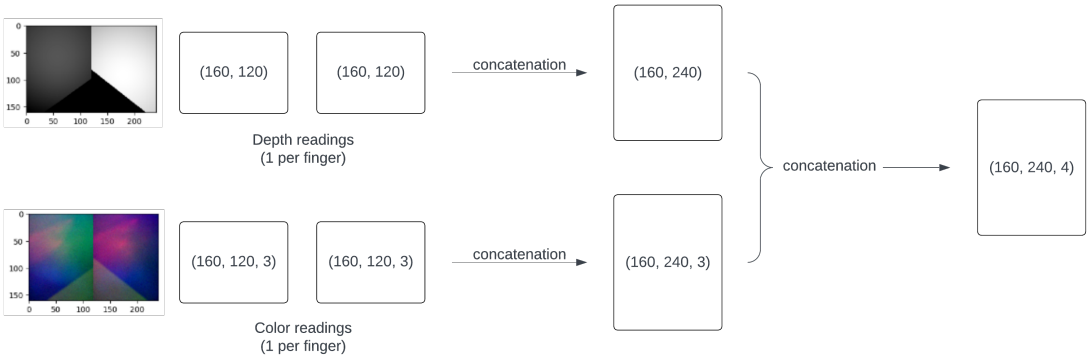


Figure 4.7: Creating the tactile dataset

The second dataset consists of end effector poses relative to the robot arm during grasping tasks. The dataset has a shape of $N \times 6$, where 6 represents the 6 degrees of freedom (DOF) of the simulation robot.

The third dataset is a combination of tactile and visual data and consists of both tactile sensor readings and end effector poses collected during grasping tasks, concatenated together as NumPy arrays along the second dimension, creating a dataset of the shape $N \times (160 \times 240 \times 4 + 6)$.

Concatenation approaches involving the combination of sensory data for robotic grasping analysis have been explored in several related works. Calandra et al. (2018) [7] used a combination of tactile and visual data to train a deep neural network for predicting grasp outcomes. The authors concatenated the tactile sensor readings and end effector poses into a single input vector for the neural network. In this project, we adopt a similar approach by concatenating our tactile and visual data.

4.4.2 Dimensionality Reduction

In this project, we represent tactile sensory readings as images, specifically, grayscale for depth data and RGB for colour data. As the dimensionality of this data has a significant impact on computational costs, we have employed dimensionality reduction techniques to alleviate this issue, including principal component analysis (PCA) and convolutional neural networks (CNN) for feature extraction.

Principal Component Analysis (PCA)

PCA is a widely used linear technique that projects the high-dimensional input data onto a lower-dimensional space, capturing the maximum amount of variance in the data with fewer dimensions. By reducing the dimensionality of the data, PCA can facilitate more efficient computation and enable the extraction of meaningful patterns and features from the data.

PCA is particularly useful for concatenated tactile and visual data, as it can extract relevant features from both modalities and combine them to form a lower-dimensional representation of the data. For example, PCA can be used to extract principal components that represent the most significant tactile and visual features, which can then be combined to form a lower-dimensional representation of the data. The resulting representation can be used as input to machine learning algorithms for classification or regression tasks.

The effectiveness of PCA for dimensionality reduction depends on several factors, such as the number of input dimensions, the amount of variance and the distribution in the data. Additionally, the choice of the number of principal components to retain can also impact the performance of the method. Hence, a careful evaluation of the effectiveness of PCA for the given task is necessary, including the selection of an appropriate number of principal components to retain, and any trade-offs between computation time and performance.

Convolution Neural Network (CNN) Feature Extraction

In the context of multi-modal sensor data, feature extraction using convolutional neural networks (CNNs) has gained popularity due to their ability to capture spatial and temporal patterns in the data. The CNN architecture, as described with interleaved convolution and pooling layers, is well-suited for detecting local features in the input data. The convolution layers extract low-level features, such as edges or textures, while the pooling layers help reduce the dimensionality of the feature maps and provide translational invariance to the learned features. The combination of these layers can create a hierarchy of feature representations, with higher-level features capturing more complex and abstract patterns in the data.

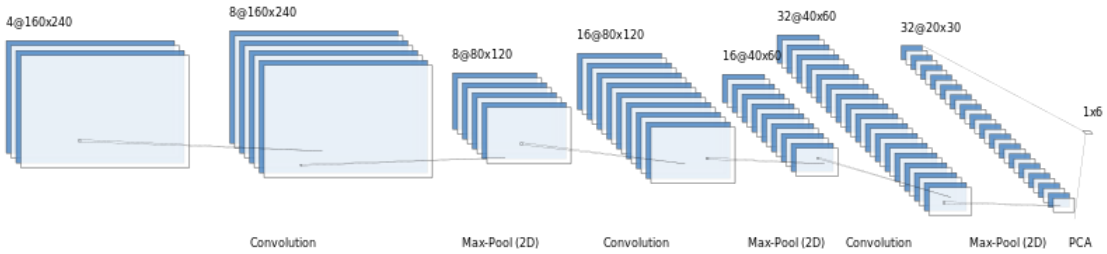


Figure 4.8: Architecture for feature extraction CNN

Figure 4.8 visualises the specific CNN architecture used in the baseline, consisting of three convolutional layers interleaved with MaxPool2D layers with a 2×2 kernel, which is effective in capturing features from concatenated tactile and visual sensory data.

4.4.3 Data Pre-processing

Normalisation of data prior to model training is a well-established practice in machine learning, known to improve the accuracy and efficiency of the learning process. In the context of tactile sensor readings and end effector poses, normalisation is particularly relevant due to the large differences in the scale of the data. For instance, RGB pixel values for the colour dataset vary between 0 to 255, while Euler angles in the hand pose dataset vary between $-\pi$ and π.

Failure to normalise the data can result in numerical instability, as differences in scale can cause the weights of the model to become disproportionately small or large, thereby hampering effective learning. Moreover, normalisation can improve model generalisation by reducing the impact of outliers and mitigating the effects of varying ranges of feature values in the data. Finally, normalisation facilitates comparison of the relative importance of different features, thereby enabling the identification of the most significant ones and elimination of redundant or less important ones.

4.5 Model Training

To train the binary logistic regression classifier on the tactile sensory data and end effector poses, an 80-20 split was used with the `train_test_split()` function provided by the `sklearn` library. The training set consisted of 80% of the data, while the remaining 20% was utilised for validation. The model was trained using the `sklearn` logistic regression implementation, which provided flexibility in setting the hyper-parameters such as the solver type and regularisation method. However, regularisation was not applied in this training process considering the high dimensionality of the dataset.

After preparing the datasets, the logistic regression (logit) classifier was trained using the three dataset combinations that were previously discussed in Section 4.4.1. To further explore the dataset characteristics, a combination of dimensionality reduction techniques was applied to these datasets, as discussed in Section 4.4.2. Specifically, the techniques used were ConvNet only and ConvNet with PCA (top k principal components). The logit model was then trained on these variations of the datasets with the aim of selecting the model and dataset combination that achieves the best accuracy. This approach allowed us to thoroughly explore the impact of different dataset combinations and dimensionality reduction techniques on the performance of the logit classifier.

4.5.1 Summary of Training Results

	Dataset			Property of sample	
Pre-processing	Visual only	Tactile only	Both	Shape	Dimensionality
Raw	70.83%	83.33%	83.33%	160x240x4	153600
CNN	N/A	83.33%	87.50%	64x10x15	9600
CNN + PCA ($k = 5$)	N/A	77.08%	77.08%	5	5

Table 4.2: Accuracy of LR model on dataset variations with N=200 samples

As previously discussed, the logistic regression model was trained using three distinct dataset segmentations, with the application of three pre-processing methods. It is noteworthy that the employment of CNN and PCA techniques was omitted for the 6-dimensional visual-only data, as its dimensionality suffices for training purposes.

4.5.2 Analysis of Training Results and Balancing the Trade-off Between Accuracy and Dimensionality

The outcomes as presented in Table 4.2 reveal that the incorporation of PCA techniques did not yield a considerable impact on the accuracy of the logit model. This observation can potentially be attributed to two underlying factors: firstly, the application of PCA to the dataset may result in the elimination of pertinent features or the introduction of noise through the utilisation of said techniques; secondly, it is plausible that non-linear relationships exist among the variables. However, it is worth noting that PCA assumes linearity in the relationships among variables, thereby implying that its accuracy may be influenced by the inability to capture relevant patterns, thereby resulting in a decline in accuracy.

This also explains why applying CNN alone preserves or even increases the accuracy of our model, since it can learn complex, non-linear relationships between the tactile and visual features and the grasp outcome. This is particularly useful when dealing with high-dimensional data, where PCA may not capture the underlying patterns in the data. Moreover, the ability of CNNs to learn hierarchical feature representations can make them more interpretable, as each layer captures increasingly complex and abstract features, leading to a better understanding of the underlying mechanisms of the problem.

Nevertheless, the application of dimensionality-reducing techniques resulted in a significant reduction of dataset dimensionality by over 90%. This demonstrates a well-balanced trade-off between the accuracy and dimensionality of our dataset.

4.6 Conclusion

Based on the results presented in Table 4.2, we have determined that using both tactile and visual data together produces the highest accuracy for our LR models. While raw data performed better than other pre-processing techniques, we recognise that the dimensionality of the data must also be considered. Therefore, we have found that extracting features from our dataset using a ConvNet and selecting only the top $k = 5$ components provides comparable accuracy with significantly lower dimensionality.

Chapter 5

Multilayer Perceptron for Grasp Stability Prediction

Achieving successful grasping of objects in a robotic environment is challenging due to the high-dimensional, noisy, and uncertain nature of the task, which requires accurate perception, planning, and control. To address this issue, we propose a multilayer perceptron (MLP) neural network that predicts the grasp stability label of an end effector pose. MLPs are feed-forward neural networks consisting of at least three fully-connected layers of perceptrons, including an input layer for receiving input data, an output layer for producing a prediction, and at least one hidden layer that enables the MLP to learn complex representations of the data. Our proposed approach is specifically designed to identify grasp configurations that yield high stability and success rates for specific objects, a critical task in robotic grasping.

MLPs are well-suited to learn non-linear relationships between input and output data, making them suitable for identifying patterned configurations from complex datasets and generalising simple grasping strategies for different object types. For example, grasping an object around its centre of gravity can minimise rotational forces that could cause it to tilt and increase the likelihood of a successful grasp. In this context, our MLP-based approach extracts important features from tactile and visual data using an identical Convolutional Neural Network architecture, from which we select the top five principal components to represent the extracted features.

To evaluate the performance of our model, we conducted several experiments on different primitive object types, including rectangular boxes, cylinders, and bottle-shaped objects (which are more rigid than typical cylinders), to assess the robustness of the model towards various objects. Additionally, we tested whether the model is able to generalise simple grasping strategies per object type. We compared the performance of our model with state-of-the-art methods for predicting grasp stability, including baseline methods on

hand-engineered features and other neural network-based models [32, 34].

In the following sections, we provide a detailed description of our approach, experimental setup, and results, and discuss the implications of our work in the field of robotic grasping.

5.1 Related Work

There have been several academic studies addressing the problem of grasp stability classification using neural networks. In this section, we will briefly review some of these works. Kumra and Kana (2017) [28] presented a novel approach to grasp detection that predicts an optimal grasping pose using RGB images of a scene. Their method considers the robotic grasp detection problem as finding a successful grasp configuration for a given object image. To address this, they proposed a single-step prediction technique, which is faster and less computationally expensive compared to previous methods that rely on running a simple classifier multiple times on small image patches.

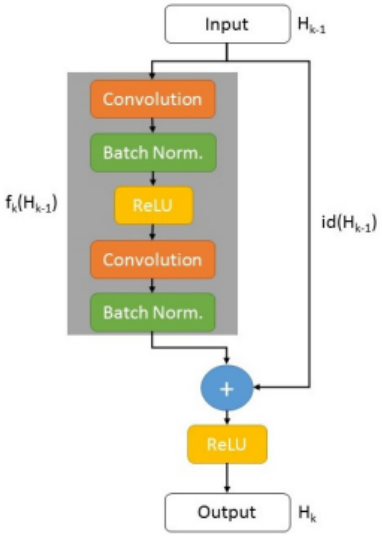


Figure 5.1: Kumra and Kanan [28]: Robotic Grasp Detection using Deep Convolutional Neural Networks: Example of a residual block in ResNet

To solve the grasp detection problem, the authors used ResNet-50, a 50-layer deep residual model (see 5.1) that incorporates residual layers to reformulate the mapping function between layers and overcome the challenge of learning an identity mapping. The authors introduced two different architectures for robotic grasp prediction: a uni-modal grasp predictor that uses only single-modality information such as RGB, and a multi-modal grasp predictor that uses multi-modal information such as RGB and depth.

Mahler et al. [32] have proposed a method for evaluating grasp quality using a CNN trained on RGB-D images and force measurements. RGB-D refers to a type of sensor that captures both colour (RGB) and depth (D) information about a scene. RGB-D sensors typically consist of a regular colour camera to capture RGB images and an additional depth sensor, such as a time-of-flight or structured light sensor, which provides a depth image. The combination of RGB and depth information can be particularly useful for robotics applications, as it allows the robot to perceive and reason about the 3D structure of the environment, and to distinguish between objects that are close together but at different depths. In the paper, the authors propose a method for evaluating grasp quality using a CNN trained on RGB-D images and force/torque measurements. The network takes as input an RGB-D image of the scene, along with the 6-dimensional force/torque measurements recorded during a grasp attempt. It then predicts a scalar value for each grasp, indicating its expected success rate. The CNN was trained on a dataset of over 10000 grasps on a variety of objects. However, a limitation of this approach is that it requires accurate force sensing during the grasp attempt, which may not be always feasible in practice. Additionally, the scalar success rate predicted by the network may not be as interpretable as other grasp stability metrics, such as the wrench space analysis or the friction cone model. Nonetheless, this paper represents an important step towards using neural networks to evaluate grasp quality in robotics applications.

5.2 Methodology

In this section, we describe the methodology used to train and evaluate our proposed approach for predicting grasp stability using a multilayer perceptron (MLP) neural network. We begin by presenting the dataset we used, which consists of tactile and visual data collected from a variety of objects, along with ground-truth labels indicating whether each grasp was successful or not. We then describe the architecture of our MLP model, including the number and size of the layers, the activation functions used, and any regularisation techniques employed. Next, we explain the training procedure used to optimise the model's weights, including the optimisation algorithm used, the learning rate schedule, the batch size, and the number of epochs. We then describe the evaluation metrics used to assess the performance of our model, along with the baseline methods against which we compared it. Finally, we present the results of our experiments and discuss their implications for the field of robotic grasping.

5.2.1 Dataset Collection

Utilising our Pybullet data collection pipeline from Section 4.2.2, we collect multi-modal data for every object variation. Then, for each variation, we manually select $i = 4$ seed poses and record the data of 20 successful and 20 unsuccessful grasps with Gaussian noise applied to the seed pose. This creates a dataset of 9x160=1440 grasps in total (720 stable and 720 unstable grasps) consisting of tactile sensor readings and hand poses.

5.2.2 Primitive Objects and Variations

To enhance the robustness of our model, we have trained it on three primitive object categories that collectively represent a sizeable proportion of daily-life objects: rectangular boxes, cylinders and bottles. For each object category, we generate three instances with slightly different dimensions. The 3D meshes of these object variations and their dimensions are listed in Table 5.1.

	 Box 1	 Box 2	 Box 3	 Cylinder 1	 Cylinder 2	 Cylinder 3	 Bottle 1	 Bottle 2	 Bottle 3
Depth	0.025	0.03	0.05	0.04	0.045	0.05	0.06	0.04	0.04
Width	0.05	0.025	0.025	0.04	0.045	0.05	0.04	0.06	0.06
Height	0.05	0.045	0.04	0.05	0.035	0.045	0.04	0.06	0.04
Radius	N/A	N/A	N/A	0.02	0.0225	0.025	N/A	N/A	N/A

Table 5.1: 3D meshes of all experimental object variations

5.2.3 Introducing Geometric Features of Objects

In this section, in addition to the end effector poses in our visual dataset, we investigate the possible impact and improvement of introducing geometric features of objects on our MLP model. Each primitive object category in our dataset possesses unique characteristics. For instance, the curvature of an object's surface can differentiate cylinders and bottles from boxes with ease. As objects within a primitive class often share similar geometric features (such as the curvature of cylinders), our model could potentially generalise successful and unsuccessful grasp poses within each object category by establishing a connection between those poses and the object's properties.

To this end, we extracted and analysed several geometric features of each of the 9 object variations in order to determine which of them can effectively differentiate between the object categories. We conclude that principal curvatures best represent the geometric features of said objects.

Principal Curvature

The principal curvatures of an object mesh are pair of maximum and minimum values of curvatures at each vertex of the mesh, expressed by the eigenvalues of the shape operator at that vertex [42]. We investigate whether the top k principal curvatures are able to categorize the object variations correctly.

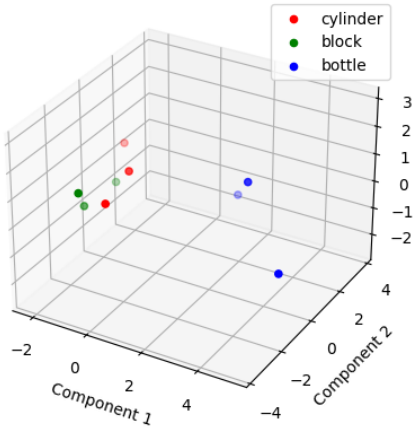


Figure 5.2: Clustering using PCA-extracted components based on principal curvatures of objects

Figure 5.2 visualises the top $k = 3$ principal curvatures for all the object variations in our dataset. We found that the principal curvatures were able to capture a high amount of variance of the curvatures for all objects (97.34%) and successfully cluster the object variations for each category.

Other Features

In addition to principal curvatures, we attempted to extract convexity and fundamental dimensions of objects, aimed at capturing their size and shape characteristics. Despite the analytical efforts, the results obtained from these features were not deemed promising and therefore were not incorporated into the model. Consequently, it can be inferred that the principal curvatures hold significant potential as a valid representation of geometric features for the object dataset under consideration.

5.2.4 Dataset Pre-processing

Based on the analysis results of our baseline approach in Section 4.7, we concluded that under limited dimensionality circumstances, the Logistic Regression model fits the dataset with the highest accuracy using a convolutional neural network as a dimensionality reduction technique. Therefore, before training our MLP, we extract important features from our dataset using the same CNN architecture as 4.4.2. This results in a condensed dataset of shape $N \times 512$ with N examples. To improve the convergence speed and performance for training our MLP model, we normalise the tactile and visual datasets with zero mean and unit variance. This also prevents the initialisation of large weights during back-propagation.

5.2.5 MLP Overview and Architecture

We define our MLP model in PyTorch and tailor it to predict grasp stability labels given a varying dataset size (depending on the dataset combination we use). This is achieved by dynamically declaring the input size of the network through its `input_size` parameter, allowing it to accommodate different dataset combinations for training. The MLP architecture consists of three fully connected layers, including two hidden layers with ReLU activation functions, and an output layer without an activation function.

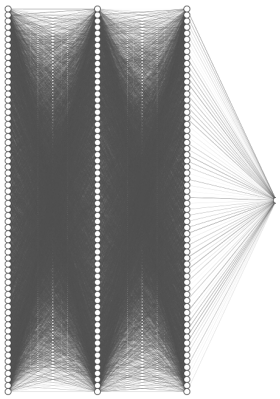


Figure 5.3: 3-Layer MLP architecture. The input size of the model is flexible depending on the dimensions of the input dataset (the input size in the figure is 64 neurons for simplicity). The remaining layers consist of 64 neurons. The MLP outputs a binary label in the output layer, corresponding to the grasp stability prediction.

5.3 Experiments on Impacts of Various Features

In this section, we present the results of some additional experiments that evaluate the robustness of the MLP model to primitive object classes and the influence of sample size on the model's performance. All models are trained for 500 epochs.

5.3.1 Robustness of MLP Model to Primitive Object Classes

To test the robustness of the MLP model to each primitive object class (rectangular box, cylinder, bottle), we trained the MLP model on two of its object variations and tested it on the remaining variation. The results are shown in the following table:

Training Objects	Testing Object	Test Accuracy (%)
Box 1, Box 2	Box 3	54.09±1.32
Box1, Box 3	Box 2	69.68±1.01
Box2, Box 3	Box 1	53.96±1.22
Bottle 1, Bottle 2	Bottle 3	53.96±1.29
Bottle 1, Bottle 3	Bottle 2	62.89±2.23
Bottle 2, Bottle 3	Bottle 1	50.31±1.00
Cylinder 1, Cylinder 2	Cylinder 3	47.55±1.30
Cylinder 1, Cylinder 3	Cylinder 2	61.13±1.66
Cylinder 2, Cylinder 3	Cylinder 1	61.76±3.17
Average		57.26±1.71

Table 5.2: Mean and standard deviation of MLP accuracy (%) by object class across 5 trials

As evident from our analysis, the MLP model's precision experienced a considerable decrement in performance when compared to its training on all object variations. This observed incongruity in model output can be ascribed to the deficiency of geometric features in distinguishing between different object variations belonging to the same object class. That is, the principal curvatures prove inadequate in encapsulating intra-class object variability, but rather effectively represent the variation exhibited between different object classes. This outcome is not unexpected, as our MLP is primed to forecast grasp stability labels for all primitive object types.

5.3.2 Influence of Sample Size of Randomly-Shuffled Dataset on Model Performance

This experiment aims to investigate the effect of sample size on the accuracy of our MLP model. Specifically, this approach aims to achieve a balance between the size of the training dataset, which is randomly shuffled without duplication prior to model training, and the desired accuracy of the model to mitigate the occurrence of overfitting or underfitting.

The investigation of the impact of sample size on the accuracy of the MLP models may also yield valuable insights into the generalisability and robustness of the model.

In general, as the size of the training data set increases, the performance of the MLP model improves due to its ability to learn more complex patterns in the data. However, adding more data may also increase the computational cost of training the model, and there may be a point of diminishing returns where adding more data does not lead to a significant improvement in accuracy. Moreover, if the size of the dataset is too small, the MLP model may suffer from overfitting, which occurs when the model learns to fit the training data too closely and is unable to generalise well to new data.

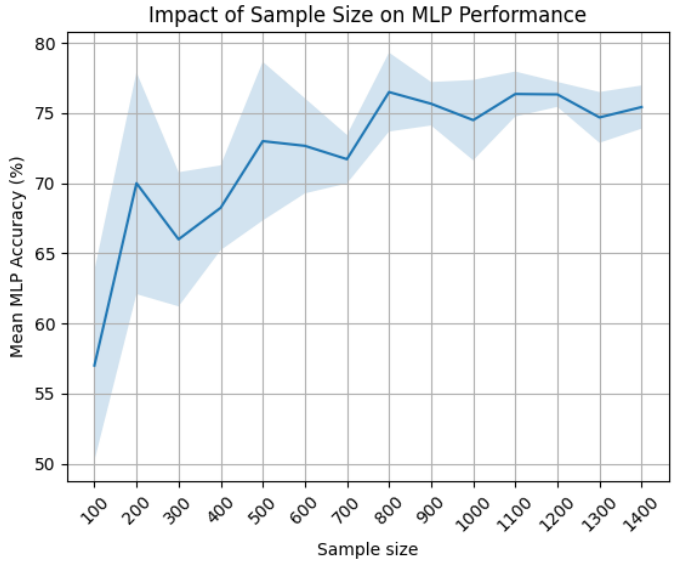


Figure 5.4: Impact of the sample size of the randomised dataset on MLP accuracy

As depicted in Figure 5.4, the accuracy of the MLP models increases with an increase in the sample size of the randomised dataset, which aligns with theoretical expectations. The MLP accuracy converges after approximately a sample size of 800 examples, thus the amount of data in this experiment is sufficient based on this analysis.

5.4 Training and Evaluation of Initial Results

In this section, we explore the effectiveness of our Multi-Layer Perceptron (MLP) model in predicting grasp stability labels. Three distinct training methods were implemented and evaluated using the same dataset discussed in Section 5.2.2.

The first method trained the MLP model on the aforementioned dataset to establish a baseline for its predictive performance. Subsequently, in the second method, the MLP model was trained on the same dataset concatenated with object features for each of the

9 object variations. This was done to investigate the impact of object features on the model's predictive performance. Finally, the third method involved applying Principal Component Analysis (PCA) to the dataset and re-training the MLP model with an input size of 5 principal components. This approach aimed to explore the effectiveness of dimensionality reduction in enhancing the model's accuracy.

Our dataset from Section 5.2.4 was split into training and validation sets with an 80:20 ratio using the `train_test_split()` function. The MLP model was trained using the Binary Cross-Entropy (BCE) logit loss function, which is commonly employed in binary classification tasks, and the Adam optimiser, a popular optimisation algorithm for training neural networks, with a learning rate of 0.001. The BCE logit loss function measures the difference between the predicted output of the model and the true label, penalising the model more for predicting the wrong label with high confidence. On the other hand, the Adam optimiser combines the benefits of two other optimisation methods: AdaGrad and RMSProp.

The validation set was used to evaluate the performance of the trained MLP model on unseen data. The results of the study are indicative of the effectiveness of the proposed training method in predicting grasp stability labels. The model's generalisation ability was assessed by validating it on unseen data, ensuring that it did not over-fit the training data.

5.4.1 Evaluation of Results

Our model is primarily trained on the dataset consisting all data sources, that is, tactile and visual (end effector poses and geometric data). For comparison of its effectiveness, we also train it on other variations of the datasets for 500 epochs. The results are as follows:

	Models				
	Tactile only	Visual only (without geometric)	Visual only (with geometric)	Both (tactile, visual)	Full (tactile, visual, geometric)
Accuracy (%)	71.81±1.92	62.22±2.19	62.36±1.96	74.17±2.37	74.86±1.36
Precision (%)	70.08±2.63	62.97±4.06	61.07±1.31	76.10±3.84	76.43±1.16
Recall (%)	76.27±1.58	62.64±3.45	63.71±3.04	73.10±1.94	75.42±2.64
F1 Score (%)	73.01±1.58	62.75±3.23	62.32±1.61	74.54±2.60	75.91±1.84

Table 5.3: Performance of various MLP models (average across 5 trials), 500 epochs

The results indicated that adding object features to the dataset marginally improved the MLP model's predictive accuracy. Based on these results, we attempt to apply PCA

to the complete dataset and reduce the input size to 5 principal components. Figure 5.5 shows the accuracies of our MLP model on varying numbers of principal components k from applying PCA across 5 separate trials:

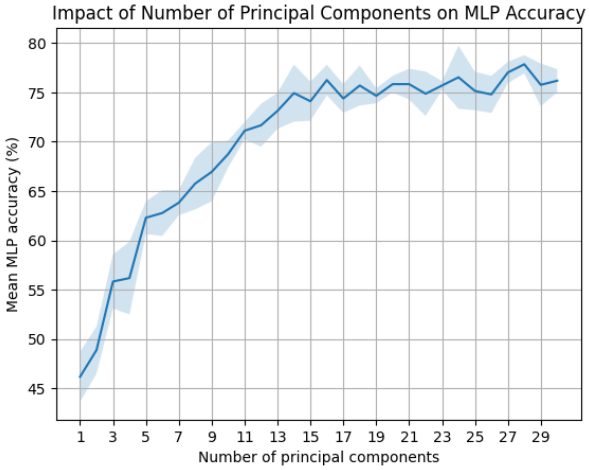


Figure 5.5: Investigating the impact of number of principal components k on the mean MLP validation accuracy across 5 trials

Mean accuracy	77.85±0.91	77.02±1.09	76.53±3.19	76.25±1.55	75.83±1.17
k	28	27	24	16	30

Table 5.4: Top 5 values and corresponding indices for PCA accuracies obtained with different numbers of components ($k = 1$ to $k = 30$)

Since this experimentation aims to balance the trade-off between dataset dimensionality and accuracy, we disregard the corresponding model for $k = 28$ and conclude that $k = 16$ principal components results in the attainment of the most elevated mean validation accuracy (76.25% on average across 5 trials) utilising our MLP model. This selection represents a judicious choice, as it reduces the dimensionality of our dataset from (1440, 512) to (1440, 16), while preserving and even enhancing the general level of accuracy.

5.4.2 Limitations

This study was subject to the relatively modest sample size of the dataset we collected. This shortcoming may have exerted an impact on the accuracy of our findings. The source of this limitation is primarily attributed to the data collection pipeline in Pybullet, which demands a substantial amount of time to gather adequate data for the purpose of training. This arises from our approach of generating random hand configurations for grasping, and imposing constraints on the collection process to yield a balanced dataset (namely, an equal number of successful and unsuccessful grasps).

As a reference point, we include in the following table a summary of the outcomes of several exemplary studies in the realm of grasp stability prediction, for the purpose of benchmarking our model's performance.

Paper & Author	Type(s) of Data	Dataset Size	No. of Objects	Approach	Accuracy
Garcia et al. (2019) [18]	Tactile	5,831	41	Graph Convolutional Network with k-NN	75.53 (without k-NN); 92.7 (with k-NN)
Chebotar et al. (2017) [9]	Tactile	3,351	3	ST-HMP, REPS, Neural Net	90.43 (Known objects); 80.7 (Unknown objects)
Bekiroglu et al. (2011) [3]	Tactile, visual	342	2	Kernel Logistic Regression	89.46
Chumbley et al. (2022) [12]	Tactile, vision	10,000	20	CNN with ResNet-18 back-bone	82.7±0.6 (Known objects); 61.3±2.8 (Unknown objects)
Calandra et al. (2018) [7]	Tactile, vision	18,070	87	CNN with MLP	80.28±0.68
Our work	**Tactile, vision**	**1,440**	**9**	**MLP**	**76.3±0.66**

Table 5.5: A comparison of performance of various grasp stability prediction models

Furthermore, Pybullet simulations may not perfectly emulate the complexity and variability of real-world scenarios. Although Pybullet provides a robust physics engine, it may lack certain nuances and intricacies that exist in real environments, such as variations in material properties, surface roughness, or lighting conditions. Consequently, the tactile data collected in the simulation may not fully capture the range of tactile interactions and sensory feedback observed in reality. Moreover, the compliance, elasticity and noise characteristics of the DIGIT tactile sensors may not be adequately modelled in the simulation, further limiting the authenticity of the tactile data acquired.

5.5 Conclusion

The purpose of this section was to train and evaluate a multi-modal multi-layer perceptron (MLP) utilising primarily multi-modal dataset combinations while examining the impact of object geometric features, particularly principal curvature, on its overall performance. The outcomes of the study indicate that the inclusion of object geometric features in our multi-modal dataset (tactile and visual) produced a minor positive impact on the MLP's accuracy, achieving an average accuracy of 76.25%. However, the MLP's robustness was not significantly demonstrated regarding in-class variation based on the results obtained.

Chapter 6

Conclusions and Future Work

This study aimed to evaluate the effectiveness and robustness of multi-modal sensory data on the performance of grasp learning models, as well as the impact of object geometric features on the performance of such models. The grasp stability predictions have shown promising results in the previous chapter, indicating the potential for further development. We achieved an average accuracy of 76.25% across a multi-modal dataset consisting of tactile sensory readings, end effector poses as well as object-specific geometric features based on 9 object variations. Our experiments have also demonstrated a sufficient degree of robustness in terms of the model's responsiveness to different object variations.

Further improvements can be made regarding model complexity, dataset sample sizes, and approaches to classifying grasp stability. The primary concern that future studies should tackle is the representation of object models and the collection of data through simulations. Pybullet employs a simplified physics model for simulating physical bodies, which can result in inaccuracies in the simulation, especially for highly detailed and complex objects like bottles. In addition, it should be noted that Pybullet is no longer actively maintained and may not receive support in the future. To ensure compatibility with the latest technologies and continued support and updates, it is recommended to upgrade to more stable and actively maintained frameworks and physics engines such as MuJoCo and ROS (Robot Operating System), both of which are widely used in robotics research. MuJoCo is known for its stability and accuracy in physics simulation, while ROS provides a comprehensive set of tools and libraries for robotics research. Upgrading to these platforms will allow the project to benefit from the latest advancements in robotics research and explore new features and functionalities that were not previously available in Pybullet.

To evaluate the performance of the MLP neural network model in real-world scenarios, it is crucial to test the model on real data collected from a physical robot setup. Therefore, it is recommended to gather real-world data and assess the MLP's performance on it. This will provide a better understanding of the model's performance, identify potential issues

when deploying the model in the real world, and determine its robustness and reliability in different environments and conditions.

Although principal curvatures are a powerful tool for feature extraction in object recognition and segmentation, the current implementation may not be the best approach for clustering primitive object types. To achieve better clustering results for these types of objects, it is recommended to explore alternative methods of principal curvature implementation, such as alternative feature extraction techniques like deep learning-based methods.

Given that neural networks can predict other grasping properties, such as force distribution, grasp robustness, and grasp efficiency, it is recommended to explore the possibility of developing a multi-task learning framework that can predict multiple grasping properties simultaneously. A multi-task learning framework can provide a more comprehensive understanding of the grasping process by predicting multiple grasping properties, improving overall prediction accuracy, and reducing the need for extensive data labelling for individual tasks. Careful consideration of different architectures for multi-task learning is necessary to evaluate their performance against single-task learning models and understand the benefits and drawbacks of multi-task learning.

Bibliography

[1] Rami Alkhatib, Wajih Mechlawi, and Rabab Kawtharani. "Quality Assessment of Robotic Grasping Using Regularized Logistic Regression". In: *IEEE Sensors Letters* 4.6 (2020), pp. 1–4. DOI: 10.1109/LSENS.2020.2994166.

[2] Yasemin Bekiroglu. *Learning to Assess Grasp Stability from Vision, Touch and Proprioception*. 2012.

[3] Yasemin Bekiroglu, Renaud Detry, and Danica Kragic. "Learning tactile characterizations of object- and pose-specific grasps". In: *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2011, pp. 1554–1560. DOI: 10.1109/IROS.2011.6094878.

[4] Lars Berscheid, Pascal Meißner, and Torsten Kröger. *Self-supervised Learning for Precise Pick-and-place without Object Model*. 2020. arXiv: 2006.08373 [cs.RO].

[5] Jeannette Bohg et al. "Data-Driven Grasp Synthesis—A Survey". In: *IEEE Transactions on Robotics* 30.2 (Apr. 2014), pp. 289–309. DOI: 10.1109/tro.2013.2289018. URL: https://doi.org/10.1109%2Ftro.2013.2289018.

[6] Michel Breyer et al. "Volumetric Grasping Network: Real-time 6 DOF Grasp Detection in Clutter". In: *Conference on Robot Learning*. 2020.

[7] Roberto Calandra et al. "More Than a Feeling: Learning to Grasp and Regrasp Using Vision and Touch". In: *IEEE Robotics and Automation Letters* 3.4 (Oct. 2018), pp. 3300–3307. DOI: 10.1109/lra.2018.2852779. URL: https://doi.org/10.1109/LRA.2018.2852779.

[8] U. Castiello. *The neuroscience of grasping*. 2005. DOI: https://doi.org/10.1038/nrn1744.

[9] Yevgen Chebotar et al. "Generalizing Regrasping with Supervised Policy Learning". In: Mar. 2017, pp. 622–632. ISBN: 978-3-319-50114-7. DOI: 10.1007/978-3-319-50115-4_54.

[10] Sachin Chitta, Matthew Piccoli, and Jürgen Sturm. "Tactile object class and internal state recognition for mobile manipulation". In: June 2010, pp. 2342–2348. DOI: 10.1109/ROBOT.2010.5509923.

[11] Han-Pang Chiu et al. “Class-specific grasping of 3D objects from a single 2D image”. In: *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2010, pp. 579–585. DOI: 10.1109/IROS.2010.5652597.

[12] Lachlan Chumbley et al. *Integrating High-Resolution Tactile Sensing into Grasp Stability Prediction*. 2022. arXiv: 2206.05714 [cs.RO].

[13] Erwin Coumans and Yunfei Bai. *PyBullet, a Python module for physics simulation for games, robotics and machine learning*. http://pybullet.org. 2016–2021.

[14] Michael Danielczuk et al. *Exploratory Grasping: Asymptotically Optimal Algorithms for Grasping Challenging Polyhedral Objects*. 2020. arXiv: 2011.05632 [cs.RO].

[15] Dan Ding, Yun Liu, and Shuguo Wang. “Computing 3-D optimal form-closure grasps”. In: vol. 4. Feb. 2000, 3573–3578 vol.4. ISBN: 0-7803-5886-4. DOI: 10.1109/ROBOT.2000.845288.

[16] Cristiana de Farias et al. “Simultaneous Tactile Exploration and Grasp Refinement for Unknown Objects”. In: *IEEE Robotics and Automation Letters* 6.2 (Apr. 2021), pp. 3349–3356. DOI: 10.1109/lra.2021.3063074. URL: https://doi.org/10.1109/LRA.2021.3063074.

[17] Peter I. Frazier. *A Tutorial on Bayesian Optimization*. 2018. arXiv: 1807.02811 [stat.ML].

[18] Alberto Garcia-Garcia et al. *TactileGCN: A Graph Convolutional Network for Predicting Grasp Stability with Tactile Sensors*. 2019. arXiv: 1901.06181 [cs.LG].

[19] Sebastian Geidenstam et al. “Learning of 2D grasping strategies from box-based 3D object approximations”. In: June 2009. DOI: 10.15607/RSS.2009.V.002.

[20] Luis González-Jiménez et al. “Robust Pose Control of Robot Manipulators Using Conformal Geometric Algebra”. In: *Advances in Applied Clifford Algebras* 24 (June 2014). DOI: 10.1007/s00006-014-0448-2.

[21] Nicolas Gorges et al. “Haptic Object Recognition using Passive Joints and Haptic Key Features”. In: May 2010, pp. 2349–2355. DOI: 10.1109/ROBOT.2010.5509553.

[22] Di Guo et al. “Robotic grasping using visual and tactile sensing”. In: *Information Sciences* 417 (2017), pp. 274–286. ISSN: 0020-0255. DOI: https://doi.org/10.1016/j.ins.2017.07.017. URL: https://www.sciencedirect.com/science/article/pii/S002002551730837X.

[23] Robert Haschke et al. “Task-oriented quality measures for dextrous grasping”. In: July 2005, pp. 689–694. ISBN: 0-7803-9355-4. DOI: 10.1109/CIRA.2005.1554357.

[24] Kaijen Hsiao, Leslie Kaelbling, and Tomás Lozano-Pérez. “Task-Driven Tactile Exploration”. In: June 2010. DOI: 10.15607/RSS.2010.VI.029.

[25] Eric Jang et al. *End-to-End Learning of Semantic Grasping*. 2017. arXiv: 1707.01932 [cs.RO].

[26] A.R. Jiménez et al. "Featureless classification of tactile contacts in a gripper using neural networks". In: *Sensors and Actuators A: Physical* 62.1 (1997), pp. 488–491. ISSN: 0924-4247. DOI: https://doi.org/10.1016/S0924-4247(97)01496-9. URL: https://www.sciencedirect.com/science/article/pii/S0924424797014969.

[27] Kleeberger K., Bormann R., and Kraus W. et al. *A Survey on Learning-Based Robotic Grasping*. 2020. URL: https://doi.org/10.1007/s43154-020-00021-6.

[28] Sulabh Kumra and Christopher Kanan. "Robotic grasp detection using deep convolutional neural networks". In: *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2017, pp. 769–776. DOI: 10.1109/IROS.2017.8202237.

[29] Mike Lambeta et al. "DIGIT: A Novel Design for a Low-Cost Compact High-Resolution Tactile Sensor with Application to In-Hand Manipulation". In: *IEEE Robotics and Automation Letters (RA-L)* 5.3 (2020), pp. 3838–3845. DOI: 10.1109/LRA.2020.2977257.

[30] Ian Lenz, Honglak Lee, and Ashutosh Saxena. *Deep Learning for Detecting Robotic Grasps*. 2014. arXiv: 1301.3592 [cs.LG].

[31] Sandra Q. Liu and Edward H. Adelson. *GelSight Fin Ray: Incorporating Tactile Sensing into a Soft Compliant Robotic Gripper*. 2022. arXiv: 2204.07146 [cs.RO].

[32] Jeffrey Mahler et al. *Dex-Net 2.0: Deep Learning to Plan Robust Grasps with Synthetic Point Clouds and Analytic Grasp Metrics*. 2017. arXiv: 1703.09312 [cs.RO].

[33] José Nogueira et al. *Unscented Bayesian Optimization for Safe Robot Grasping*. 2016. arXiv: 1603.02038 [cs.RO].

[34] Lerrel Pinto and Abhinav Gupta. *Supersizing Self-supervision: Learning to Grasp from 50K Tries and 700 Robot Hours*. 2015. arXiv: 1509.06825 [cs.LG].

[35] Robert Platt. *Grasp Learning: Models, Methods, and Performance*. 2022. arXiv: 2211.04895 [cs.RO].

[36] A. Sahbani, S. El-Khoury, and P. Bidaud. "An overview of 3D object grasp synthesis algorithms". In: *Robotics and Autonomous Systems* 60.3 (2012). Autonomous Grasping, pp. 326–336. ISSN: 0921-8890. DOI: https://doi.org/10.1016/j.robot.2011.07.016. URL: https://www.sciencedirect.com/science/article/pii/S0921889011001485.

[37] Schneider et al. "Object Identification with Tactile Sensors using Bag-of-Features". In: Dec. 2009, pp. 243–248. DOI: 10.1109/IROS.2009.5354648.

[38] Abdulrahman Al-Shanoon et al. *Learn to grasp unknown objects in robotic manipulation*. 2021. DOI: 10.1007/s11370-021-00380-9. URL: https://doi.org/10.1007/s11370-021-00380-9.

[39] Zilin Si et al. *Grasp Stability Prediction with Sim-to-Real Transfer from Tactile Sensing*. 2022. arXiv: 2208.02885 [cs.RO].

[40] Wang, Shaoxiong and Lambeta, Mike and Chou, Po-Wei and Calandra, Roberto. "TACTO: A Fast, Flexible, and Open-source Simulator for High-resolution Vision-based Tactile Sensors". In: *IEEE Robotics and Automation Letters (RA-L)* 7.2 (2022), pp. 3930–3937. ISSN: 2377-3766. DOI: 10.1109/LRA.2022.3146945. URL: https://arxiv.org/abs/2012.08456.

[41] Eric W. Weisstein. *Euler Angles. From MathWorld–A Wolfram Web Resource.* URL: https://mathworld.wolfram.com/EulerAngles.html.

[42] Wikipedia contributors. *Principal curvature — Wikipedia, The Free Encyclopedia.* [Online; accessed 12-April-2023]. 2023. URL: https://en.wikipedia.org/w/index.php?title=Principal_curvature&oldid=1141654906.

[43] Hanbo Zhang et al. *Robotic Grasping from Classical to Modern: A Survey*. 2022. arXiv: 2202.03631 [cs.RO].

Appendix A

System Flowchart

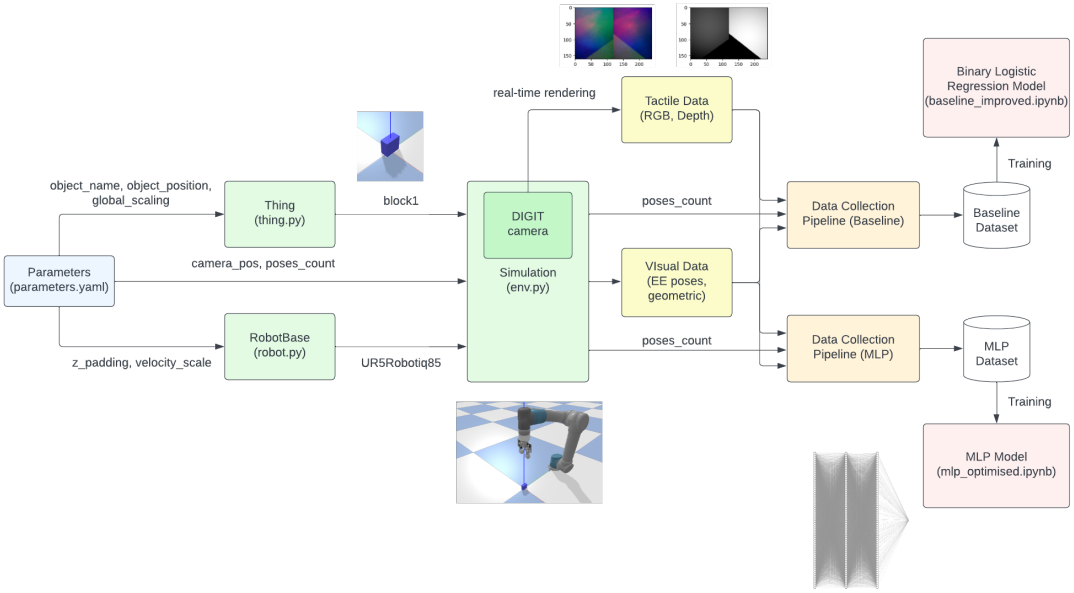


Figure A.1: A graphical representation of the project's full system, including simulation, data collection and modelling components

Appendix B

System Manual

B.1 Prerequisites

We recommend downloading the zip file containing all relevant code for this project separately either from the submission or this link. The repository is anonymous. Before compiling and testing the listed code in Appendix 3, please ensure that your system has been properly configured with the following requirements:

1. Linux Operating System (preferably Ubuntu 20.04). The code for this project was only developed and tested on Ubuntu 20.04. It is not guaranteed to compile and run fully functionally on other operating system and devices.
2. Python 3.8.16 is installed.
3. The `virtualenv` package for creating Python virtual environments is installed.

B.2 Code Deployment

Please follow these steps to deploy the project code:

1. Clone the repository from https://github.com/uclzzjn7/UCL-FYP.git or download a zip file containing the repository code.
2. If you already have Python 3.8.16 installed on your OS, please skip to step 4.
3. In a new terminal, run the following commands in sequence:

```
sudo apt update
```

```
sudo apt install software-properties-common
```

```
sudo add-apt-repository ppa:deadsnakes/ppa
```

```
sudo apt-cache policy python3.8
```

This should install Python 3.8.16.

4. Find the root path of Python 3.8.16 in your OS:

```
ls /usr/bin/python*
```

5. Create a Python virtual environment with Python 3.8.16:

```
virtaulenv venv --python="<path of Python 3.8.16>"
```

Assuming that the path of Python 3.8.16 is "/usr/bin/python3.8", then the command for creating a virtual environment should be:

```
virtualenv venv --python="/usr/bin/python3.8"
```

Ensure that the virtual environment is created in the root directory of the project:

```
/docs
/src
/venv
    /bin
    activate
    ...
```

6. Activate the virtual environment:

```
source venv/bin/activate
```

7. Install `PyTorch`:

```
pip3 install torch==1.13.1+cu116 torchaudio==0.13.1+cu116
    torchvision==0.14.1+cu116 --extra-index-url
    https://download.pytorch.org/whl/cu116 --no-cache-dir
```

8. Install all required Python packages:

```
pip install -r requirements.txt
```

9. Run the Pybullet simulation:

```
python src/main.py
```

B.3 Using the Simulation

There are several core functionalities provided by our Pybullet simulation. These functionalities are listed as either sliders or buttons on the right panel of the simulation GUI.

- **Manual manipulation of the robot**: Adjust the six available sliders for the end effector pose (x,y,z,roll,pitch,yaw) and the slider for the gripper length (0 = gripper is completely closed, maximum = gripper is completely open).
- **Reset simulation**: Resets all sliders and the robot (end effector position).
- **Get joint coordinates**: Retrieves all movable joints of the robot arm and gripper.
- **Get object features**: Retrieves the object features of the created object mesh in the simulation. Currently this function returns the 3 principal curvatures of the object, one for each dimension (width, height, depth).
- **Collect data (baseline)**: Data collection pipeline to collect data for the baseline approach. To run this function, please ensure the `object_name` property is set to `block` in the `parameters.yaml` file. This function will not work for any other object name since manually-selected end effector poses were only collected for block.
- **Collect data (proposed)**: Data collection pipeline to collect data for the proposed approach (MLP). To run this function, please ensure the `object_name` property is set to one of the following in the `parameters.yaml` file:

```
object_name: block1/block2/block3/bottle1/bottle2/bottle3/
cylinder1/cylinder2/cylinder3
```

B.4 Training Models

The baseline and proposed models for this project are trained separately from the simulation code since it requires a significantly larger amount of computing power. Before training, please download the `datasets` folder from this link:
https://drive.google.com/drive/folders/1d14Ul5YTjX-6w56OCxLbCXTLicDE3qnB?usp=sharing
This folder contains all required datasets for training the models. Extract all the contents of the `datasets` folder to under the `src/` directory (i.e. `src/datasets/`).

The relevant code for training can be found in the Jupyter notebooks (with the `.ipynb` file extension) in the `models/baseline_model` or `models/mlp_model` directories. If this code needs to be assessed, please ensure that these notebooks are run with `cuda` enabled, and the correct Python interpreter is selected for running the notebook (Python 3.8.16). Otherwise, there is a risk of insufficient memory allocation to training the models.

Appendix C

Code Listing

Due to the code listing page limit, this section only documents the intriguing bits of code snippets for the project. The complete code base with the simulation code, training datasets, etc. can be found via https://github.com/uclzzjn7/UCL-FYP.git.

C.1 Baseline Model Training Code

```
## UCL COMP0029 Individual Project for Year 3 BSc
### Robust Robotic Grasping Utilising Touch Sensing - Baseline Approach Notebook
### This notebook contains the code for developing a baseline approach to
    grasping using classifiers: given some combinations of tactile data, end
    effector poses relative to the robot hand (visual data), etc., determine
    whether these constraints will produce a successful/unsuccessful grasp.
### 1. Load packages
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from mpl_toolkits.mplot3d.art3d import Poly3DCollection

import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.init as init
import gc

from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, confusion_matrix,
    ConfusionMatrixDisplay
from sklearn.decomposition import PCA

from math import sin, cos
```

```
import seaborn as sns
# Set device for ‘PyTorch‘ training
# Use GPU if available, else CPU
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
device
# Empty PyTorch cache
gc.collect()
torch.cuda.empty_cache()
# Set numpy seed
np.random.seed(seed=1)
### 2. Load datasets from saved .npy files
# To collect data for this experiment, you can run the "Collect Sensory Data"
    button in the Pybullet simulation. This generates a predefined number of
    Gaussian grasps randomly generated from a base hand pose. Each individual
    grasp is considered as an individual experiment, and the data collected from
    this experiment is split into four, each stored in its own dataset.

# For all object models used in this experiment, each object has 4 datasets
    which include:
# - ‘depth_ds.npy‘ which stores the depth tactile data from the mounted DIGIT
    sensors
# - ‘color_ds.npy‘ which stores the colored (RGB) version of the depth tactile
    data from the mounted DIGIT sensors
# - ‘poses_ds.npy‘ which stores the randomly-generated 6d hand poses from the
    simulation
# - ‘outcomes_ds.npy‘ which stores the outcomes of each random pose
root = "../models/baseline_model/"
object_name = "block"    # object_name should be in [bottle, block, mug]

depth_data = np.load(root + object_name + "/depth_ds.npy")
color_data = np.load(root + object_name + "/color_ds.npy")
poses_data = np.load(root + object_name + "/poses_ds.npy")
grasp_outcomes_data = np.load(root + object_name + "/grasp_outcomes.npy")
# These datasets should all be in the form of $(N\times...)$ where $N$ is the
    number of examples:
print(f"Shape of depth_data: {depth_data.shape}")
print(f"Shape of color_data: {color_data.shape}")
print(f"Shape of poses_data: {poses_data.shape}")
print(f"Shape of grasp_outcomes_data: {grasp_outcomes_data.shape}")
# Additionally, we confirm the number of successful and unsuccessful grasps
    recorded. This helps us in the next section to determine how many examples
    we should include for each class in order to produce a balanced dataset.
print(f"# of sucessesful grasps: {(grasp_outcomes_data == 1).sum()}")
print(f"# of unsuccessful grasps: {(grasp_outcomes_data == 0).sum()}")
```

```
### 3. Data preprocessing
# We will train a logistic regression classifier on 3 combinations of data:
# - **Tactile** only: concatenated and flattened depth and color data
# - **Visual** only: 6D end effector poses consisting of position (x,y,z) and
    orientation (r,p,y) data
# - **Both**: concatenated and flattened tactile and visual data
# We define some reusable functions for later:
def normalize(arr):
    # Normalize & standardize each column
    mean = np.mean(arr, axis=0)
    std = np.std(arr, axis=0)

    arr = (arr - mean) / std
    arr = np.nan_to_num(arr, 0)
    arr[np.isinf(arr)] = 0
    return arr
# Since each tactile reading (depth and color) is a pair of images (one on each
    finger), we concatenate them together as a single 160x240 image.
depth_data = np.concatenate((depth_data[:, 0], depth_data[:, 1]), axis=2)
color_data = np.concatenate((color_data[:, 0], color_data[:, 1]), axis=2)
print(f"Shape of depth_data: {depth_data.shape}")
print(f"Shape of color_data: {color_data.shape}")

depth_ds = torch.from_numpy(normalize(depth_data))
color_ds = torch.from_numpy(normalize(color_data))
visual_ds = torch.from_numpy(np.nan_to_num(normalize(poses_data)))
visual_ds
# We then concatenate the depth and color datasets to produce the flattened
    tactile dataset:
tactile_ds = torch.cat([depth_ds.unsqueeze(-1), color_ds], dim=-1)
tactile_ds = torch.nan_to_num(tactile_ds)
complete_ds = torch.cat([tactile_ds.reshape(tactile_ds.shape[0], -1),
    visual_ds], dim=1)
complete_ds = torch.nan_to_num(complete_ds)
tactile_ds.shape, complete_ds.shape
### 4. Dataset visualisation (optional)
# Only run this section to produce figures and plots for the project report.
Z_PADDING = 0.2      # Adjust for visualisation of end poses


def plot_ee_pose(pose_data, grasp_outcome, ax, orientation):
    x, y, z, alpha, beta, gamma = pose_data
    z -= Z_PADDING * 1.6

    # create rotation matrix based on Euler angles
```

```
    Rx = np.array([[1, 0, 0], [0, cos(alpha), sin(alpha)], [0, -sin(alpha),
        cos(alpha)]])
    Ry = np.array([[cos(beta), 0, -sin(beta)], [0, 1, 0], [sin(beta), 0,
        cos(beta)]])
    Rz = np.array([[cos(gamma), sin(gamma), 0], [-sin(gamma), cos(gamma), 0],
        [0, 0, 1]])
    R = Rz.dot(Ry.dot(Rx))

    # calculate endpoints of line based on orientation
    vec = np.array([0, 0.015, 0])
    vec_rotated1 = R.dot(vec)
    endpoint1 = [x + vec_rotated1[0], y + vec_rotated1[1], z + vec_rotated1[2]]

    vec_rotated2 = R.dot(-vec)
    endpoint2 = [x + vec_rotated2[0], y + vec_rotated2[1], z + vec_rotated2[2]]

    # set midpoint to the actual point
    midpoint = [x, y, z]

    # plot line through point in orientation direction
    ax.plot([endpoint1[0], endpoint2[0]], [endpoint1[1], endpoint2[1]],
        [endpoint1[2], endpoint2[2]], color='green' if grasp_outcome==1 else
        'red', zorder=20)

    # plot vertical lines starting from endpoints
    ax.plot([endpoint1[0], endpoint1[0]], [endpoint1[1], endpoint1[1]],
        [endpoint1[2], endpoint1[2]-0.035], color='green' if grasp_outcome==1
        else 'red', zorder=20)
    ax.plot([endpoint2[0], endpoint2[0]], [endpoint2[1], endpoint2[1]],
        [endpoint2[2], endpoint2[2]-0.035], color='green' if grasp_outcome==1
        else 'red', zorder=20)

    # Display plot viewing orientation
    ax.view_init(elev=orientation[0], azim=orientation[1], roll=orientation[2])


# Draw block object
def plot_3d_box(ax, orientation):
    # Dimensions of the box object: (W=0.025, H=0.05, D=0.05)
    # Dimensions for unit box: (0,0,0), (0,1,0), (1,0,0), (0,0,1)
    cube_definition = [(-0.0125, -0.025, 0), (0.0125, -0.025, 0), (-0.0125,
        0.0255, 0), (-0.0125, -0.025, 0.05)]
    cube_definition_array = [np.array(list(item)) for item in cube_definition]
    points = []
    points += cube_definition_array
```

```
    vectors = [
        cube_definition_array[1] - cube_definition_array[0],
        cube_definition_array[2] - cube_definition_array[0],
        cube_definition_array[3] - cube_definition_array[0]
    ]
    points += [cube_definition_array[0] + vectors[0] + vectors[1]]
    points += [cube_definition_array[0] + vectors[0] + vectors[2]]
    points += [cube_definition_array[0] + vectors[1] + vectors[2]]
    points += [cube_definition_array[0] + vectors[0] + vectors[1] + vectors[2]]
    points = np.array(points)
    edges = [
        [points[0], points[3], points[5], points[1]],
        [points[1], points[5], points[7], points[4]],
        [points[4], points[2], points[6], points[7]],
        [points[2], points[6], points[3], points[0]],
        [points[0], points[2], points[4], points[1]],
        [points[3], points[6], points[7], points[5]]
    ]
    faces = Poly3DCollection(edges, linewidths=1, edgecolors='k')
    faces.set_facecolor((0,0,1,0.1))
    faces.set_zorder(3)
    ax.add_collection3d(faces)
    # Plot the points themselves to force the scaling of the axes
    ax.scatter(points[:,0], points[:,1], points[:,2], c=points[:,2], s=0)
    ax.set_aspect('equal')
    ax.view_init(elev=orientation[0], azim=orientation[1], roll=orientation[2])
#### 4a. Visualise tactile and visual data for 1 successful and 1 unsuccessful
    grasp
successful_grasps = np.where(grasp_outcomes_data == 1)[0]
unsuccessful_grasps = np.where(grasp_outcomes_data == 0)[0]

# Randomly select one index from each set
successful_rand_idx = np.random.choice(successful_grasps)
unsuccessful_rand_idx = np.random.choice(unsuccessful_grasps)

rand_indices = np.array([successful_rand_idx, unsuccessful_rand_idx])


# Note that the plots use the "..._data" datasets instead of the "..._ds"
    datasets since
# the "..._ds" datasets are already flattened for training
for i in range(2):
    fig = plt.figure(figsize=(14, 3))
    # fig.suptitle("Successful grasp" if
        grasp_outcomes_data[rand_indices[i]].item() == 1.0 else "Unsuccessful
```

```
        grasp")

    ax1 = fig.add_subplot(1, 3, 1)
    ax1.set_xlabel("Depth")
    ax1.imshow(np.array(depth_data[rand_indices[i]]), cmap='gray')

    ax2 = fig.add_subplot(1, 3, 2)
    ax2.set_xlabel("Colour")
    ax2.imshow(np.array(color_data[rand_indices[i]]) / 255., cmap='gray')

    ax3 = fig.add_subplot(1, 3, 3, projection='3d')
    ax3.set_xlabel('x position')
    ax3.set_ylabel('y position')
    ax3.set_zlabel('z position')
    ax3.set_xlim3d(-0.05, 0.05)
    ax3.set_ylim3d(-0.05, 0.05)
    ax3.set_zlim3d(0, 0.1)

    ax3.spines['top'].set_visible(False)
    ax3.spines['right'].set_visible(False)
    ax3.spines['bottom'].set_visible(False)
    ax3.spines['left'].set_visible(False)

    pose_data = poses_data[rand_indices]
    plot_ee_pose(pose_data=poses_data[rand_indices[i]],
        grasp_outcome=grasp_outcomes_data[rand_indices[i]], ax=ax3,
        orientation=(30, 45, 0))
    plot_3d_box(ax3, orientation=(30, 45, 0))

fig.subplots_adjust(wspace=0.3, hspace=0.3)
plt.show()
#### 4b. Visualise all end effector poses on box as skeleton hands
# Create 3 plots: 1 with the box only, remaining 2 with all poses each at
    different viewing orientations
for i in range(3):
    fig = plt.figure(figsize=(6, 6))
    ax = fig.add_subplot(111, projection='3d', computed_zorder=False)

    # Plot unit vectors
    x_unit = np.array([0.1, 0, 0])
    y_unit = np.array([0, 0.075, 0])
    z_unit = np.array([0, 0, 0.125])
    ax.plot([0, x_unit[0]], [0, x_unit[1]], [0, x_unit[2]], color='r')
    ax.plot([0, y_unit[0]], [0, y_unit[1]], [0, y_unit[2]], color='g')
    ax.plot([0, z_unit[0]], [0, z_unit[1]], [0, z_unit[2]], color='b')
```

```
    ax.text(x_unit[0], x_unit[1], x_unit[2], 'X')
    ax.text(y_unit[0], y_unit[1], y_unit[2], 'Y')
    ax.text(z_unit[0], z_unit[1], z_unit[2], 'Z')

    # Remove all the axis and their labels
    ax.axis('off')
    ax.set_aspect('equal')
    plot_3d_box(ax, orientation=(15, 30, 0))
    if i == 0:
        ax.set_xlim3d(-0.05, 0.05)
        ax.set_ylim3d(-0.05, 0.05)
        ax.set_zlim3d(0, 0.1)
    if i > 0:
        ax.set_xlim3d(-0.05, 0.05)
        ax.set_ylim3d(-0.05, 0.05)
        ax.set_zlim3d(0, 0.1)
        for i in range(poses_data.shape[0]):
            plot_ee_pose(pose_data=poses_data[i],
                grasp_outcome=grasp_outcomes_data[i], ax=ax, orientation=(15,
                30+90*(i-1), 0))
### 5. Model training
# We now train our Logistic Regression models on the 3 combinations of our data
    (tactile, visual, both):
# - Raw data
# - Principal Component Analysis - 2 main components
# - Convolutional Neural Network processed data
#### Prepare training and testing datasets
X_tactile_train, X_tactile_test, y_tactile_train, y_tactile_test =
    train_test_split(tactile_ds.reshape(tactile_ds.shape[0], -1),
    grasp_outcomes_data, test_size=0.2, random_state=0)
X_visual_train, X_visual_test, y_visual_train, y_visual_test =
    train_test_split(visual_ds, grasp_outcomes_data, test_size=0.2,
    random_state=0)
X_complete_train, X_complete_test, y_complete_train, y_complete_test =
    train_test_split(complete_ds, grasp_outcomes_data, test_size=0.2,
    random_state=0)
### 5.1 Raw data
#### 5.1.1 Raw data (tactile only) + LR
model_511 = LogisticRegression(random_state=0, max_iter=1000)
model_511.fit(X_tactile_train, y_tactile_train)
model_511_predictions = model_511.predict(X_tactile_test)
#### 5.1.2 Raw data (visual only) + LR
model_512 = LogisticRegression(random_state=0, max_iter=1000)
model_512.fit(X_visual_train, y_visual_train)
```

```
model_512_predictions = model_512.predict(X_visual_test)
#### 5.1.3 Raw data (both) + LR
model_513 = LogisticRegression(random_state=0, max_iter=1000)
model_513.fit(X_complete_train, y_complete_train)
model_513_predictions = model_513.predict(X_complete_test)
### 5.3 CNN for dimensionality reduction
# A simple convolutional neural network that extracts features from an input
    tensor
class FeatureExtractorCNN(nn.Module):
    def __init__(self):
        super(FeatureExtractorCNN, self).__init__()
        self.conv1 = nn.Conv2d(4, 8, kernel_size=3, stride=1, padding=1)
        self.conv2 = nn.Conv2d(8, 16, kernel_size=3, stride=1, padding=1)
        self.conv3 = nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1)

        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)


    def forward(self, x):
        # print(f"Input size: {x.shape}")
        x = self.conv1(x)
        # print(f"Shape after conv1: {x.shape}")
        x = nn.functional.relu(x)
        x = self.pool(x)
        # print(f"Shape after pool1: {x.shape}")

        x = self.conv2(x)
        # print(f"Shape after conv2: {x.shape}")
        x = nn.functional.relu(x)
        x = self.pool(x)
        # print(f"Shape after pool2: {x.shape}")

        x = self.conv3(x)
        # print(f"Shape after conv3: {x.shape}")
        x = nn.functional.relu(x)
        x = self.pool(x)
        # print(f"Shape after pool3: {x.shape}")

        return x
# Preprocess data using CNN feature extraction
cnn = FeatureExtractorCNN()
cnn_tactile = torch.cat([cnn(img.float().permute(2,0,1)).unsqueeze(0) for img in
    tactile_ds])
cnn_tactile = cnn_tactile.reshape(cnn_tactile.shape[0], -1)
cnn_tactile.shape
```

```
# We simply combine the cnn-processed tactile data (from Section 5.3.1) with the
    visual data
cnn_complete_ds = torch.cat([cnn_tactile.reshape(cnn_tactile.shape[0], -1),
    visual_ds], dim=1)
cnn_complete_ds.shape
X_cnn_tactile_train, X_cnn_tactile_test, y_cnn_tactile_train, y_cnn_tactile_test
    = train_test_split(cnn_tactile.detach().numpy(), grasp_outcomes_data,
    test_size=0.2, random_state=0)
X_cnn_complete_train, X_cnn_complete_test, y_cnn_complete_train,
    y_cnn_complete_test = train_test_split(cnn_complete_ds.detach().numpy(),
    grasp_outcomes_data, test_size=0.2, random_state=0)
#### 5.3.1 CNN (tactile only) + LR
model_531 = LogisticRegression(random_state=0, max_iter=1000)
model_531.fit(X_cnn_tactile_train, y_cnn_tactile_train)
model_531_predictions = model_531.predict(X_cnn_tactile_test)
#### 5.3.2 CNN (visual only) + LR
#### 5.3.3 CNN (both) + LR
model_533 = LogisticRegression(random_state=0, max_iter=1000)
model_533.fit(X_cnn_complete_train, y_cnn_complete_train)
model_533_predictions = model_533.predict(X_cnn_complete_test)
#### 5.3.4 CNN (tactile only) + PCA + LR
def pca(tensor, k, verbose=False):
    if len(tensor) == 3:
        tensor = tensor.reshape((tensor.shape[0], -1))

    pca = PCA(n_components=k)
    pca.fit(tensor)

    variance = sum(pca.explained_variance_ratio_) * 100
    # print(f"Variance captured for {k} components: {variance:.2f}%")

    if verbose:
        return pca.transform(tensor), k, variance
    else:
        return pca.transform(tensor)
# We select $k$ principal components that correspond to 90% variance of the
    dataset.
K = 0

k_variances = []
for i in range(1, 30):
    pca_cnn_tactile_ds, k, variance = pca(cnn_tactile.detach().numpy(), k=i,
        verbose=True)
    k_variances.append(variance)
    if variance > 90:
```

```
        print(f"Choose {k} components for CNN + tactile only dataset.")
        K = k
        break
for k in range(0, K, 5):
    if k > 0:
        print(f"Variance for {k} components: {k_variances[k]:.2f}%")
# Plot number of components against variance captured
plt.plot(k_variances)
plt.xlabel('Number of components ' + r'$k$')
plt.ylabel('Variance (%)')
plt.show()
pca_cnn_tactile_ds = pca(cnn_tactile.detach().numpy(), k=5)
X_pca_cnn_tactile_train, X_pca_cnn_tactile_test, y_pca_cnn_tactile_train,
    y_pca_cnn_tactile_test = train_test_split(pca_cnn_tactile_ds,
    grasp_outcomes_data, test_size=0.2, random_state=0)
model_534 = LogisticRegression(random_state=0, max_iter=1000)
model_534.fit(X_pca_cnn_tactile_train, y_pca_cnn_tactile_train)
model_534_predictions = model_534.predict(X_pca_cnn_tactile_test)
pca_cnn_complete_ds = pca(cnn_complete_ds.detach().numpy(), k=5)
X_pca_cnn_complete_train, X_pca_cnn_complete_test, y_pca_cnn_complete_train,
    y_pca_cnn_complete_test = train_test_split(pca_cnn_complete_ds,
    grasp_outcomes_data, test_size=0.2, random_state=0)
model_535 = LogisticRegression(random_state=0, max_iter=1000)
model_535.fit(X_pca_cnn_complete_train, y_pca_cnn_complete_train)
model_535_predictions = model_535.predict(X_pca_cnn_complete_test)
### 5.4 Results
def plot_confusion_matrices(model_data, fig_height):
    fig, axn = plt.subplots(1, len(model_data), sharex=True, sharey=True,
        figsize=(12, fig_height))

    for i, ax in enumerate(axn.flat):
        try:
            model_name = list(model_data.keys())[i]
            model = model_data[model_name]
            preds = model["preds"]
            score = model["score"]
            test_set = model["test_set"]

            if score is not None and preds is not None and test_set is not None:
                cm = confusion_matrix(test_set, preds)
                sns.heatmap(cm, linewidths=1, ax=ax, annot=True, fmt='g')
                ax.set_title(model_name + f": {score*100:.2f}%", fontsize=8)
            else:
                ax.set_title(model_name + "No results", fontsize=8)
            continue
```

```
        except IndexError:
            pass
model_data = {
    "Raw (tactile only)": {"score": model_511.score(X_tactile_test,
        y_tactile_test), "preds": model_511_predictions, "test_set":
        y_tactile_test},
    "Raw (visual only)": {"score": model_512.score(X_visual_test,
        y_visual_test), "preds": model_512_predictions, "test_set":
        y_visual_test},
    "Raw (both)": {"score": model_513.score(X_complete_test,
        y_complete_test),"preds": model_513_predictions,"test_set":
        y_complete_test},
}
plot_confusion_matrices(model_data, fig_height=3)
model_data = {
    "CNN (tactile only)": {"score": model_531.score(X_cnn_tactile_test,
        y_cnn_tactile_test), "preds": model_531_predictions,"test_set":
        y_cnn_tactile_test},
    "CNN (both)": {"score": model_533.score(X_cnn_complete_test,
        y_cnn_complete_test), "preds": model_533_predictions, "test_set":
        y_cnn_complete_test},
    "CNN (tactile only) + PCA": {"score":
        model_534.score(X_pca_cnn_tactile_test, y_pca_cnn_tactile_test),
        "preds": model_534_predictions, "test_set": y_pca_cnn_tactile_test},
    "CNN (both) + PCA": {"score": model_535.score(X_pca_cnn_complete_test,
        y_pca_cnn_complete_test), "preds": model_535_predictions, "test_set":
        y_pca_cnn_complete_test}
}
plot_confusion_matrices(model_data, fig_height=2)
```

C.2 Proposed Model Training Code

```
## UCL COMP0029 Individual Project for Year 3 BSc
### Robust Robotic Grasping Utilising Touch Sensing - Proposed Learning
    Framework Notebook
# This notebook contains the essential code for the proposed Multilayer
    Perceptron (MLP) approach to grasp stability prediction. The whole notebook
    should take approximately 10 minutes to run.
### 1. Load packages
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d.art3d import Poly3DCollection
```

```
import torch
import torch.nn as nn
import torch.nn.init as init
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms
from PIL import Image
import os
import gc

from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, precision_score, recall_score,
    f1_score, confusion_matrix, ConfusionMatrixDisplay
from sklearn.decomposition import PCA

import seaborn as sns
# Set device for `PyTorch` training
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
device
# PyTorch configs
gc.collect()
torch.cuda.empty_cache()
torch.set_num_threads(2)
np.errstate(invalid='ignore', divide='ignore')
### 2. Load datasets from saved .npy files
# To collect data for this experiment, you can run the "Collect Sensory Data"
    button in the Pybullet simulation. This generates a predefined number of
    Gaussian grasps randomly generated from a base hand pose. Each individual
    grasp is considered as an individual experiment, and the data collected from
    this experiment is split into four, each stored in its own dataset.

# For all object models used in this experiment, each object has 4 datasets
    which include:
# - `depth_ds.npy` which stores the depth tactile data from the mounted DIGIT
    sensors
# - `color_ds.npy` which stores the colored (RGB) version of the depth tactile
    data from the mounted DIGIT sensors
# - `poses_ds.npy` which stores the randomly-generated 6d hand poses from the
    simulation
# - `outcomes_ds.npy` which stores the outcomes of each random pose
# In this notebook, we will use the following dataset combinations:
# - `tactile` only (depth + colour)
# - `visual` with geometric features
# - `multi-modal` consisting of `tactile` and `visual`
```

```
# - ‘complete‘ consisting of ‘tactile‘ and ‘visual‘ with geometric features
root = "../datasets/"
object_names = ["block1", "block2", "block3", "cylinder1", "cylinder2",
    "cylinder3", "mustard_bottle1", "mustard_bottle2", "mustard_bottle3"]
# Load ‘.npy‘ datasets via ‘np.load()‘
depth_data = np.empty((0, 2, 160, 120))
color_data = np.empty((0, 2, 160, 120, 3))
poses_data = np.empty((0, 6))
grasp_outcomes_data = np.empty((0,))

for object_name in object_names:
    # Construct the relative paths of each dataset and load them into the
        notebook
    depth_data_temp = np.load(root + object_name + "_ds/depth_ds.npy",
        mmap_mode='r')
    colour_data_temp = np.load(root + object_name + "_ds/color_ds.npy",
        mmap_mode='r')
    poses_data_temp = np.load(root + object_name + "_ds/poses_ds.npy",
        mmap_mode='r')
    grasp_labels_temp = np.load(root + object_name + "_ds/grasp_outcomes.npy",
        mmap_mode='r')

    depth_data = np.append(depth_data, depth_data_temp, axis=0)
    color_data = np.append(color_data, colour_data_temp, axis=0)
    poses_data = np.append(poses_data, poses_data_temp, axis=0)
    grasp_outcomes_data = np.append(grasp_outcomes_data, grasp_labels_temp,
        axis=0)
del depth_data_temp, colour_data_temp, poses_data_temp, grasp_labels_temp
torch.cuda.empty_cache()
# These datasets should all be in the form of $(N\times...)$ where $N$ is the
    number of examples:
print(f"Shape of depth_data: {depth_data.shape}")
print(f"Shape of color_data: {color_data.shape}")
print(f"Shape of poses_data: {poses_data.shape}")
print(f"Shape of grasp_outcomes_data: {grasp_outcomes_data.shape}")
# Additionally, we confirm the number of successful and unsuccessful grasps
    recorded. This helps us in the next section to determine how many examples
    we should include for each class in order to produce a balanced dataset.
print(f"# of sucessesful grasps: {(grasp_outcomes_data == 1).sum()}")
print(f"# of unsuccessful grasps: {(grasp_outcomes_data == 0).sum()}")
### 3. Preprocessing
In this section, we create the required datasets specified from section 2, and
    normalise the datasets.
#### 3.1 Normalization
def normalize(arr):
```

```
    # Normalize & standardize each column
    mean = np.mean(arr, axis=0)
    std = np.std(arr, axis=0)

    arr = (arr - mean) / std
    arr = np.nan_to_num(arr, 0)
    arr[np.isinf(arr)] = 0
    return arr
# Depth and colour data consists of pairs of tactile sensory readings. Thus, we
    concatenate each pair together into single images
depth_data = np.concatenate((depth_data[:, 0], depth_data[:, 1]), axis=2)
color_data = np.concatenate((color_data[:, 0], color_data[:, 1]), axis=2)
depth_data = torch.from_numpy(normalize(depth_data))
color_data = torch.from_numpy(normalize(color_data))
visual_data = torch.from_numpy(np.nan_to_num(normalize(poses_data)))
print(f"Shape of depth_ds: {depth_data.shape}")
print(f"Shape of colour_ds: {color_data.shape}")
print(f"Shape of visual_ds: {visual_data.shape}")
# Create the tactile dataset
tactile_data = torch.cat([depth_data.unsqueeze(-1), color_data], dim=-1)
tactile_data = torch.nan_to_num(tactile_data)
print(f"Shape of tactile_ds: {tactile_data.shape}")
del depth_data, color_data, poses_data
torch.cuda.empty_cache()
#### Dimensionality reduction
# A simple convolutional neural network that extracts features from an input
    tensor
class FeatureExtractorCNN(nn.Module):
    def __init__(self):
        super(FeatureExtractorCNN, self).__init__()
        self.conv1 = nn.Conv2d(4, 16, kernel_size=3, stride=1, padding=1)
        self.conv2 = nn.Conv2d(16, 64, kernel_size=3, stride=1, padding=1)
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1)
        self.conv4 = nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1)

        self.pool = nn.MaxPool2d(kernel_size=3)


    def forward(self, x):
        x = self.conv1(x)
        x = nn.functional.relu(x)
        x = self.pool(x)

        x = self.conv2(x)
        x = nn.functional.relu(x)
```

```
        x = self.pool(x)

        x = self.conv3(x)
        x = nn.functional.relu(x)
        x = self.pool(x)

        x = self.conv4(x)
        x = nn.functional.relu(x)
        x = self.pool(x)

        return x
# Preprocess data using CNN feature extraction
cnn = FeatureExtractorCNN()
cnn_tactile = torch.cat([cnn(img.float().permute(2,0,1)).unsqueeze(0) for img in
    tactile_data])
cnn_tactile = cnn_tactile.reshape(cnn_tactile.shape[0], -1)
cnn_tactile.shape
del tactile_data
### 4. Testing different data representations
# In our MLP model, there are two fully connected (dense) layers, each with an
    activation function (ReLU for the first layer and no activation for the
    second layer). The input size, hidden size, and output size are parameters
    that need to be specified when creating an instance of the MLP.
class MLP(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, hidden_size)
        self.fc3 = nn.Linear(hidden_size, output_size)
        self.relu = nn.ReLU()

        self.criterion = nn.BCEWithLogitsLoss()
        self.optimizer = optim.Adam(self.parameters())

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        x = self.relu(x)
        x = self.fc3(x)
        return x

    def train_mlp(self, epochs, X_train, y_train):
        self.losses = []
        for i in range(epochs):
```

```
            inputs = torch.from_numpy(X_train).float()
            labels = torch.from_numpy(y_train).float().view(-1, 1)

            # Zero the gradients
            self.optimizer.zero_grad()

            # Forward pass
            preds = self(inputs)
            loss = self.criterion(preds, labels)
            self.losses.append(loss.item())

            # Backward pass and optimization
            loss.backward()
            self.optimizer.step()

            if i % 10 == 0:
                torch.cuda.empty_cache()

    def eval_mlp(self, X_test, y_test, verbose=False):
        # Evaluate the performance of the model on the testing set
        with torch.no_grad():
            inputs = torch.from_numpy(X_test).float()
            labels = torch.from_numpy(y_test).float().view(-1, 1)

            # Forward pass
            final_preds = self(inputs)
            predicted = (final_preds > 0).float()

            # Confusion matrix
            if verbose:
                cm = confusion_matrix(y_test, predicted)
                sns.heatmap(cm, linewidths=1, annot=True, fmt='g')

            # Performance metrics
            accuracy = round(accuracy_score(labels, predicted) * 100, 2)
            precision = round(precision_score(labels, predicted) * 100, 2)
            recall = round(recall_score(labels, predicted) * 100, 2)
            f1 = round(f1_score(labels, predicted) * 100, 2)
            return accuracy, precision, recall, f1

    def plot_losses(self):
        plt.plot(self.losses)
        plt.xlabel('Epochs')
        plt.ylabel('Training Loss')
        plt.show()
```

```
#### 4a. Testing datatset combinations (CNN dimensionality-reduced) without
    geometric features
def train_mlp_multitrial(dataset, grasp_outcomes_data, trials_count: int = 5,
    sampling=False):
    performance_metrics = np.empty((0, 4))
    dataset = dataset.detach().numpy() if isinstance(dataset, torch.Tensor) else
        dataset

    for i in range(trials_count):
        X_train, X_test, y_train, y_test = train_test_split(dataset,
            grasp_outcomes_data, test_size=0.2)
        mlp = MLP(input_size=X_train.shape[1], hidden_size=64, output_size=1)
        mlp.train_mlp(epochs=500, X_train=X_train, y_train=y_train)
        accuracy, precision, recall, f1 = mlp.eval_mlp(X_test=X_test,
            y_test=y_test)
        metrics_row = np.array([accuracy, precision, recall, f1]).reshape(1, 4)
        performance_metrics = np.append(performance_metrics, metrics_row, axis=0)

        torch.cuda.empty_cache()
    return performance_metrics
# Train Tactile-only dataset + CNN
metrics = train_mlp_multitrial(cnn_tactile, grasp_outcomes_data, trials_count=5)
print("Tactile-only MLP")
print(f"Accuracy:{metrics[:, 0]}")
print(f"Mean accuracy score: {np.mean(metrics[:, 0]):.2f} + {np.std(metrics[:,
    0]):.2f}")
print(f"Mean precision score: {np.mean(metrics[:, 1]):.2f} + {np.std(metrics[:,
    1]):.2f}")
print(f"Mean recall score: {np.mean(metrics[:, 2]):.2f} + {np.std(metrics[:,
    2]):.2f}")
print(f"Mean f1 score: {np.mean(metrics[:, 3]):.2f} + {np.std(metrics[:,
    3]):.2f}")
torch.cuda.empty_cache()
# Train Visual-only dataset + CNN
metrics = train_mlp_multitrial(visual_data, grasp_outcomes_data, trials_count=5)
print("Visual-only MLP")
print(f"Accuracy:{metrics[:, 0]}")
print(f"Mean accuracy score: {np.mean(metrics[:, 0]):.2f} + {np.std(metrics[:,
    0]):.2f}")
print(f"Mean precision score: {np.mean(metrics[:, 1]):.2f} + {np.std(metrics[:,
    1]):.2f}")
print(f"Mean recall score: {np.mean(metrics[:, 2]):.2f} + {np.std(metrics[:,
    2]):.2f}")
print(f"Mean f1 score: {np.mean(metrics[:, 3]):.2f} + {np.std(metrics[:,
    3]):.2f}")
```

```
torch.cuda.empty_cache()
# Train Multi-modal dataset (tactile + visual) + CNN
# We simply combine the cnn-processed tactile data (from Section 5.3.1) with the
    visual data
cnn_complete = torch.cat([cnn_tactile.reshape(cnn_tactile.shape[0], -1),
    visual_data], dim=1)
metrics = train_mlp_multitrial(cnn_complete, grasp_outcomes_data, trials_count=5)
print("Multi-modal MLP")
print(f"Accuracy:{metrics[:, 0]}")
print(f"Mean accuracy score: {np.mean(metrics[:, 0]):.2f} + {np.std(metrics[:,
    0]):.2f}")
print(f"Mean precision score: {np.mean(metrics[:, 1]):.2f} + {np.std(metrics[:,
    1]):.2f}")
print(f"Mean recall score: {np.mean(metrics[:, 2]):.2f} + {np.std(metrics[:,
    2]):.2f}")
print(f"Mean f1 score: {np.mean(metrics[:, 3]):.2f} + {np.std(metrics[:,
    3]):.2f}")
del cnn_complete
torch.cuda.empty_cache()
### 5. Determining impact of geometric features of objects on MLP accuracy
# This section aims to determine the capability of the selected object features
    (principal components) in identifying primitive object classes.
from sklearn.decomposition import PCA
from sklearn.cluster import KMeans, SpectralClustering
from sklearn.preprocessing import StandardScaler


np.set_printoptions(precision=2, formatter={'float_kind': "{:.3f}".format})
#### 5a. Clustering for object features using PCA
# Loading object features
### Object features
block_features = np.array([
    [0.025, 0.05, 0.05, 0., 0., 0., 0., 0., 0.],
    [0.03, 0.025, 0.045, 0., 0., 0., 0., 0., 0.],
    [0.05, 0.025, 0.04, 0., 0., 0., 0., 0., 0.],
])

cylinder_features = np.array([
    [0.04, 0.04, 0.05, 950.21606561, 14540.28434464, 950.21606561,
        14540.28434464, 950.21606561, 14540.28434464],
    [0.045, 0.045, 0.035, 750.78800246, 11488.6197291, 750.78800246,
        11488.6197291, 750.78800246, 11488.6197291],
    [0.034, 0.034, 0.045, 1315.17794549, 20124.96103064, 1315.17794549,
        20124.96103064, 1315.17794549, 20124.96103064]
])
```

```
bottle_features = np.array([
    [0.06, 0.04, 0.04, 43195.64459266, 114198.0441697 , 45229.93706864,
        135651.61794731, 75768.06626518, 83802.00991944],
    [0.04, 0.06, 0.06, 68993.34322089, 220902.86884084, 75354.47923164,
        239160.99530455, 109695.41304924, 147938.06047763],
    [0.04, 0.06, 0.04, 61744.75459905, 168925.8805044 , 68143.84372052,
        148775.07141178, 73102.67367022, 102953.40831915]
])

object_names = {0: 'cylinder', 1: 'block', 2: 'bottle'}
# Feature transformation - concatenating features together
object_features = (block_features, cylinder_features, bottle_features)
features = np.concatenate(object_features)
labels = np.array([0, 0, 0, 1, 1, 1, 2, 2, 2])
# Standardize the data
scaler = StandardScaler()
features = scaler.fit_transform(features)
# PCA
def pca(data, k, verbose=True):
    pca = PCA(n_components=k)
    pca.fit(data)

    if verbose:
        print(f"Variance captured: {sum(pca.explained_variance_ratio_)*100:.2f}%")
    return pca.transform(data)
# Determine the number of components to use
for i in range(1, 5):
    pca_res = pca(features, k=i)
### Plot 3-feature PCA results
pca3d = pca(features, k=3)

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

colors = ['r', 'g', 'b']
for i, c in zip(np.unique(labels), colors):
    ax.scatter(pca3d[labels == i, 0], pca3d[labels == i, 1], pca3d[labels == i,
        2], c=c, label=object_names[i])

ax.set_xlabel('Component 1')
ax.set_ylabel('Component 2')
ax.set_zlabel('Component 3')
ax.set_aspect('equal', 'box')
ax.legend()
```

```
plt.show()
#### 5b. MLP: Dataset combination 4: Visual (EE pose and geometric features) +
    CNN
blocks = np.repeat(np.concatenate(object_features), 160, axis=0)

visual_data = np.concatenate((visual_data, blocks), axis=1)
visual_data = torch.from_numpy(np.nan_to_num(normalize(visual_data)))
visual_data.shape
del object_features, blocks
# Train Visual-only (EE pose + geometric features) + CNN
metrics = train_mlp_multitrial(visual_data, grasp_outcomes_data, trials_count=5)
print("Visual data (EE pose + geo) MLP")
print(f"Accuracy:{metrics[:, 0]}")
print(f"Mean accuracy score: {np.mean(metrics[:, 0]):.2f} + {np.std(metrics[:,
    0]):.2f}")
print(f"Mean precision score: {np.mean(metrics[:, 1]):.2f} + {np.std(metrics[:,
    1]):.2f}")
print(f"Mean recall score: {np.mean(metrics[:, 2]):.2f} + {np.std(metrics[:,
    2]):.2f}")
print(f"Mean f1 score: {np.mean(metrics[:, 3]):.2f} + {np.std(metrics[:,
    3]):.2f}")
#### 5c. MLP: Dataset combination 5: Tactile + Visual (EE pose and geometric
    features) + CNN
torch.cuda.empty_cache()
final_dataset = torch.cat((cnn_tactile, visual_data), dim=1)
metrics = train_mlp_multitrial(final_dataset, grasp_outcomes_data,
    trials_count=5)
print(f"Final dataset (no PCA) MLP")
print(f"Accuracy:{metrics[:, 0]}")
print(f"Mean accuracy score: {np.mean(metrics[:, 0]):.2f} + {np.std(metrics[:,
    0]):.2f}")
print(f"Mean precision score: {np.mean(metrics[:, 1]):.2f} + {np.std(metrics[:,
    1]):.2f}")
print(f"Mean recall score: {np.mean(metrics[:, 2]):.2f} + {np.std(metrics[:,
    2]):.2f}")
print(f"Mean f1 score: {np.mean(metrics[:, 3]):.2f} + {np.std(metrics[:,
    3]):.2f}")
del cnn_tactile
#### 5d. MLP: Selecting number of components for PCA (with dataset combination 5)
# We conduct the following experiment to determine the optimal value of $k$ for
    PCA:
# 1. Train the MLP from $k=1$ to $k=20$
# 2. Repeat 1 for 5 times to find the average accuracy for each $k$, and choose
    $k$ value for the highest mean accuracy
MIN_K, MAX_K = 1, 30
```

```
pca_accuracies_mean = []
pca_accuracies_std = []
for i in range(MIN_K, MAX_K+1):
    transformed_dataset = pca(final_dataset.detach().numpy(), k=i, verbose=False)
    transformed_dataset = torch.from_numpy(transformed_dataset)
    metrics = train_mlp_multitrial(transformed_dataset, grasp_outcomes_data)
    pca_accuracies_mean.append(np.mean(metrics[:, 0]))
    pca_accuracies_std.append(np.std(metrics[:, 0]))
    torch.cuda.empty_cache()
    print(f"Completed PCA training for k={i} components")
del transformed_dataset
del visual_data
# Calculate the average accuracies for each $k$
pca_accuracies_mean = np.array(pca_accuracies_mean)
pca_accuracies_std = np.array(pca_accuracies_std)
K = pca_accuracies_mean.argmax(axis=0)
print(f"Number of components for PCA: {K}")
x = np.arange(MIN_K, MAX_K+1)
plt.errorbar(x, pca_accuracies_mean, yerr=pca_accuracies_std, fmt='o',
    markersize=5, ecolor='red', capsize=3, capthick=1)
plt.grid(True)
plt.xlabel("Number of principal components")
plt.xticks(np.arange(MIN_K, MAX_K+1, step=2))
plt.ylabel("Mean MLP accuracy (%)")
plt.title("Impact of Number of Principal Components on MLP Accuracy")
plt.show()
x = np.arange(MIN_K, MAX_K+1, step=1)
plt.plot(x, pca_accuracies_mean)
plt.fill_between(x, pca_accuracies_mean-pca_accuracies_std,
    pca_accuracies_mean+pca_accuracies_std, alpha=0.2)
plt.grid(True)
plt.xlabel("Number of principal components")
plt.xticks(np.arange(MIN_K, MAX_K+1, step=2))
plt.ylabel("Mean MLP accuracy (%)")
plt.title("Impact of Number of Principal Components on MLP Accuracy")
plt.show()
top_accuracy_indices = np.argsort(pca_accuracies_mean)[-5:]
top_accuracies = pca_accuracies_mean[top_accuracy_indices]
top_stds = pca_accuracies_std[top_accuracy_indices]

print("Top 5 mean accuracies:", top_accuracies)
print("Top 5 std accuracies:", top_stds)
print("Corresponding indices:", top_accuracy_indices + 1)
### 6. Experiments
```

```
# We update the train_mlp_multitrial function to accomodate pre-defined training
    and testing datasets
def train_mlp_multitrial2(X_train, X_test, y_train, y_test, trials_count: int =
    5):
    performance_metrics = np.empty((0, 4))

    for i in range(trials_count):
        mlp = MLP(input_size=X_train.shape[1], hidden_size=64, output_size=1)
        mlp.train_mlp(epochs=500, X_train=X_train, y_train=y_train)
        accuracy, precision, recall, f1 = mlp.eval_mlp(X_test=X_test,
            y_test=y_test)
        metrics_row = np.array([accuracy, precision, recall, f1]).reshape(1, 4)
        performance_metrics = np.append(performance_metrics, metrics_row, axis=0)

        torch.cuda.empty_cache()

    return performance_metrics
#### 6.1 Train MLP on 2 blocks, test on remaining block
# Objective: see MLP's robustness
combinations = [
    [[1, 2], [3]],
    [[1, 3], [2]],
    [[2, 3], [1]]
]

exp1_accuracies_mean = []
exp1_accuracies_std = []
final_dataset = final_dataset.detach().numpy()

for i in range(3): # For each primitive object type (box, cylinder, bottle)
    stidx = 480 * i # Starting index to slice final dataset
    for combination in combinations:
        training, testing = combination[0], combination[1]
        X_train1 = final_dataset[(stidx + (training[0] - 1) * 160 + 1):(stidx +
            (training[0] * 160))]
        X_train2 = final_dataset[(stidx + (training[1] - 1) * 160 + 1):(stidx +
            (training[1] * 160))]
        X_train = np.vstack((X_train1, X_train2))

        y_train1 = grasp_outcomes_data[(stidx + (training[0] - 1) * 160 +
            1):(stidx + (training[0] * 160))]
        y_train2 = grasp_outcomes_data[(stidx + (training[1] - 1) * 160 +
            1):(stidx + (training[1] * 160))]
        y_train = np.concatenate((y_train1, y_train2))
```

```
        X_test = final_dataset[(stidx + (testing[0] - 1) * 160 + 1):(stidx +
            (testing[0] * 160))]
        y_test = grasp_outcomes_data[stidx + (testing[0] - 1) * 160 + 1:stidx +
            (testing[0] * 160)]
        metrics = train_mlp_multitrial2(X_train, X_test, y_train, y_test)
        exp1_accuracies_mean.append(np.mean(metrics[:, 0]))
        exp1_accuracies_std.append(np.std(metrics[:, 0]))
        torch.cuda.empty_cache()
del X_train, X_test, y_train, y_test, training, testing, combination,
    combinations
exp1_accuracies_mean = np.array(exp1_accuracies_mean)
exp1_accuracies_std = np.array(exp1_accuracies_std)
exp1_accuracies_mean
exp1_accuracies_std
#### 6.2 Train MLP on randomly sampled subset of all objects
# Objective: see influence of dataset size on accuracy;
# Randomly sample a dataset of the specified size without shuffling the dataset.
def consistent_sampling(dataset1, dataset2, input_size):
    num_rows = len(dataset1)
    indices = np.random.choice(len(dataset1), input_size, replace=False)
    samples1, samples2 = [], []
    for i in indices:
        samples1.append(dataset1[i])
        samples2.append(dataset2[i])

    samples1 = np.stack(samples1, axis=0)
    samples2 = np.stack(samples2, axis=0)
    return samples1, samples2
exp2_accuracies_mean = []
exp2_accuracies_std = []

for input_size in range(100, 1500, 100):
    dataset, grasp_labels = consistent_sampling(final_dataset,
        grasp_outcomes_data, input_size=input_size)
    metrics = train_mlp_multitrial(dataset, grasp_labels)
    exp2_accuracies_mean.append(np.mean(metrics[:, 0]))
    exp2_accuracies_std.append(np.std(metrics[:, 0]))
exp2_accuracies_mean = np.array(exp2_accuracies_mean)
exp2_accuracies_std = np.array(exp2_accuracies_std)
x = np.arange(100, 1500, step=100)
plt.plot(x, exp2_accuracies_mean)
plt.fill_between(x, exp2_accuracies_mean-exp2_accuracies_std,
    exp2_accuracies_mean+exp2_accuracies_std, alpha=0.2)
plt.grid(True)
```

```
plt.xlabel("Sample size")
plt.xticks(np.arange(100, 1500, step=100), rotation=45)
plt.ylabel("Mean MLP Accuracy (%)")
plt.title("Impact of Sample Size on MLP Performance")
plt.show()
```