Utilizing Deep Q-Learning for Efficient Object Sorting with a Robotic Arm

1st Takara Busby
California State University, Fresno
College of Science and Mathematics
Fresno, United States
takara47@mail.fresnostate.edu

2nd Tomas Garcia Gallardo
California State University, Fresno
College of Science and Mathematics
Fresno, United States
tomasggck@mail.fresnostate.edu

3rd Marcus Ramirez
California State University, Fresno
College of Science and Mathematics
Fresno, United States
RamiroSchmarim0@mail.fresnostate.edu

***Abstract*—Robotics is a growing field in today's market and the growing capabilities of machine learning has been a major boon for it. In this project we attempt to create a neural network that can control a robotic arm comprised of six servos to sort colored cubes. To do this we used Deep Q Learning and a simulation for the training of the model. The model used CNN's and fully connected layers in order to map features from input of a camera and servo angles. The simulation was done in PyBullet and the network made with PyTorch. Testing showed that the model was very susceptible to changes in learning rate and we actually had problems getting it to converge with it, oscillating between good rewards and bad rewards while remaining in the initial phase. This implies the model requires further fine-tuning and testing in order to get it to a state where it can be used in the real world.**

***Index Terms*—Reinforcement Learning, Q-Learning, Deep Q-Learning, Neural Networks, Artificial Intelligence, Robotics**

I. Introduction

Recent advancements in robotic systems have demonstrated their potential across a wide range of industries, including manufacturing, healthcare, and logistics. However, as these systems increase in complexity and capability, the challenge of enabling autonomous learning and decision-making becomes more pronounced. Traditional robotic approaches often rely on predefined programming or rule-based control strategies, which are limited in their ability to handle tasks that require real-time adaptation to dynamic and uncertain environments. One promising solution to this challenge is the application of reinforcement learning, specifically Deep Q-Learning Networks (DQN), which facilitate the autonomous optimization of robotic behaviors through iterative interactions with the environment. DQN is particularly suited for tasks that involve large state and action spaces, where traditional methods may not scale effectively or efficiently.

This project aims to apply a DQN to the problem of object sorting, using a 6 Degrees of Freedom (DOF) robotic arm to sort colored blocks (such as Legos) based on their color. The robot operates in a PyBullet simulation, where it learns to perform this task autonomously through reinforcement learning. By using a stationary camera for vision, the robot must detect the color of blocks and make decisions about how to manipulate and place them into the correct bins. The focus of the project is on enabling the robot to learn this sorting task effectively, even in the face of large state and action spaces that arise from the need for precise movements and real-time decision-making.

A. Context of Use

This project investigates the application of DQNs for robotic control in complex, high-dimensional environments. While object sorting may appear simple from a human perspective, it becomes significantly more intricate when decomposed into its individual components in robotics. The robot must engage in precise decision-making, considering factors such as object positioning, environmental dynamics, and sensor input. Traditional control methods, such as rule-based programming, finite state machines (FSM), and proportional-integral-derivative (PID) control, are often effective for simpler, more deterministic tasks. These methods typically rely on predefined instructions or fixed algorithms that are tailored to specific conditions or environments. However, they struggle with scalability and adaptability in complex, dynamic settings where the robot must not only recognize objects but also learn the optimal sequence of actions to complete tasks efficiently. Rule-based systems, for example, require exhaustive hand-coding of possible scenarios, while PID control can struggle to handle non-linear dynamics or highly variable environments. By leveraging DQN, the robotic arm autonomously navigates this expansive state and action space, optimizing its performance over time through trial-and-error interactions with the environment. This method enables the robot to gradually acquire the skills necessary for picking up objects and moving them around, a task that would otherwise require exhaustive manual programming and overly simplified control strategies.

B. Design Goals

The primary objective of this project is to develop a 6 DOF robotic arm capable of autonomously picking up a block and sorting it based on color. The core of this system is a DQN trained within a simulated environment using PyBullet, a physics simulation engine. PyBullet provides an ideal platform for training the robotic arm, as it allows for rapid iteration and testing in a controlled virtual environment, where the robot can interact with its surroundings and learn to perform the sorting task through reinforcement learning. The robot's task is to accurately detect a block, align its position with the block so as to pick it up, select the correct sorting bin, and manipulate the blocks accordingly.

While the simulation served as the primary focus of this project, it is also designed with the potential for real-world application in mind. By training the DQN in PyBullet, the system could be tested under a variety of conditions and configurations, simulating the complexities of real-world environments without the risks or constraints of physical hardware. Key challenges in this project include ensuring precise object detection and manipulation within the simulation, as well as achieving real-time decision-making based on visual inputs from the robot's camera. As the model's robustness improved through reinforcement learning, it would be a step closer to being transferred to a physical robotic arm, enabling real-world testing and deployment. This approach ensures that the robot's learned behaviors are both scalable and adaptable, laying the groundwork for future integration into actual robotic systems for object sorting tasks.

C. Contributions

This project makes significant contributions to the fields of robotics and artificial intelligence by integrating deep reinforcement learning with a 6 DOF robotic arm for real-time object sorting. By leveraging a PyBullet-based simulation for training and a Raspberry Pi for real-world execution, the project advances the practical application of AI in physical systems. The use of lidar for depth perception, combined with adaptive neural network models, offers a cost-effective solution to enhance spatial awareness and precision in robotic tasks. This work bridges the gap between simulated environments and real-world robotics, providing a foundation for further developments in autonomous object manipulation.

Each team member played a crucial role in the successful implementation of the project. Takara, being the most knowledgeable in neural network design, took the lead in setting up classes for the robot's states and actions, ensuring seamless integration with the DQN. Additionally, she set up the physics simulation environment and a model of the 6 DOF robot arm, laying a solid foundation for our deep Q-learning approach while ensuring accurate interaction in the virtual environment.

Tomas took on the role of our de facto electrical engineer, focusing on the physical setup of the robot arm, despite its current omission from the project's active phase. Additionally, he contributed to earlier simulation efforts in ROS and Gazebo, which provided a stepping stone before transitioning to PyBullet.

Marcus concentrated on hardware-related code, developing systems for servo control and managing camera inputs and outputs. This included adapting the project from dual-camera depth mapping to a single-camera setup and configuring the lidar sensor for depth perception. He also facilitated communication between the Raspberry Pi and PC via TCP socket programming, which supported a brief exploration of real-world training without relying on simulation before the idea was abandoned in favor of a virtual physics simulation.

II. System Overview

Much of the system and its design was based around a 6 DOF robot arm kit found off of Amazon. The kit itself included servos to be connected to an external controller, advertised to work with a Raspberry Pi (denoted from here on as just Pi). As such, we had to ensure our system for the robot could be controlled by a Pi and any additional components required on the hardware side could mesh well with these two core components

A. Foundational Hardware

1) Servos: As previously mentioned, the kit included six MG996R digital servo motors, each weighing 55g with dimensions of approximately 40.7 x 19.7 x 42.9 mm. These servos deliver a stall torque of 9.4 kgf and can generate up to 10 kg of torque, enabling the arm to handle various tasks. Each motor operates within a voltage range of 4.8V to 7.2V, with a stall current of 2.5A at 6V. They support a 120-degree rotation range (60 degrees in each direction) and function within a temperature range of 0°C to 55°C. The robotic arm also includes multiple mechanical and structural components, such as brackets, bearings, and extensions, to assemble a fully functional mechanism.

2) Raspberry Pi 5: To serve as the computational hub of the project for control of the robot arm, we opted to use a Rasperry Pi 5 (the latest model as of the time of writing this) with 8 GB of ram. It features a Broadcom BCM2712 2.4GHz quad-core 64-bit Arm Cortex-A76 CPU – suitable for handling data processing and control tasks. The device also includes a VideoCore VII GPU for graphical processing and supports dual 4K displays through HDMI. Connectivity to the Pi is facilitated by dual-band 802.11ac Wi-Fi, Bluetooth 5.0, and multiple USB ports. The Pi supports high-speed data transfer via PCIe 2.0 and provides compatibility with a wide range of peripherals through its GPIO header. These capabilities make it a versatile platform for integrating sensors, cameras, and control algorithms.

3) Raspberry Pi Camera Module V2: The camera module used in this project is designed specifically for compatibility with a Pi. It employs a Sony IMX219 sensor with an 8-megapixel resolution, offering sufficient detail for object detection and classification tasks. It also captures video at resolutions up to 1080p at 30fps, with additional frame rate options for lower resolutions. The lens is fixed-focus with a focal length of 3.04 mm and an aperture of f/2.0, providing a horizontal field of view of 62.2 degrees and a vertical field of view of 48.8 degrees. Its lightweight and compact design allow for easy integration with the Pi via the CSI port using a 15-pin ribbon cable.

4) TF-Luna Lidar Sensor: The TF-Luna lidar sensor was considered for depth perception within the project due to its ability to operate over a range of 0.2m to 8m and its accuracy of ±6 cm for short distances and ±2% for longer ranges. However, during testing, challenges arose in maintaining a consistent data flow through the serial ports, which prevented the sensor from being successfully integrated into the system.

Despite these issues, the lidar remains in consideration for future developments, as improvements to the data handling and communication processes could allow for its reimplementation to enhance depth perception capabilities.

5) Power Supply: The robot arm utilized a Zeee 7.2V NiMH battery with a capacity of 3600mAh as the primary power source. This battery operates with a 6-cell configuration, providing sufficient voltage and capacity to drive the robotic arm's servos and other connected components. It features a Tamiya connector for secure and reliable connections to the power distribution system. With dimensions of 135.5 x 45.5 x 22.5 mm and a weight of approximately 367.3 g, the battery is compact and portable, making it suitable for integration into the project's hardware assembly. The battery's high capacity ensures extended operational time during real-world testing and deployment.

B. Software

1) Python: Most of the code for this project was written in Python due to its extensive ecosystem of libraries, which provided robust support for the various tasks we needed to execute. Python's versatility allowed us to seamlessly integrate components for machine learning, hardware control through the Raspberry Pi, and simulation. This rich set of libraries significantly accelerated development and simplified the integration of complex systems.

2) PyTorch: PyTorch is a free open-source library for implementing neural networks. It contains a large amount of tools for using different types of layers in Neural networks. It also allows easy implementation of various loss functions and back-propagation algorithms.

3) PyBullet: PyBullet is free open-souce physics engine good for simulating a likeness of the real world. It is an allows for easy loading of custom robots through urdf files. It is useful as there is not a lot of setup required while still offering a wide range of options. Such as, support to get camera output and handling collision checking.

4) Blender: Is a free 3d modeling software that is used to model the robot.

5) Google Colab: Google Colab is an online service provided by Google that allows the use of their high end computers to run code that otherwise might be too costly to run for others. It also allows a group to share code and work on it together. This was used mostly for the Google hardware to facilitate more testing as it severely reduced the time required to perform a test.

6) OpenCV: In order for the robotic arm to make sense of the video feed data we turned to OpenCV software. The main objective of our project is to train the Robotic Arm to pick up an object therefore the object detection capabilities of OpenCV were instrumental in successfully carrying out the reinforcement learning training. "OpenCV (Open Source Computer Vision Library) is an open source computer vision and machine learning software library"(2020 OpenCV). OpenCV greatly facilitates the computer vision aspect of the project allowing us to reuse already fine-tuned algorithms and freeing us to focus on training the model.

7) GPIO: The GPIO (General Purpose Input/Output) library is a Python library that provides an interface for controlling the GPIO pins on the Raspberry Pi. These pins are used to send signals to and receive signals from external devices, making them ideal for controlling the six servos in our project. Each servo motor is connected to a designated GPIO pin, and through the library, we can generate PWM (Pulse Width Modulation) signals to control their rotation angles precisely. By specifying the duty cycle of the PWM signal, the library allows us to adjust the position of the servos, enabling the robot arm to move in the desired directions. This setup provides a direct and efficient way to translate commands from the trained neural network into real-world motion, ensuring the servos respond accurately to the input signals.

III. Experimentation Procedure

There were multiple factors that needed to be determined before any type of training was implemented. First, the neural network architecture needed to be deicide upon. This included examining what type of input would be used and what type of output desired. Second, is the rewards system that will be used to direct the network towards the desired behavior. Thirdly, the model policy needs to be optimized and hyperparameters set.

A. Neural Network

The first consideration for the setup of the Neural Network was the type of input that the model would receive. The robot is planned to use a camera for data on its environment, with an image quality of 480 x 270 x rgb. The network would also take the angles of the servos as an input.

The network starts as a Convolutional Network in order to get features from the image. This part of the network consists of 3 layers. Each layer is a 2D convolution with 3 channel input and output for the rgb. This is important since the eventual goal is to sort things based on color. Each layer also has a max pooling to shrink the image dimensions by two for each layer. This is helpful to sort out important features and ensure that the number of units is not too large when transitioning to fully connected layers. On the third layer, the output is flattened to one dimension and concatenated with the servo angles.

The fourth layer is the first fully connected layer and has an input size of 5947 which is the flattened output of the third layer concatenated with the servo angles. This layer has an output size of 5832. The next three layers halve the size of the input until on the seventh layer there is an output of 729. This corresponds with the action space of the robot.

B. Training via PyBullet

In order to simulate the robot, Pybullet was used as a virtual environment. In the environment are three main elements. The robot arm, the cube to be picked up, and the tray to place the cube in. The robot arm was modeled in blender based on measurements from a physical robot. The obj files of the components were put together in a .urdf file and loaded in the environment. The cube and tray were made with simple shapes in the .urdf file.

Next the camera needs to be setup in order to get feed back for the network. The camera is setup to be linked to a specific object part of the arm. The camera is designed to be stationary looking over the scene and is positioned behind and to the right of the arm. It looks down and to the right giving a good view of the right and front of the arm. The image resolution is 480 x 270 with rgb and uses openGL to get a image from the 3d simulation. The combination of the image and the servo angles represents the state space of the model

The simulation is also responsible for the rewards as meta-data can be used to determine what is a good action and what is not. For the arm to pick up and move the block it was decided to break the rewards into four phases. One rule we used throughout the phases was that if an action gave a reward the opposite action would give a negative reward. This was done to try and keep the network from learning to game the system for greater rewards. The first phase gave rewards for opening the claw and moving the center point between the claw pads closer toward the center point of the cube. After it got close enough the simulation would move to phase 2. In phase 2 the arm is rewarded for closing the claw and will move to phase 3 when both claw pads touch the cube. Once they are touching the rewards come for moving the cube closer to a point above the tray where it switches to phase 4. Phase 4 rewards the network the closer the cube gets to the base of the tray. In phases 2 through 3 there is a penalty if the cube gets farther away from the center point between the pads of the claw. If the distance is too great the sim moves back to phase 1.

The action space consists of the 729 possible actions that the arm can be taken. Each servo consists of 3 possible actions. They can move in either direction or not move at all. There are six servos so there are 729 possible permutations. Representing this for control in the simulation the action is represented from the index in the output of the model with the highest expected value. That number is then changed to base 3 with each digit representing the control command for each servo. This allows the model to control each servo simultaneously.

C. Optimizing Model

Optimizing the model comes in two steps the architecture for optimizing at each step and the hyerparameters to control how this occurs. The architecture reflects the methodology at each step and takes certain changable parameters to change things faster or slower.

1) Optimization Architecture: The optimization uses two models, a target network and a policy network. There is also a replay memory which records transitions which are used in batches with the target network in order to optimize the policy network. The policy is optimized at every step with the exception of the beginning when the replay memory is too small. During optimization the batch is used with the target network to update the weights of the policy network. This helps reduce the noise to increase the chances of convergence. The target network is soft updated at every step as a weighted combination of the target networks current weights and the policy weights.

At each optimization the weights of the policy network are updated based on the target network and expected rewards based on the replay memory. The policy is then put through back-propagation with a loss function. This network uses Huber loss which can help with outliers.[1]

2) Hyperparameters: The first hyperparameter is the size of the replay memory. The large this is the better that the model can generalize but it does take up GPU memory so it needs to be managed to not be two large. The second hyperparmaeter is the learning rate which controls how fast the weights can change during each update. The third hyperparameter is the Epsilon which represents whether a random action should be taken in order to explore more or two use the policy in order too go with what has already been learned. This uses the following equation:

$$eps\_thresold = EPS\_END + \frac{(EPS\_START - EPS\_END)}{\exp 1 * \frac{steps\_done}{EPS\_DECAY}} \tag{1}$$

This will steadily decrease the chances of making random steps as the equation goes on. The model has a steps_done variable for all the phases so that the model can explore at each phase separate from each other. The Epsilon consists of three parts the start the end and the decay. The start and end control the range of the Epsilon threshold and the decay represents how many steps are needed to reach the end.

The fourth hyperparameter is the Gamma which simulates how much the length of time is detrimental to getting the reward. A higher Gamma means that time is not as much of an issue whereas a lower gamma makes time a more important factor.

The fifth hyperparameter is tau which controls the ration that the target network is soft updated. The equation for updating the target net is:

$$T\_weights = P\_weights * TAU + T\_weights * (1 - TAU) \tag{2}$$

The tau controls wether the new target weights favor the policy weights more or the target weights.

The sixth hyperparameter is the batch size which is used to determine what size of the replay memory used when optimizing the policy network. Larger batch sizes can give more variety but also take longer for computation and can also help to converge faster.

The seventh hyperparameter is the max amount of time that can be spent on each simulation. This ensures that it will not continue in one forever and eventually reset. The only other way for the sim to end is for the cube to be successfully moved to the tray. Setting this to higher lets the sim explore more as the arm can move farther from it's position but will increase the time it takes to do each episode.

The eighth hyperparameter is related to the seventh in that it is the total amount of episodes to be run with this training cycle. Having more episodes gives more training time and coupled with a smaller time spent on each episode explores more from the start point hopefully learning that faster.

D. *Recording Results*

Certain information was recorded and stored for each episode. The duration of the episode was stored to see runtime as well as if the episode ended early. Next both the max phase reached and the phase ended on are also recorded. The reward was also recorded with the total reward for the entire episode and total reward for each phase. All this information was recorded in a .csv file for easy access. Also the finished model was also stored so it could be used again if needed.

IV. Results

The model was tested on a number of different configurations of the hyperparameters. With the bulk of the focus being on the epsilon decay and duration of the episodes. There was also some testing done on the learning rate, but unfortunately due to time constraints we were unable to do as much testing as we would of likes. This led to a lot of the testing to only take place within the first phase where the arm would be rewarded for going toward the block and penalized for moving away from the block.

A. *Simulation Observations*

On the majority of our tests we were seeing very erratic behavior with the total reward gained in each episode fluctuating widely despite the model staying in its initial searching state. An interesting thing to note though was that on a test where the max duration per episode was increased to 1000, there was a noticeable trend towards negative rewards as seen in figure 1. This coupled with the erratic behavior led to the conclusion that the model was not learning enough since it would not settle even on a bad action.

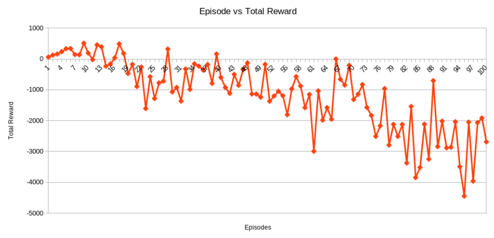


Fig. 1. Graph of rewards episode for duration of 1000

Following this it was decided to cut back the episode durations and increase the number of episodes to decrease computation time and see if convergence could be achieved. The next step was to increase the learning rate by a good factor to observe the results. Unfortunately this did not get the desired results and showed more chaos as seen in figure 2.

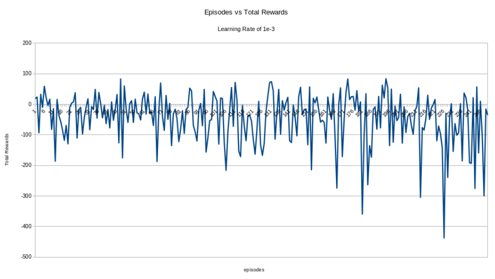


Fig. 2. Graph of rewards per episode for LR=1e-3

From this we rethought our idea of it needing to converge faster and guessed that it was learning too fast and as a result was jumping all over the place. Because of this, we lowered the learning rate and ran another test to see how that would go. This proved to be a good decision as the rewards seemed to stabilize in the positive direction as shown in figure 3.

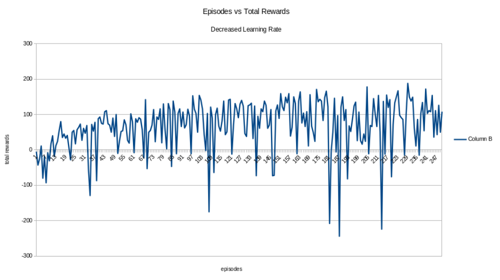


Fig. 3. Graph of rewards per episode for LR=1e-5

B. *Real World Implications*

The results of our simulation indicate significant challenges that would arise in real-world applications of the robot arm, particularly in locating and manipulating objects. In our tests, the arm was unable to reliably find the block, leading to a constant exploration of the state space in search of it. This erratic behavior, where the arm oscillated between actions without converging on a solution, would not be effective in real-world environments where precision and efficiency are critical.

One major contributing factor to this issue is the reliance on a single camera as the sole source of input for determining the position of the arm and block. The absence of depth perception from the camera led to confusion in spatial awareness, with the arm being unable to correctly estimate distances or adjust its movements to successfully approach and manipulate the block. In real-world scenarios, this limitation would likely result in the arm continuously searching without ever successfully completing the task, thereby failing to achieve the goal of object manipulation.

To address these challenges in a practical setting, the addition of depth-sensing technology, such as a lidar sensor, would provide critical spatial data to the system. Depth perception would enable the arm to better understand its environment and the block's location, reducing the trial-and-error process observed in the simulation. This would improve the efficiency of the robot's decision-making and allow it to more accurately approach, grasp, and manipulate objects. Thus, the simulation results highlight the insufficiency of using a single camera for visual input in a task that requires precise spatial awareness and manipulation, underscoring the need for more comprehensive sensor integration in real-world applications.

V. Conclusions

In evaluating our simulation, we concluded that it was not behaving as expected. This issue arose due to inefficiencies in our setup and the reliance on the camera alone for input, and would thus make it currently unsuitable for real world testing of the robot.

A. Retrospective

Reflecting on our project, there are areas where we believe different decisions could have optimized our workflow and efficiency. A significant challenge was the time invested in setting up ROS and Gazebo, which, while powerful tools, required a steep learning curve for configuration and integration. This effort ultimately proved unnecessary after transitioning to PyBullet, a simpler and more suitable alternative for our needs.

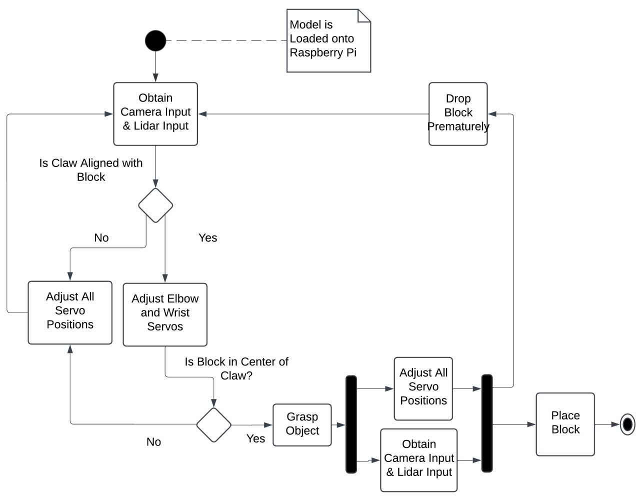
Additionally, our initial indecision about whether to prioritize physics simulation or real-world training led to inefficiencies. We dedicated time and energy to developing and debugging code for real-world hardware control that went unused. These experiences highlight the importance of clear project goals and early decision-making to minimize wasted effort and streamline development.

B. Future Endeavors

In future endeavors, we plan to dedicate more time to training the model and performing hyperparameter tuning, as we believe this project requires extensive fine-tuning to achieve the desired performance. Given the complexity of the tasks at hand, especially in integrating both vision and depth perception, we anticipate that the neural network will need further refinement to handle more intricate scenarios effectively. The additional time will allow us to optimize the model's parameters and improve its ability to accurately detect and manipulate objects in dynamic environments. This enhanced focus on training and fine-tuning will be essential in ensuring that the model behaves as intended when applied to real-world hardware and scenarios.

Upon getting the model to a desirable state, the next main task we would like to take is to apply the trained model in the real world for testing, given that we already have the hardware needed and have established plans for the way information would flow through the Pi and the arm.

As depicted in Figure 4, the Pi would serve as the main host for the trained model, being loaded onto it via PyTorch so as to act with the cameras and lidar sensor to adjust the servos of the robot arm so it can pick up and move the block around. Since the simulation currently relies on a camera as a source of input, we would of course need to go back and update it to accomodate for the lidar input before we test it in the real world, so the simulation would behave in accordance to the depth perception provided by the lidar sensor. A key part of this future work involves reconfiguring and continuing to train the neural network to improve its performance and ensure the model behaves as intended in more complex scenarios.



Model is Loaded onto Raspberry Pi
Obtain Camera Input & Lidar Input
Drop Block Prematurely
Is Claw Aligned with Block
No
Yes
Adjust All Servo Positions
Adjust Elbow and Wrist Servos
Is Block in Center of Claw?
No
Yes
Grasp Object
Adjust All Servo Positions
Obtain Camera Input & Lidar Input
Place Block

Fig. 4. An activity diagram to present how the robot behaves as it finds, grabs, and places a block.

Since the simulation currently relies solely on a camera for input on the arm and block positions, we would also need to update it to accommodate lidar input. The TF Luna lidar sensor provides distance measurements, temperature readings, and signal strength through specific byte sequences transmitted to the Raspberry Pi [2]. The distance data, in millimeters, allows the robot to measure the distance to objects, while the temperature reading helps account for sensor performance changes due to internal temperature fluctuations. The signal strength indicates the quality of the reflected signal. Incorporating this lidar data into the simulation would enable depth perception, ensuring the simulation behaves in alignment with real-world conditions. Additionally, resolving the initial issues with the flow of information across serial ports on the Raspberry Pi would enable seamless integration of the lidar sensor, providing the robot with enhanced spatial awareness and the capability to perform more precise object manipulation tasks.

Acknowledgment

We would like to thank Dr. David Ruby for his guidance in the comprehensive Principles of AI course at Fresno State. His lectures and online resources deepened our understanding of reinforcement learning, primarily q learning and its various forms, giving us ideas on how to implement them in the field of robotics. This knowledge will be a valuable asset in our future academic and professional endeavors.

References

[1] Paszke, A., & Towers, M. (n.d.). Reinforcement learning (DQN) tutorial. Reinforcement Learning (DQN) Tutorial - PyTorch Tutorials 2.5.0+cu124 documentation. https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html

[2] MakerPortal. (n.d.). tfluna-python. GitHub. https://github.com/makerportal/tfluna-python

[3] About. OpenCV. (2020, November 4). https://opencv.org/about/

[4] Dr. David Ruby, California State Univeristy - Fresno