RobotDockCenter: A Supervised Learning Approach to Robot Docking using Monocular Vision

Edward Ferrari

June 4, 2025

Abstract

Autonomous robot docking enables robots to recharge or continue their tasks without human intervention. Traditional docking solutions rely on multiple sensors and reinforcement learning to achieve precise alignment. In this paper, we propose an alternative approach using monocular vision and basic deep learning techniques to predict docking actions from a single input frame. Our method eliminates the need for additional sensors, policy optimization, or reward functions. We conduct experiments in the Godot game engine to simulate real-world deployment. Our results demonstrate that a supervised learning approach can effectively guide docking maneuvers while maintaining simplicity and practicality.

1 Introduction

Robot docking is a fundamental task in autonomous systems, enabling robots to recharge and continue operations without human intervention. Traditional docking solutions rely on multiple sensors such as LIDAR, ultrasonic sensors, and cameras [Jia et al., 2023]. While effective, these approaches increase hardware complexity and cost.

Reinforcement learning has gained popularity in robotic control, but it often requires extensive training, computational resources, reward function optimization, and at times compromises when implementing them in low-cost systems [Deisenroth et al., 2012]. These challenges make reinforcement learning impractical for resource-constrained robots that need a way to dock and charge.

To address these limitations, we propose an alternative approach that utilizes monocular vision and supervised learning to achieve reliable docking without additional sensors or reinforcement learning. Our method predicts docking actions from a single input frame, identifying two key targets: one for alignment and the other representing the docking station.

We use the Godot game engine to generate synthetic training data, capturing diverse docking scenarios. A deep learning model, implemented in PyTorch, is trained on these images to make high-frequency predictions. The model is lightweight, allowing real-time inference on low-power hardware, such as a Raspberry Pi [Ameen et al., 2023]. A Flask server facilitates communication between the trained model and the robot's virtual control system for integration into a simulated docking environment.

Our results suggest that monocular vision-based docking is a feasible alternative to sensor-heavy and reinforcement learning-based approaches. This method offers a cost-effective and computationally efficient solution for autonomous docking, with potential for real-world deployment.

The paper is structured as follows. After going over related work, we go in depth regarding our methodology, which includes data generation, data preprocessing, model architecture, training, inference, and Flask server integration. We then present our results and discuss the possible implications of our work. We then conclude our paper with a discussion on the limitations of our work and potential future directions.

2 Related Work

The work of Jia et al. [2023] provides a comprehensive review of existing docking methods. **Keep this section for the end once we have collected all of our sources**

3 Methodology

In this section, we display how effective image preprocessing and our docking procedure collectively lead to a plausible method to train our model without manual labeling, which would be required, for our case, in real-world applications. Our methodology includes data collection, data preprocessing, model

training, inference, server integration, evaluation, and experimental setup.

3.1 Data Collection

We collect training data using a simulated environment created in a game engine [Hoster et al., 2024]. Our game engine of choice is Godot, a free and open source game engine. The environment consists of a robot, a data camera, the alignment target (target 1) and the docking station (target 2). The data camera is scripted so that it captures images of the environment from various angles. The target that the data camera focuses on during the data gathering process can either be the alignment target or the docking station. What is important is that, in our data preprocessing, we exclude images that don't have their respective target in them, as explained in the next section.

The data camera is not randomly moving, and then taking a picture. Rather, it moves right to left in a predefined amount of steps. At each step, the data camera looks at our target, and rotates a predefined angle horizontally. This is because each step has its own number of what we call rotational images, which are images that are taken at the position of the step, but horizontally rotated slightly each time. Once the data camera has reached the end of the row moving left, we can move forward to the next row, and repeat the process. We do this until we are close enough to the dock.

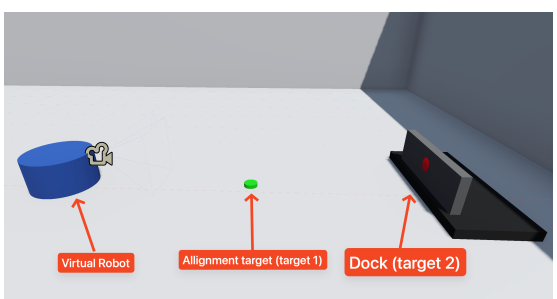

Figure 1: **Virtual world & Labels** In our virtual world, we make targets 1 and 2 distinguishable from their surrounding environment by giving them a unique color.

The reason we want to collect our data this way is because we want to make our data inclusive. With randomization, we don't know for sure if there will be images that are extremely similar, and we don't know if each angle is represented fairly. The diagram below shows what the data camera movement looks like from a bird's eye view.

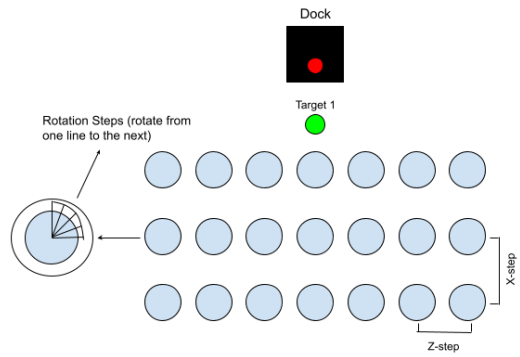

Figure 2: **Data Camera Movement Procedure** Our data camera moves in z-steps and x-steps, with rotations for each z-step. We obtain an image with each rotation step.

3.2 Data Preprocessing

We preprocess the collected images in order to have a dataset of images and their corresponding target values. In order to do this, we use OpenCV to detect the targets. This is really only a viable option due to the fact that we chose unique colors for our targets that aren't present anywhere else in the virtual world. In real life, contour detection like this wouldn't work, and we would need to manually label where the targets are.

Contour detection returns a list of coordinates that outline a target. We, however, only want one coordinate. We can achieve this by approximating the center of the target, which can be done by taking the mean of both the x and the y values of the contour coordinates list:

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i \ , \ \bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$$

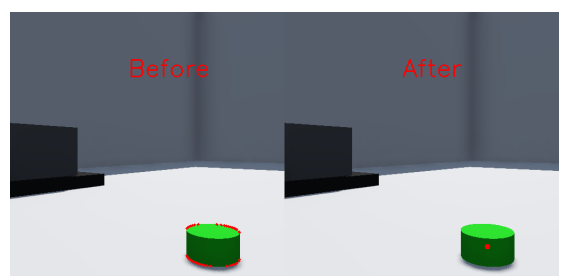

Figure 3: **Averaging contour points** By averaging contour points, we have one point of reference for the target object.

Where n is the number of points in the contour, and x_i and y_i are the x and y parts of each point in the contour, respectively. This gives us a single coordinate for each target, which we can then use as our target value for training. Due to our images being 512x512 pixels, the coordinates are in range from 0 to 512 (for both x and y). This means that the coordinates are then normalized from 0 to 512 to 0-1 by dividing the coordinates by 512. This way, model training and inference becomes easier and more efficient, as we are not dealing with large numbers [Singh and Singh, 2020]. Of course, when it comes to inferencing, we then multiply the model's outputs by 512 to get the actual coordinate values.

Our data is then sorted into two sets of data, one set specifically for the alignment target, and one specifically for the docking station. The reason we do this is because our custom models are not trained the same way YOLO models are trained, meaning we can't know for sure if the object is actually in frame. So, if one target is in an image, it doesn't necessarily mean that the other one is as well. The two sets of data share images, but one set may have more or less images than the other. This is because at some angles, one target may be visible while the other is not. In our case, the set of data that has more images is the one where we focused the target on during data gathering in the Godot game engine, which was target 1 (the green target).

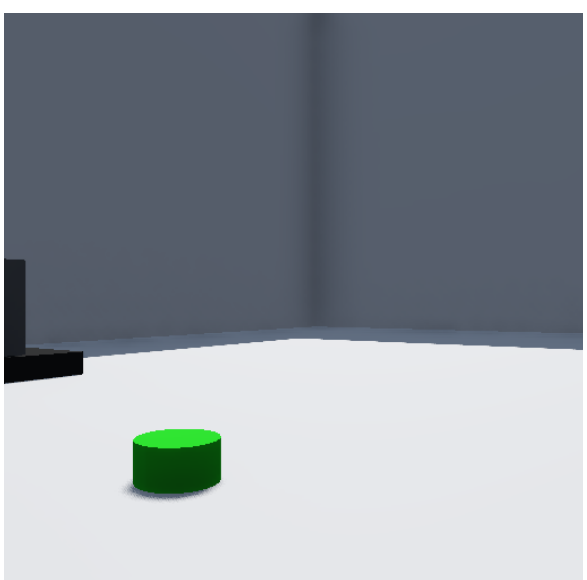

Figure 4: **Differing Image Amounts** An example of an image that includes target 1 (alignment target) but not the red part of target 2 (dock).

This sorting of data for each model is done through OpenCV. If contour detection is not present for one of the targets, the image is not included in the dataset for the target not found. This applies to both targets in the same function, meaning, if no targets are found, the image is essentially discarded. Again, using OpenCV for these operations is highly possible because we are in a virtual environment where the colors of the targets are unique to themselves.

The sorting of data for the two separate models also means that each model may be performed differently. However, in our data collection, the difference in image amount was around 1,148, meaning we had 1,148 more images for the target we focused on. This is not a concerning difference, and we can still train the models in very similar ways. The only difference is that the model that has less images may have a lower accuracy, which is where our custom data augmentation comes in.

3.3 Model Training

Near the end of this paper, we describe how our methodology can be implemented using a single YOLO model for object detection and prediction, all in one package. The reason why we have initially developed our project with custom models is because we have more flexibility regarding the architecture and performance of the models. By implementing the models using PyTorch, we can make lower level modifications to essentially see what the most conservative model is that can still perform well. This is important because we want to make sure that our model is as lightweight as possible while delivering adequate results. We don't want to have a model that is too heavy, because it will be running on a robot that has limited computational power.

Our training process is quite standard and is as follows:

1. Load and preprocess the images using the 'transforms' module from torchvision.
2. Define the custom dataset class 'TargetsDataset' to handle the input data.
3. Instantiate the model, loss function (L1 Loss), and optimizer (Adam).
4. Train the model over multiple epochs, applying custom image augmentation techniques to improve generalization.

The first, second, and third steps in the list above are all standard ways to train a PyTorch model, or

any basic deep learning model like the one we are creating. For the fourth step, the reason we need to apply custom image augmentation is because of coordinate predictions on the image. If we use the built-in method of image augmentation that PyTorch provides, the coordinates of the data will be off. The image is being transformed, but the coordinates are not.

In order to solve this, we implement our own vertical flip augmentation, which returns the transformed image and the transformed coordinates. This way, the model can learn to predict the correct coordinates, even if the image is flipped. The reason we only need to implement a vertical flip is because, if we also implemented the horizontal flip, the data would have very similar looking images, as the flipped images would be almost identical to the images taken on the other side of the robot during data gathering.

3.4 Inference

For inference, we use the trained models to output the values necessary to determine the next action, all based on a single input frame. When targeting either one of the targets during the docking procedure, we use some mathematical boundaries to determine the action the robot should take.

There are three areas the predicted coordinates can be in which are listed below. Each area corresponds to an action, which is referenced to in the server integration section:

1. The coordinates are outside the boundaries of what is considered centered with the target. In this case, the robot should turn either left or right, depending on the side the coordinates are on.

2. The coordinates are inside the boundaries of what is considered centered with the target. In this case, the robot should move forward.

3. The coordinates are inside the boundaries of what is considered centered with the target, under a certain threshold. This is a special case where the robot moves forward a fixed amount, and only applies to the alignment target (target 1).

Here we add a diagram showing the three areas on an example image

For the third case, which only applies to the alignment target (target 1) the robot has reached the target if the predicted coordinates are within the boundaries for the robot to be considered centered with the target, but are under a certain threshold. So, if the predicted coordinates are within a bottom center area of the image, the robot should move forward a fixed amount. The reason we move forward a fixed amount is because we want to be above the alignment target in order to ensure we are then centered with the docking station (target 2).

3.5 Server Integration

To facilitate real-time predictions, we can integrate the inference script with a Flask server. The server receives images via POST requests, runs the inference, and returns the next action the virtual robot should take. The actions possible are shown in the table below:

Action	Description
0	Turn left
1	Turn right
2	Move forward
3	Move forward fixed amount

We use a Flask server so that GDScript can communicate with the Python script. The Python script is running the server, and the GDScript is sending the encoded images to it. The server then sends back the action the robot should take.

Again, we incorporate such a process in the first place because we are dealing with a virtual world that needs to communicate with Python, which is separate. In reality, however, we would have the robot do everything on its own, without the need for a server. Everything would be incorporated between scripts rather than between a separate server and a script.

3.6 Evaluation

Todo: Determine if this section is necessary. We evaluate the performance of our model using various metrics, including accuracy and loss values. The evaluation scripts are included in tests/tests.py, which contain unit tests to ensure the correctness and robustness of our data processing and model training pipelines.

3.7 Experimental Setup

The experimental setup we have presented so far is about the virtual world and the docking procedure.

However, we also need to consider the hardware requirements for the training of our models, and what it took for us to get them running.

The models were trained on an M1 Macbook Air with 16 GB of RAM. The training process took around 40 minutes for each model using the CPU, and around 15-20 minutes using the GPU through PyTorch's MPS backend. The models trained using the hardware mentioned were the ones used to make our conclusions. Our purpose of training the models on a Macbook Air, or any other computational device without expensive hardware, is to show that our methodology is robust and that our inference requires very little computational power.

3.8 Conclusion

The methods used to get synthetic data were simple yet effective. We used the Godot game engine, which runs efficiently on low-power hardware (such as an arm chip MacBook), and we used OpenCV to obtain ground truth data. Using this data, we sorted it and trained our custom models, of which we had low-level control of to determine sufficient proficiency at an efficient computational cost. Our methodologies can be expanded in a similar manner to real-world applications and images, with the only difference being that we would need to manually gather and label the images instead of having them automatically taken by Godot and labeled using OpenCV.

4 Experimental Results

5 Discussion

Here we can, for example, compare how our methods can perform better than other methods that already exist. Example:

There have been many instances of robotic entities docking on their own without human intervention, even without the use of vision-based systems. For example, infrared sensors have been used for a while in order to detect and align with the dock. Other sensors include LiDAR and ultrasonic sensors. Our method of using only vision would work much better with these systems, however, when paired in conjunction to a pathfinding algorithm, likely based on machine learning. We would use the same cameras for docking to automate path making and traveling around an area. This allows for less hardware and complications regarding the overall robot functionality. Plus, relying on vision-based systems is much more flexible and adaptable to future and different environments.

6 Conclusion

In this paper, we presented a novel approach to robot docking using monocular vision and deep learning techniques. Our methodology leverages a simulated environment created in Godot for data collection, followed by a robust training pipeline using PyTorch. The trained model predicts the necessary rotation and distance adjustments for the robot to dock accurately.

We demonstrated the effectiveness of our approach through extensive experiments, showing that our model can achieve high accuracy in predicting docking maneuvers. The integration of the inference script with a Flask server enables real-time predictions, facilitating seamless communication with the robot's control system.

Our results indicate that monocular vision, combined with deep learning, can be a viable solution for robot docking, eliminating the need for additional sensors or complex reinforcement learning algorithms. This approach not only simplifies the hardware requirements but also reduces the computational overhead, making it suitable for real-world applications.

Future work will focus on improving the model's robustness and generalizability by incorporating more diverse training data and exploring advanced neural network architectures. Additionally, we plan to integrate the system with physical robots to validate its performance in real-world scenarios.

Overall, our work contributes to the field of autonomous robotics by providing an efficient and scalable solution for robot docking, paving the way for more advanced and autonomous robotic systems.

References

Salem Ameen, Kangaranmulle Siriwardana, and Theo Theodoridis. Optimizing deep learning models for raspberry pi. *arXiv preprint arXiv:2304.13039*, 2023. URL `https://arxiv.org/abs/2304.13039`.

Marc Peter Deisenroth, Carl Edward Rasmussen, and Dieter Fox. Learning to control a low-cost manipulator using data-efficient reinforcement learning. In Hugh Durrant-Whyte, Nicholas Roy, and Pieter Abbeel, editors, *Robotics: Science and Systems VII*, 2012. URL `https://www.roboticsproceedings.org/rss07/p08.pdf`.

Johannes Hoster, Sara Al-Sayed, Felix Biessmann, Alexander Glaser, Kristian Hildebrand, Igor Moric, and Tuong Vy Nguyen. Using game engines and machine learning to create synthetic satellite imagery for a tabletop verification exercise. *arXiv*

preprint arXiv:2404.11461, 2024. URL https://arxiv.org/abs/2404.11461.

Feiyu Jia, Misha Afaq, Ben Ripka, Quamrul Huda, and Rafiq Ahmad. Vision- and lidar-based autonomous docking and recharging of a mobile robot for machine tending in autonomous manufacturing environments. *Applied Sciences*, 13(19), 2023. ISSN 2076-3417. doi: 10.3390/app131910675. URL https://www.mdpi.com/2076-3417/13/19/10675.

Dalwinder Singh and Birmohan Singh. Investigating the impact of data normalization on classification performance. *Applied Soft Computing*, 97:105524, 2020. ISSN 1568-4946. doi: https://doi.org/10.1016/j.asoc.2019.105524. URL https://www.sciencedirect.com/science/article/pii/S1568494619302947.