OOP Grasp Classifier Coursework Report

Sebastian Ren : 23036338
Yan Pan : 23158397

December 2024

1 Introduction

This project aims to develop a generic gripper grasping simulation system using PyBullet. The system is designed to simulate various robotic grippers performing grasping operations on a diverse range of objects within a 3D simulation environment. The primary objectives are to evaluate grasp quality under varying scenarios and to collect labeled data for subsequent analysis and machine learning applications.

To achieve this, we need to accomplish the following objectives:
1. To develop modular gripper models that can handle different gripper types, such as two-finger ('PR2') and three-finger ('F3') configurations.
2. To implement object-agnostic grasping mechanisms, enabling the system to support various object types by simply updating object definitions.
3. To provide a configurable simulation interface where users can specify gripper types, grasping directions, target objects, and modes of operation.
4. To collect data systematically, including gripper positions, orientations, and grasp success labels, for further evaluation or research.

We designed the system to achieve its objectives and realize its flexibility and scalability through a structured design:
1. An abstract Gripper class defines a common API for operations such as loading objects, moving grippers, and collecting data. Subclasses inherit from Gripper and implement specific mechanics tailored to different gripper types.
2. The function dynamically loads objects by name and supports generating random positions around the object for grasping, enabling the inclusion of new objects with minimal modifications.
3. The simulation includes a visualized graphical user interface (GUI), where simulation parameters such as gripper type, object name, grasping direction, and the amount of data to be collected during the simulation can be configured through a unified 'RUN' interface.
4. The system is modular, enabling easy adaptation to new environments, gripper types, or object configurations. To integrate a new gripper, it only requires implementing the abstract methods defined in the abstract class.

With this design, the system offers a generic and reusable framework for robotic grasping tasks, ensuring it can support a wide range of research and development needs.

Below is a simplified flowchart of the system, which provides a clear and concise visual representation of its workflow, including input parameters, operational steps, and data outputs:

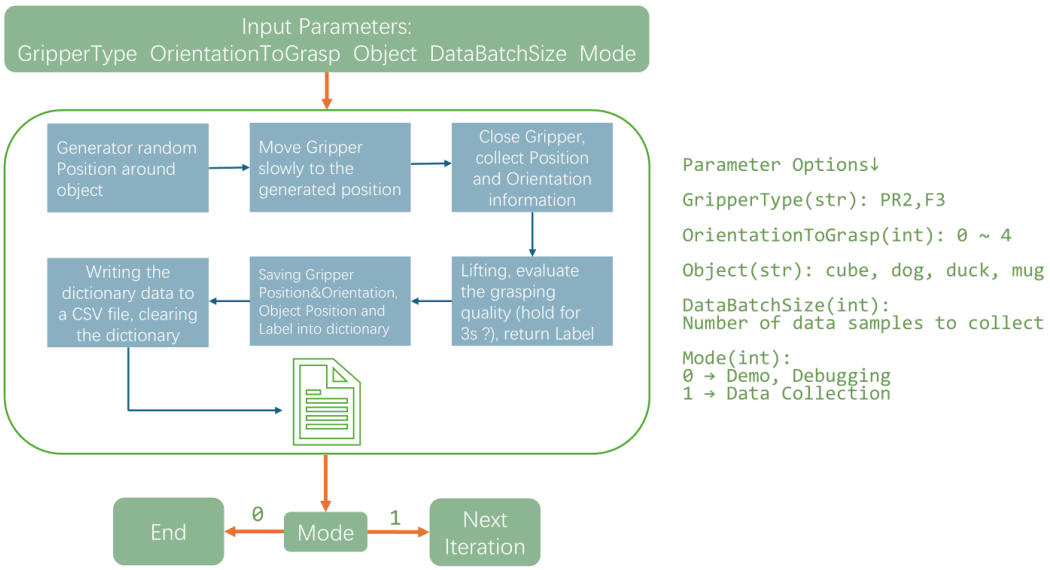


Figure 1: Simplified System Flowchart

2 Method: System Architecture and Class Hierarchy

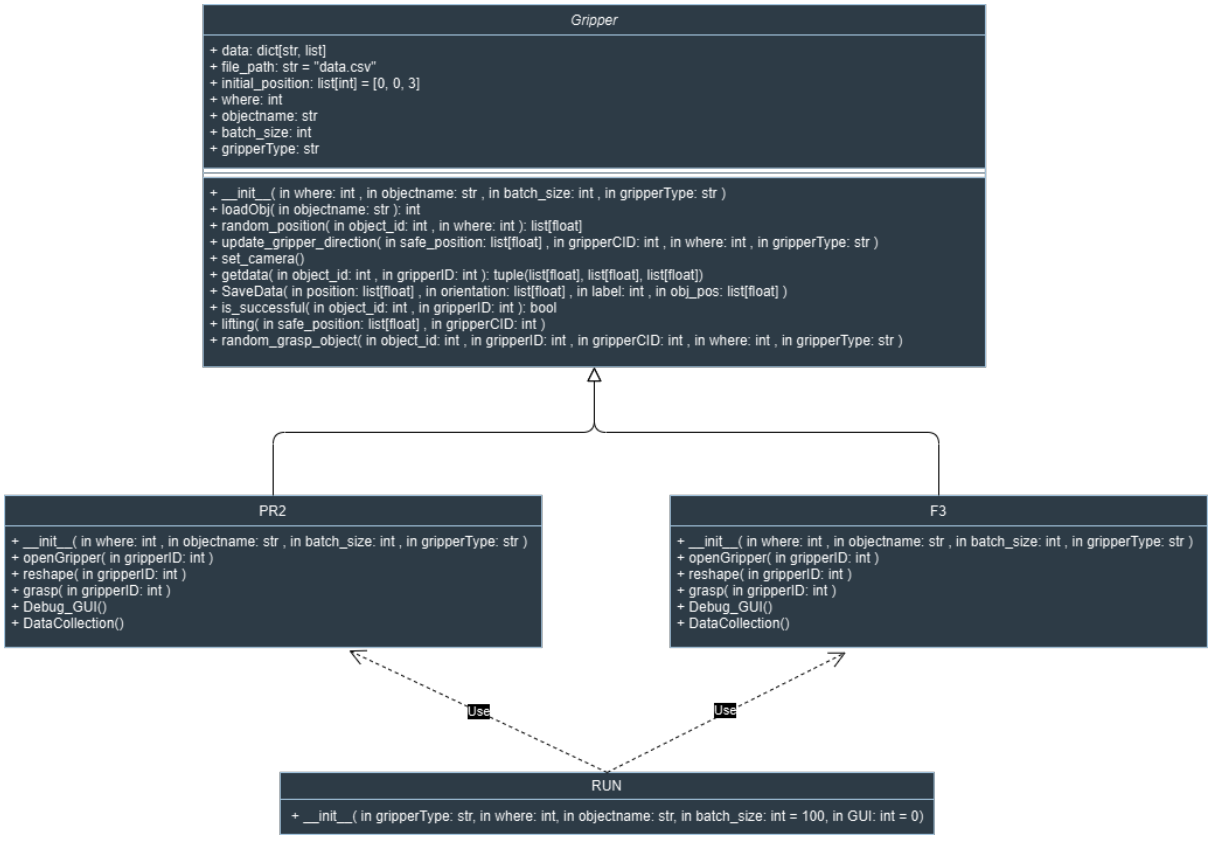


Figure 2: Unified Modeling Language

2.1 The Abstract 'Gripper' Class

Gripper serves as the abstract base class, encapsulating universal functionalities and specifying an interface for concrete implementations.

Upon initialization, it stores key parameters:
where (int): The approach direction for grasping. 0 represents grasping from above, while 1 to 4 represent different lateral or angled approaches.
objectname (str): The name of the object to be grasped, e.g., "cube", "dog", "duck", "mug".
batch_size (int): The number of repeated grasp attempts to be performed in data-collection mode.
gripperType (str): The type of gripper in use, e.g. "PR2" or "F3".

The Gripper class includes general-purpose methods:
loadObj(objectname): Load a specified URDF object into the scene.
loadTable(): Load a table URDF as the environment base.
random_position(object_id, where): Generate a random grasp point around the object, adding Gaussian noise to increase variability. Return 'safe_position'
update_gripper_direction(safe_position, gripperCID, where, gripperType): Move and orient the gripper towards the target grasp point by using method 'p.changeConstraint(gripperCID)'.
set_camera(): Adjust the camera for GUI.
getdata(), SaveData(): Record and store data into a CSV file.
is_successful() and lifting(): Evaluate grasp success and test object stability by lifting it.
random_grasp_object(object_id, gripperID, gripperCID, where, gripperType): Integrate all steps to perform a single grasp attempt and record the data.

Abstract methods, which must be implemented by subclasses, include:
openGripper(gripperID): Commands the gripper joints to open.
reshape(gripperID): Pre-shape the gripper fingers for an optimal approach before closing.
grasp(gripperID): Close the gripper to attempt to hold the object.
Debug_GUI(): A mode to run a single grasp process visually for debugging.
DataCollection(): A looped mode to collect multiple data samples and store them for offline analysis.

2.2 The Subclasses: 'PR2' and 'F3'

The PR2 and F3 classes inherit from Gripper and provide concrete implementations of the abstract methods. For instance, PR2 controls a simpler two-finger pair structure, while F3 manages a more complex three-finger design with multiple joints per finger. The specifics of joint angles, target positions, and forces are implemented in these subclasses.This design allows the entire system to run on different objects and grippers without any structural changes.

2.3 The 'RUN' Class

RUN acts as the user-facing entry point. Users specify the **gripperType**, **where**, **objectname**, **batch_size**, and GUI **mode**. Based on these parameters, RUN instantiates the corresponding gripper class (e.g., PR2 or F3) and calls either Debug_GUI() (if mode=0) for a single demonstration or DataCollection() (if mode=1) for repeated automated data collection.

3 Processes and Computational Details

The workflow can be divided into several logical stages, each involving specific computations and parameter selections:

1) Environment and Object Setup:
The system begins by resetting the simulation environment and loading a stable surface (a table model) into the scene. Next, it imports the chosen object's URDF file at a predefined initial position and scaling factor, ensuring the object rests stably on the table. The selection of the object (e.g., cube, dog, duck, mug) influences the model dimensions and placement but does not alter the core logic.

2) Camera and Visualization:
A fixed camera viewpoint can be set, ensuring that during debugging, the user observes the scene from a consistent angle. Although not essential for automated data collection, a stable camera view aids in developing and verifying the grasping strategies visually.

3) Random Grasp Position Generation:
The system queries the object's axis-aligned bounding box to determine its spatial extent. Using this bounding box, it computes a nominal target grasp point, typically near the object's center or relevant surface, depending on the approach direction. The approach direction parameter(**where**) determines whether the gripper is positioned above or to the side of the object. To introduce diversity in the data, Gaussian noise with adjustable means and standard deviations is added to these nominal coordinates. This stochastic element ensures that the dataset reflects a wide range of slight positional variations and approach angles.

4) Gripper Movement and Orientation:
Once a target grasp point is established, the system linearly interpolates the gripper's position from a high, safe start point('safe' here means does not collide with object) down to the target position over multiple discrete steps, it updates the gripper's position and orientation constraints, gradually bringing it into the desired pose. The orientation depends on the approach direction; for example, a side approach demands a different rotation.
The conversion from intuitive Euler angles to quaternions (the format required by PyBullet) occurs internally. By carefully selecting the Euler angles for each approach direction and gripper type, the system ensures the gripper points the fingers optimally toward the target area of the object.

5) Opening, Pre-shaping, and Grasping Actions:
With the gripper at the correct pose, it first fully opens to avoid premature collisions. Subsequently, it reshapes or reconfigures its fingers into a pre-grasp posture tailored to the upcoming closing action. Once pre-shaped, it commands the fingers to close around the object, applying appropriate joint targets, velocities, and force limits.

6) Lifting and Grasp Quality Evaluation:
After closing the fingers, the system lifts the gripper and the presumably grasped object upward in small increments. If the object remains attached to the gripper for at least 3 seconds, it suggests a stable grasp(Label:1). This evaluation uses contact point queries between the gripper and the object. A sufficient number of contact points indicates the object is firmly held. If very few or no contact points remain after some stabilization time, the grasp is deemed unsuccessful(Label:0).

7) Data Generation and Storage:
The system records the gripper(Base)'s final position and orientation, the object's position after the attempt, and the label. These data are appended as a new entry in a CSV file. If the file does not exist, it is created with appropriate headers. In repetition-based runs, each grasp attempt contributes one row to the dataset, systematically building a diverse and labeled collection of grasp trials.

4 Data Visualization

We record the gripper's (Base) final position and orientation, the object's final position, and a binary success label. These features emphasize the spatial relationship between the gripper and the object,

making them directly relevant to grasp stability. This representation is concise, generalizable, and provides sufficient context for a classifier to learn which poses are likely to result in successful grasps, without relying on object- or gripper-specific details.
To better understand the data, we separately visualized the gripper's position, orientation, and grasp success positions using scatter plots and thermal density maps. The visualizations provide intuitive insights into the spatial distribution of successful and unsuccessful grasps.
The experiments were conducted with the PR2 and F3 grippers, which were used to perform grasping tasks on the objects "cube" and "duck," respectively.

Three Finger Gripper(F3)

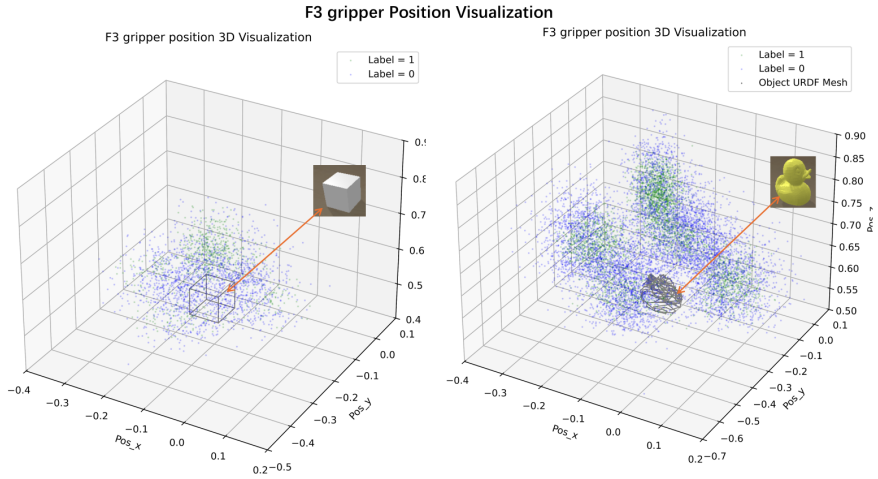


Figure 3: F3 gripper position visualization, green(1) blue(0)

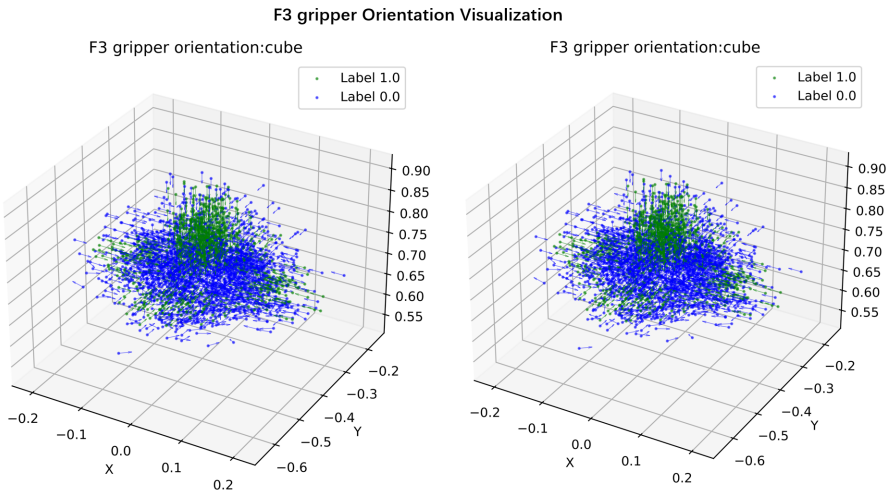


Figure 4: F3 gripper orientation visualization by vector

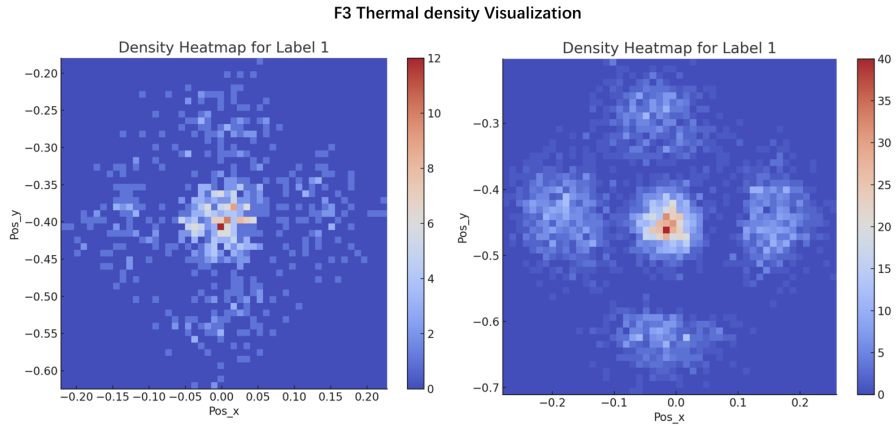


Figure 5: F3 gripper success grasp positions visualization

Two Finger Gripper(PR2)

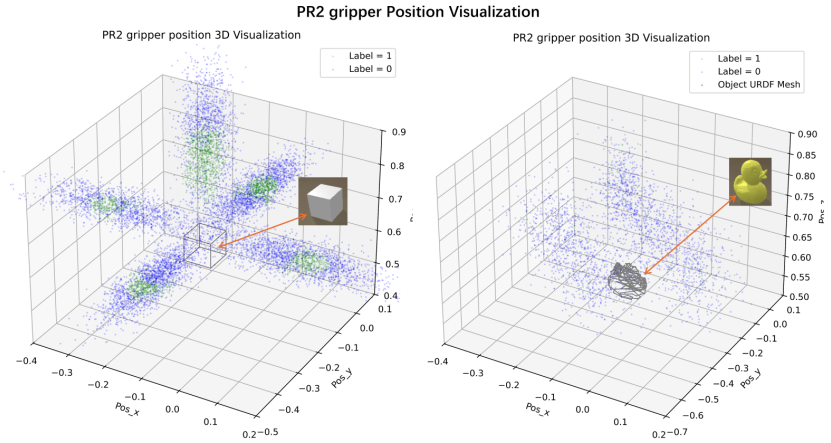


Figure 6: PR2 gripper position visualization, green(1) blue(0)

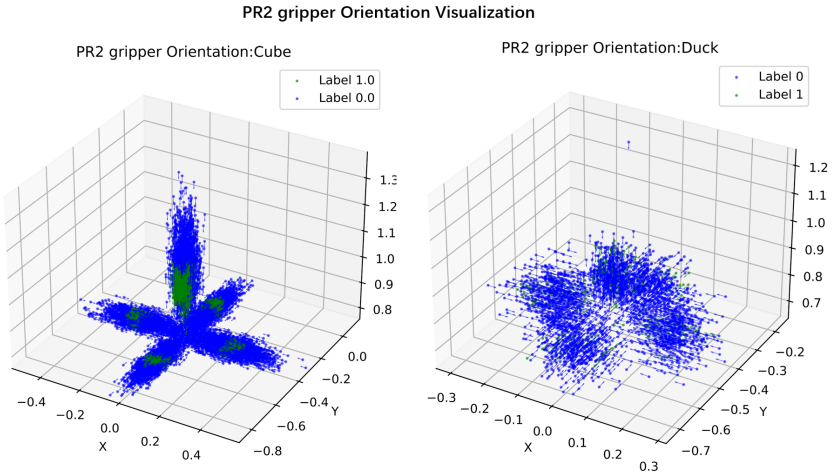


Figure 7: PR2 gripper orientation visualization by vector

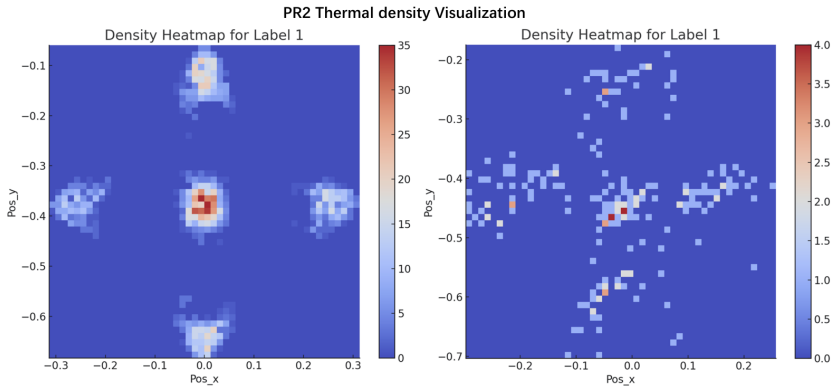


Figure 8: PR2 gripper success grasp positions visualization

We can see that the F3 gripper's strengths lie in its versatility and adaptability to diverse objects, with broader positional and orientational success ranges. On the other hand, the PR2 gripper excels in tasks requiring high precision and is particularly effective for symmetric objects. However, the F3 gripper's complexity requires more computational resources for planning, while the PR2 gripper's limited adaptability makes it less effective for generalized grasping tasks. These differences underscore the trade-offs between complexity and precision in gripper design, with the F3 being more suitable for general-purpose tasks and the PR2 excelling in specific, controlled scenarios.

5 Classifier Model: Setup,Training and Analysis

In this section, we compared the performance of two classifiers, Neural Network (NN) and XGBoost, in classifying grasp outcomes for two objects (Cube and Duck) using two grippers (F3 and PR2). Before training, the data was normalized to ensure all features were scaled to the same range, minimizing biases from differing feature scales and improving model convergence. Additionally, we applied Random Search to identify the best hyperparameters for both classifiers, ensuring their optimal performance.

5.1 Model Performance Analysis

NN vs. XGBoost: Performance on F3 Gripper for Cube and Duck

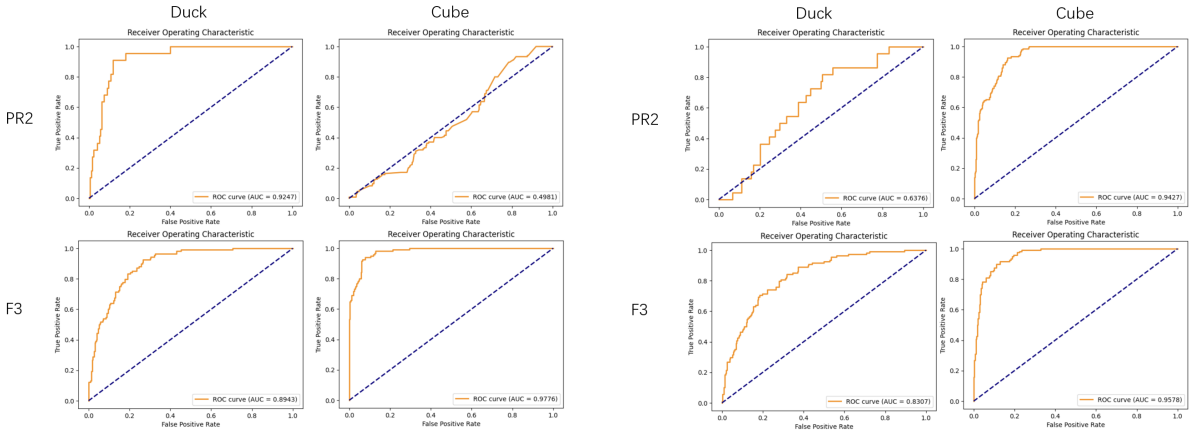
Cube	Train Accuracy	Test Accuracy	Test Precision	Test Recall	F1-Score
NN	0.8381	0.8825	0.9011	0.6833	0.7773
XGBoost	0.9262	0.9250	0.8689	0.8833	0.8760
Duck	**Train Accuracy**	**Test Accuracy**	**Test Precision**	**Test Recall**	**F1-Score**
NN	0.7746	0.7775	0.5798	0.6389	0.6079
XGBoost	0.8180	0.8125	0.6410	0.6944	0.6667

NN vs. XGBoost: Performance on PR2 Gripper for Cube and Duck

Cube	Train Accuracy	Test Accuracy	Test Precision	Test Recall	F1-Score
NN	0.9028	0.8524	0.6055	0.9778	0.7479
XGBoost	0.9322	0.8839	0.6738	0.9333	0.7826
Duck	**Train Accuracy**	**Test Accuracy**	**Test Precision**	**Test Recall**	**F1-Score**
NN	0.9298	0.8900	0.0000	0.0000	0.0000
XGBoost	0.9447	0.8850	0.4000	0.0909	0.1481

Figure 9: Confusion Matrix Metrics

For the F3 Gripper, XGBoost consistently outperformed NN in both Cube and Duck classification tasks. In the Cube classification, NN achieved a test accuracy of 0.8825 and an F1-Score of 0.7773, showing high precision (0.9011) but lower recall (0.6833), indicating missed positive samples. XGBoost performed better with a test accuracy of 0.9250 and an F1-Score of 0.8760, demonstrating a good balance between precision (0.8689) and recall (0.8833). For Duck classification, NN struggled with precision (0.5798) and recall (0.6389), resulting in an F1-Score of 0.6079. In contrast, XGBoost achieved higher precision (0.6410) and recall (0.6944), with an F1-Score of 0.6667, making it the more reliable classifier.
In the PR2 Gripper Cube classification, NN achieved a test accuracy of 0.8524 and an F1-Score of 0.7479, with a very high recall (0.9778) but low precision (0.6055), suggesting a high false positive rate. XGBoost provided a better balance with a test accuracy of 0.8839 and an F1-Score of 0.7826, improving precision to 0.6738 while maintaining a strong recall (0.9333). For Duck classification, both classifiers struggled due to severe data imbalance. NN completely failed, with a precision, recall, and F1-Score of 0.0000. XGBoost performed slightly better, achieving a precision of 0.4000 and a recall of 0.0909, resulting in an F1-Score of 0.1481, though still inadequate for reliable classification.
The observed performance issues for the PR2 Duck dataset are largely attributed to the severe class imbalance, which makes it challenging for classifiers to learn and predict minority class samples effectively.The F3 Gripper datasets for both Cube and Duck are relatively balanced, making the classification results reliable. Similarly, the PR2 Cube dataset is perfectly balanced, ensuring the reported performance reflects actual classifier capabilities. However, the PR2 Duck dataset is severely imbalanced, with only 8.23% of samples belonging to the positive class. This imbalance led to poor performance from both classifiers, particularly NN, which failed to identify any positive samples. The analysis of XGBoost and



(a) XGBoost (b) NN

Figure 10: ROC Curves

Neural Network (NN) models based on ROC curves and AUC values reveals distinct strengths for each model. For the PR2 gripper's Duck classification, XGBoost excelled with an AUC of 0.9247, handling the imbalanced dataset far better than NN, which achieved a lower AUC of 0.6376. Conversely, for PR2 Cube classification, NN significantly outperformed XGBoost, achieving an AUC of 0.9427 compared to XGBoost's poor performance (AUC 0.4981), highlighting NN's suitability for this task.
On the F3 gripper, both models performed well, with XGBoost showing a slight edge in both Cube and Duck classification (AUC 0.9776 and 0.8943, respectively). NN also achieved high AUC values (0.9578 for Cube and 0.8307 for Duck), demonstrating its competitiveness on balanced datasets.
Overall, XGBoost is the preferred model for imbalanced datasets and most tasks, while NN shines in specific cases like PR2 Cube classification. To improve performance, techniques like oversampling for imbalanced data and feature engineering for challenging datasets could enhance both models. An ensemble approach combining XGBoost and NN may further optimize results.

5.2 Model Performance Analysis: K-Cross Validation

In addition to Random Search, we also used K-fold Cross Validation to further explore model performance improvements. Below are the metrics and ROC curves for the two grippers applied to the two objects after implementing K-fold Cross Validation.

NN vs. XGBoost: Performance on PR2 Gripper for Cube and Duck

Cube	Test Accuracy	Test Precision	Test Recall	F1-Score
NN	0.8474	0.6126	0.8667	0.7178
XGBoost	0.8939	0.6940	0.9407	0.7987

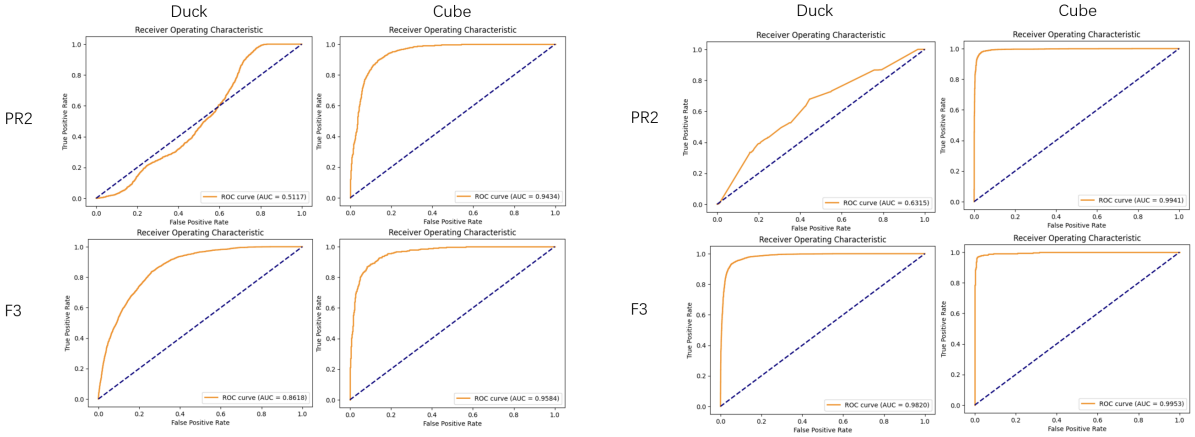
Duck	Test Accuracy	Test Precision	Test Recall	F1-Score
NN	0.8900	0.0000	0.0000	0.0000
XGBoost	0.8800	0.3333	0.9090	0.1429

NN vs. XGBoost: Performance on F3 Gripper for Cube and Duck

Cube	Test Accuracy	Test Precision	Test Recall	F1-Score
NN	0.9100	0.8559	0.8417	0.8487
XGBoost	0.9300	0.8710	0.9000	0.8852

Duck	Test Accuracy	Test Precision	Test Recall	F1-Score
NN	0.7700	0.5741	0.5741	0.5741
XGBoost	0.8100	0.6311	0.7130	0.6696

Figure 11: PR2 gripper success grasp positions visualization



(a) NN

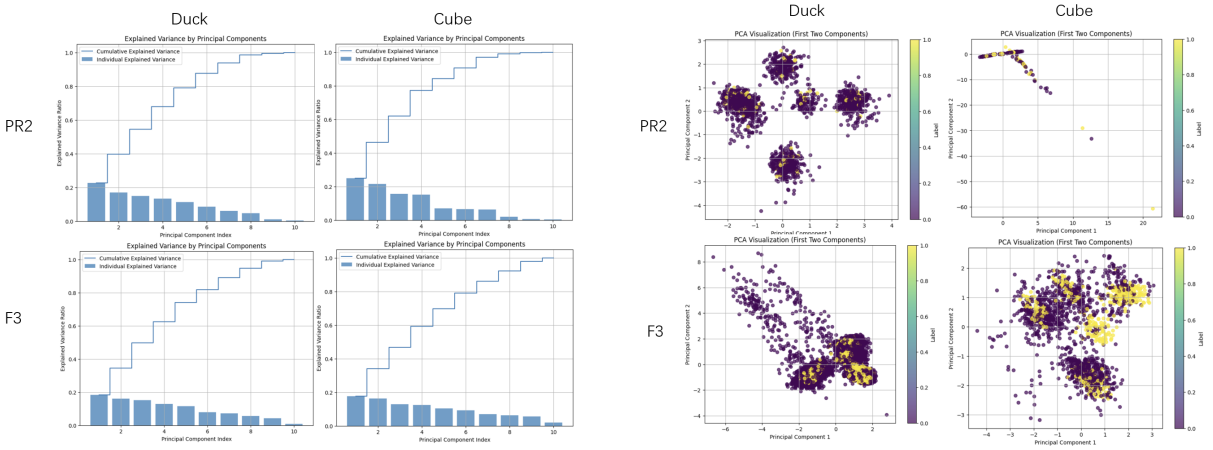
(b) XGBoost

Figure 12: ROC Curves

In datasets where class 0 is overly represented and class 1 is severely underrepresented, models tend to predict 0 as the dominant class. This bias occurs because predicting the majority class allows the model to minimize training loss and reduce the overall error rate. However, this strategy significantly impacts the performance on the minority class (class 1), particularly in terms of key metrics like Precision, Recall, and F1-Score. While AUC may improve, it often fails to reflect the model's struggles in effectively capturing minority class patterns under default threshold settings.

For instance, in the PR2 Duck classification task, under Random Search, the neural network (NN) completely ignored class 1, achieving Precision and Recall of 0. With K-Fold Cross-Validation, metrics like Precision and F1-Score showed little to no improvement, indicating that the default threshold (0.5) still favored the majority class. This highlights that, despite improvements in overall discrimination (as measured by AUC), the models still struggled to address the imbalance in the dataset effectively.

5.3 Dimension Reduction: PCA on XGBoost Model



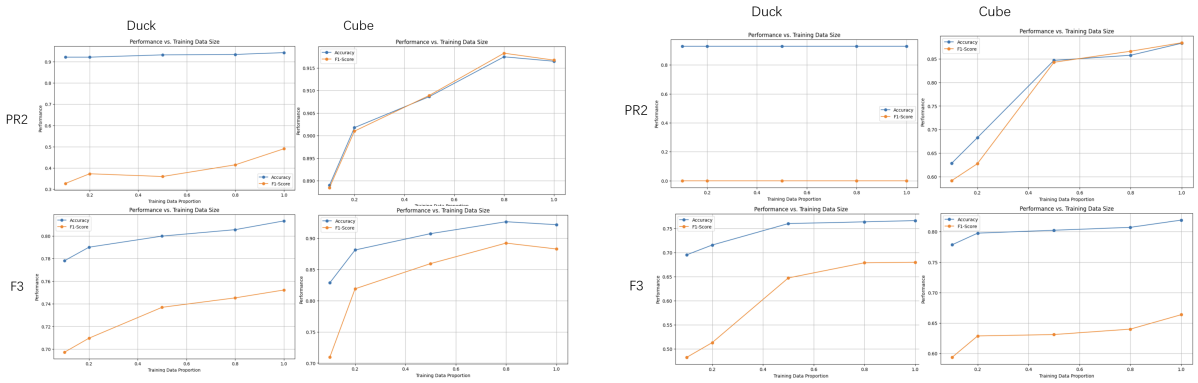
(a) cumulative and individual Variance

(b) Feature visualization

Figure 13: PCA: Components Variance and Feature Visualization

By using the code to identify the top features that preserve the most variance, we determined that the three features with the highest PCA weights are Pos_x, Pos_y, and Pos_z. These features contributed significantly to the first few principal components, which captured the majority of the variance in the data. Specifically, the cumulative explained variance plot demonstrated that the first three components already accounted for a substantial portion of the total variance (e.g., over 60% for PR2 Gripper - Cube). This finding aligns with the nature of the dataset, where positional attributes are likely to dominate the variance due to their strong correlation with object placement and grasp outcomes.

5.4 Model performance for different data size



(a) XGBoost Model

(b) NN Model

Figure 14: Accuracy and F1-Score for different data size

Overall, the trend across all datasets shows that as the training data size increases, classifier performance, measured by accuracy and F1-score, tends to improve. This behavior is expected, as more data generally enables models to learn better representations and generalize effectively. However, the rate of improvement and the final plateau vary across different scenarios.

The most notable trend is that performance improves significantly at lower data proportions (e.g., from 10% to 50%), after which the rate of improvement slows down, indicating diminishing returns with additional data. This suggests that for some tasks, a relatively small portion of the data may already capture most of the useful patterns for classification.

Anomalies arise in cases where F1-scores lag behind accuracy, especially in imbalanced datasets. This indicates that while accuracy appears high due to correct predictions of the majority class, the model struggles with the minority class, leading to low F1-scores. This disparity between metrics highlights

the impact of class imbalance, as the model becomes biased towards predicting the majority class to minimize overall errors.
In summary, the trend confirms that increasing data size is generally beneficial, but issues like class imbalance can distort improvements in certain performance metrics. This underscores the need to pair increased data with techniques to address dataset imbalances, ensuring meaningful gains across all evaluation metrics.

6 Discussion and conclusion

Since the quality labels for our successful grasps are determined by the time the object can be held by the gripper, we didn't have enough time to collect equivalent datasets for each gripper across all objects. For some objects, we collected as many as 20,000 samples (e.g., F3_Duck), while for others, we only collected around 2,000 samples. This led to a situation where, for some simple and less functional grippers, most of the data collected when attempting to grasp irregularly shaped and more challenging objects consisted of 0s. As a result, the model's training outcomes on these datasets are suboptimal, lacking much value or significance, and the visualizations of these data appear highly anomalous.
Additionally, during data collection on different computers, one of the setups inadvertently used an incorrect noise generation scale and range, leading to a specific 3D distribution that stands out from the others. Despite these challenges, we believe that our code demonstrates strong generalization capability. If new grippers need to be introduced, we can achieve this by defining a few necessary abstract class methods to enable the collection of data for various objects.
It is also worth noting that our gripper is always initialized around the object because we use the 'getAABB()' function. The gripper doesn't suddenly appear around the object, which could result in the object being knocked away. Instead, it gradually descends and moves to a randomly generated position to attempt to grasp the object. Lastly, we did not introduce variance in orientation because this would significantly increase the cost of data collection, leading to even more 0s in the dataset.
Finally, even though we didn't have ample time and resources to perfect every aspect of this project, we firmly believe that we have accomplished everything essential—and perhaps even beyond—within the constraints we faced.