NORTHWEST NAZARENE UNIVERSITY

Robotic Manipulators Using Deep Meta Reinforcement Learning to Catch Objects in Space

THESIS
Submitted to the Department of Mathematics and Computer Science
in partial fulfillment of the requirements
for the degree of
BACHELOR OF ARTS

Aaron Borger
2022

THESIS
Submitted to the Department of Mathematics and Computer Science
in partial fulfillment of the requirements
for the degree of
BACHELOR OF ARTS)

Aaron Borger
2022

Robotic Manipulators Using Deep Meta Reinforcement Learning to Catch Objects in Space

Author: Aaron Borger
Aaron Borger

Approved: Dale A Hamilton
Dale Hamilton, Ph.D., Associate Professor of Computer Science
Department of Mathematics and Computer Science, Faculty Advisor

Approved: Stephen Parke
Stephen Parke, Ph.D., Professor of Electrical Engineering Department of Engineering and Physics, Second Reader

Approved: Barry Myers
Barry L. Myers, Ph.D., Chair,
Department of Mathematics & Computer Science

Abstract

Robotic Manipulators Using Deep Meta Reinforcement Learning to Catch Objects in Space
BORGER, AARON (Department of Mathematics and Computer Science),
HAMILTON, DR. DALE (Department of Mathematics and Computer Science).

This work proposes a method to train an artificial intelligent robot in simulation and then re-train it in the real-world space environment. This method utilizes a meta learning algorithm that allows an agent to re-train while in space and learn to catch an object with a robotic manipulator after only a few shots. This work seeks to develop the building block for intelligent astro-robotics by learning to catch a ball in space. Subsequent versions may be able to use the same method to learn to grasp uncontrollable, non-uniform objects in space. This effort will be beneficial towards solving multiple challenging problems such as assembling and servicing spacecraft in orbit by learning to manipulate tools and components in zero gravity as well as space debris removal by learning to catch resident space objects. The reinforcement learning algorithm at the core combines the Twin Delayed Deep Deterministic Policy Gradients algorithm (TD3) with the implicit model-agnostic meta-learning algorithm (iMAML). This new algorithm allows for efficient few-shot learning with continuous observations, such as the position and velocity of the object as detected by a Mask Region-Based Convolutional Neural Network (MR-CNN), and continuous actions such as the location, orientation, and time of the grasp.

Acknowledgements

I would like to begin by thanking the students on the Rocksat team who spent countless hours making this insane idea of a project a reality: Jonathan Herberger, Riley Mark, Juliette Haggith, Nathan Appleby, Samuel Mark, Jacob Sutherland, Jakob Waltman, Cole McCall, and Camden McGath. Thank you for your hard work and friendship.

In addition, I would not have been able to succeed in completing this project without the expertise of my professors who helped me avoid numerous pitfalls: Dr. Dale Hamilton, Dr. Dan Lawrence, Dr. Autumn Pratt, and Dr. Stephen Parke. Without you, I would still be writing my thesis as I attempt to cram every little detail into it or making wire harnesses with fifteen connectors when I could have used one.

Finally, I would like to thank NASA's Idaho Space Grant Consortium and Colorado Space Grant Consortium for funding this project and the RockSat-X program. You have given me the opportunity to gain the experience required to accomplish my dream of working in both artificial intelligence and aerospace industries.

Table of Contents

Abstract ..... **iii**

Acknowledgements ..... **iv**

List of Figures ..... **vii**

1. Introduction 1

2. Background 3

2.1. Solving Rubik's Cube with a Robot Hand 3

2.2. Optimal Stroke Learning with Policy Gradient Approach 4

2.3. 6-DOF GraspNet 4

2.4. MAML Algorithm 5

2.5. Meta-Learning with Implicit Gradients (iMAML Algorithm) 6

2.6. TD3 Algorithm 7

3. Experiment Overview 10

4. Artificial Intelligence Subsystem 12

4.1. Subsystem Overview 12

4.2. Object Detection 14

4.3. Kalman Filter 15

4.4. Catching Agent 20

4.5. Trajectory Prediction 24

4.6. Grasp Generator 28

4.7. Meta TD3 Algorithm 29

5. Discussion 32

References 36

Appendix 39

List of Figures

Figure 1.1 - Payload Design (Launch Configuration) 2

Figure 1.2 - Simulated Throw and Catch Maneuver..... 2

Figure 2.1 - GraspNet VAE Diagrm 5

Figure 2.2 - General TD3 Exploration Diagram 7

Figure 2.3 - TD3 Critic Update Diagram..... 9

Figure 4.1 - AI Subsystem Flow Diagram 12

Figure 4.2 - Object Detection with Similar Background 15

Figure 4.3 - Kalman Filter Block Diagram 16

Figure 4.4 - Catching Agent Exploration Diagram..... 20

Figure 4.5 - AI Learning Kinematic Possible Catches 22

Figure 4.6 - Average Reward During Training Session 23

Figure 4.7 - Trajectory Slice Denormalization 24

Figure 4.8 - 2D Trajectory of an Object 27

Figure 4.9 - Distance from Object to Robot Base..... 27

Figure 4.10 - iMAML Algorithm (Rajeswaran et al., 2019) 30

Figure 4.11 - Meta-TD3 Training 32

1. Introduction

Current space fairing robotics struggle to adapt to varying tasks and environments. Intelligent space fairing robots have the potential to assist astronauts and solve challenging problems such as space debris removal, collision avoidance, and in-orbit spacecraft servicing and assembly. Current space fairing robots require manual control, which may prove difficult for distant uncrewed missions where communication times deem remote control impractical. Remote control will also hinder large robotic workforces as the necessity for human involvement would be too great. Deep Reinforcement Learning (Deep RL) has demonstrated it can learn solutions to complex problems, such as solving a Rubix Cube with a dexterous robotic hand (OpenAI et al., 2019). This technology could be revolutionary in the aerospace industry as an intelligent, dexterous robotic hand and arm combo could learn to perform tasks in space and decrease the need for human involvement.

This work aims to develop adaptive robotic arms that train in simulation and then complete a complex task in the real-world space environment. The complex task involves, two robotic arms that throw and catch a ball in space. The main contribution of this work is to demonstrate deep meta reinforcement learning can bridge the gap between simulation and the harsh environment of space and should become a focus for developing intelligent robots for the aerospace industry.

Figure 1.1 displays a simulated version of the robotic arms in the launch configuration. After the skirt of the rocket deploys and the robotic arms are exposed to the vacuum of space, the arms will move to the extended position and begin the experiment, as shown in Figure 1.2.

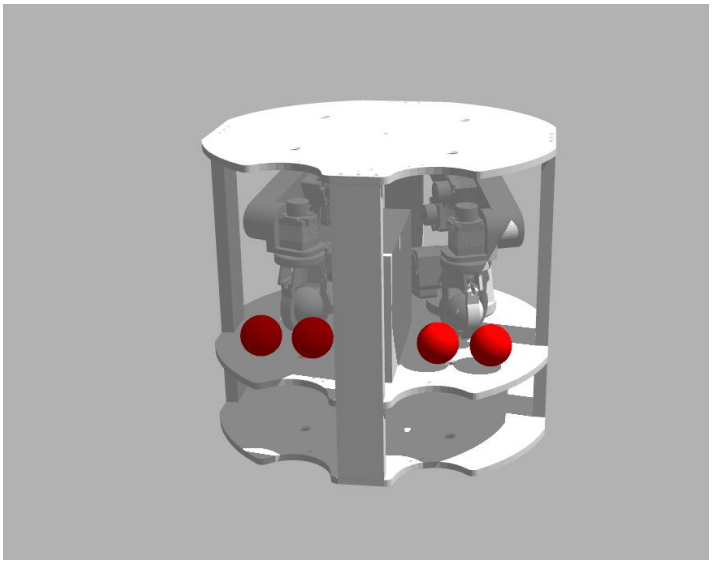


Figure 1.1 - Payload Design (Launch Configuration)

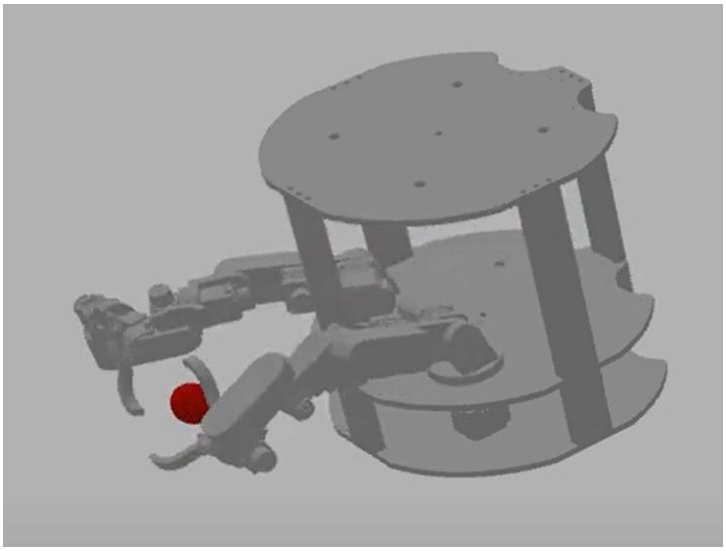


Figure 1.2 - Simulated Throw and Catch Maneuver

2. Background

Recently, deep reinforcement learning (deep RL) has become a standard method to enable robots to adapt to changes in the environment (Mahmood et al., 2018). Deep RL is a branch of machine learning that combines learning through trial and error (reinforcement learning) with deep neural networks (deep learning). OpenAI has provided an extensive repository of deep RL documentation (Achiam, 2018). Although deep RL has shown promising results, reinforcement learning simulates learning through trial and error, which requires attempting a task thousands of times before the agent is successful. Training for deep RL is impractical for robots as their mechanical parts may not be able to endure the training process. In addition, creating an environment that a robot can train in without human involvement is impractical, if not impossible. Artificial intelligence researchers have begun to train intelligent robots in simulations and then deploy them into real-world robots to solve this issue. Training an agent in simulation is a challenging task as it is nearly impossible to simulate the physical world perfectly. However, recent advancements have shown it is possible to overcome the imperfections in simulators (Cutler & How, 2015; Koos et al., 2010). The following publications discuss methods helpful towards learning how to catch objects in space.

2.1. Solving Rubik's Cube with a Robot Hand

OpenAI developed a robotic hand that learned with reinforcement learning in simulation to solve a Rubix cube in the real world. OpenAI demonstrated training in simulation with domain randomization (randomizing aspects of the environment to

generate a large dataset), and meta-learning could successfully transfer simulated learning to real-world performance. However, OpenAI was able to test the performance of their algorithm multiple times as their goal was to complete a task on Earth. If MARSHA combines domain randomization and meta-learning, the model does not need to be successful from training in simulation alone because it can continue to learn while in space (OpenAI et al., 2019). In addition, once the arm catches an object, a combination of MARSHA and OpenAI's robotic hand could be used to complete complex procedures in space.

2.2. Optimal Stroke Learning with Policy Gradient Approach

(Gao et al., 2021) proposes a method to use the TD3 algorithm to learn the optimal stroke to hit a ping pong ball at the desired target and does so by training in simulation and then performing the task in the real world. Catching and hitting an object are similar problems so this paper has provided many valuable ideas.

2.3. 6-DOF GraspNet

Nvidia developed a method of generating and evaluating grasps from 3D point clouds of non-uniform shaped objects with a variational auto-encoder (VAE) and grasp evaluator network (Mousavian et al., 2019). The GraspNet learns grasps by encoding successful grasp configurations to a latent space and then using that as well as the point cloud as the input to the decoder network which outputs a reconstructed grasp. This can be trained by using the difference between the reconstructed grasp and the original grasp as the loss. After training is complete, random grasps can be generated by using

random variables for the latent space. A distribution of these grasps are then evaluated with an evaluator network where the highest scoring one is chosen. MARSHA has been designed to be easily adjusted to learn how to catch non-uniform shaped objects. Currently MARSHA's action space contains spherical coordinates that represent the grasp. Therefore, the decoder is simply a manual conversion between spherical coordinates in the object space to a grasp vector. The TD3 algorithm was chosen due to its continuous action space as reconstructed grasps no longer need to be evaluated as MARSHA can directly output the latent space representation of the optimum grasp configuration. Figure 2.1 shows a diagram from the paper that demonstrates the VAE.

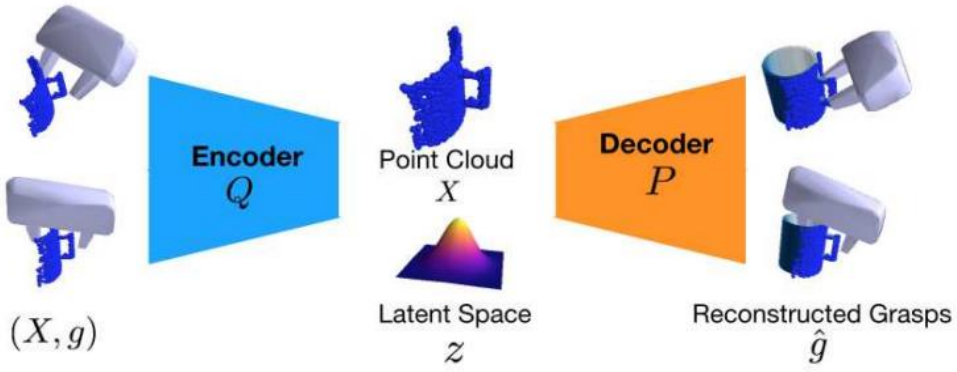


Figure 2.1 - GraspNet VAE Diagrm

2.4. MAML Algorithm

While OpenAI's Rubix cube solver (OpenAI et al., 2019) implements meta learning with an LSTM, a model-agnostic meta-learning (MAML) method (Finn et al., 2017) can be implemented with any model or algorithm. In the reinforcement learning version of the model-agnostic method, two sets of models are learned, a meta-model and a mesa-model. The mesa model's parameters are initialized with the meta model's parameters. The mesa model is then trained with just a few episodes in a specific task

or environment. After the mesa model has been trained the meta model is updated with stochastic gradient descent where the loss function is simply the inverse of the reward function. This is repeated over multiple tasks or environments until the meta-model learns parameters that can be trained with a few shots to succeed in an unseen task or environment. In the case of astro-robotics, meta-learning could allow robots to learn how to manipulate objects or maneuver in varying levels and orientations of gravity. Machine learning usually requires a large amount of data to produce sufficient results, but (Finn et al., 2017) show that meta-learning is capable of solving few shot learning problems where there are only a few opportunities to learn.

2.5. Meta-Learning with Implicit Gradients (iMAML Algorithm)

Model-Agnostic Meta-Learning (MAML) is an algorithm that allows meta-learning to be used for any neural network architecture or reinforcement learning algorithm (Finn et al., 2017). The iMAML algorithm is the sequel to MAML and features a far more efficient way to calculate the gradients of gradients required to learn how to learn (Rajeswaran et al., 2019**).** Both methods involve calculating the initial parameters (weights and biases) that minimize the loss on multiple tasks after the initial parameters are retrained in a few shots. In MAML, the meta-gradient is calculated by differentiating over every operation required to train the parameters for each task and then summing the gradients. This calculation would be very expensive and would take a long period of time to perform on a super-computer, much less a Jetson Nano. iMAML proposes a solution to this problem by approximating the gradients. (Rajeswaran et al., 2019). Section 5.6 describes how iMAML is combined with the

TD3 algorithm to allow meta reinforcement learning with continuous observations and actions.

2.6. TD3 Algorithm

The Twin Delayed Deep Deterministic Policy Gradients Algorithm (TD3) was proposed by (Fujimoto et al., 2018). TD3 is a deep reinforcement learning algorithm, so it combines deep neural networks with reinforcement learning (learning through trial and error). TD3 is an off-policy algorithm so it collects percepts from the environment, performs an action to "explore" the environment and then receives a reward for how well that action achieved the given goal. However, rather than an on-policy algorithm which would update its parameters after every iteration, the percept, action, and reward are stored in a replay buffer to be trained on later. (Achiam, 2018; Fujimoto et al., 2018). Figure 2.2 shows a block diagram for one iteration of this loop.

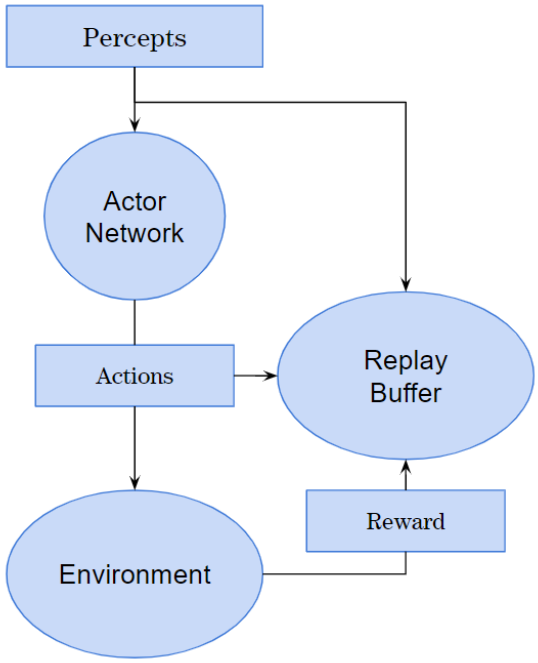


Figure 2.2 - General TD3 Exploration Diagram

The TD3 algorithm is also an actor critic algorithm. Therefore, the actor network shown in Figure 2.2 is responsible for producing an action given the percepts and a critic network is used to estimate the Q-values similar to Q-learning and the deep Q network (DQN) (Mnih et al., 2013; Watkins & Dayan, 1992). Reinforcement learning uses a reward value to determine how good an action was given the percepts, but the use of Q-values allows for temporal difference learning or the ability to predict the future rewards that may be gained from the chosen action (Barto, 2007). While the DQN algorithm can estimate the Q-value for a discrete set of actions with a single neural network, the actor-critic method uses two neural networks (actor network and critic network) to estimate the Q-value for a continuous set of actions from a continuous set of states or percepts. However, the TD3 algorithm utilizes an additional Critic network similar to the Double DQN algorithm which decreases the overestimation of the Q-function due to temporal difference learning (van Hasselt et al., 2015). Figure 2.3 demonstrates how the double Q-functions are used in the TD3 algorithm where (s, a, r) is a state, action, and reward sample.

The diagrams in Figure 2.2 and Figure 2.3 are simplified diagrams that are useful for understanding the algorithm. However, they do not include the actor and critic target networks that are copies of the main networks and are updated less frequently to stabilize learning (Achiam, 2018; Fujimoto et al., 2018; Mnih et al., 2013).

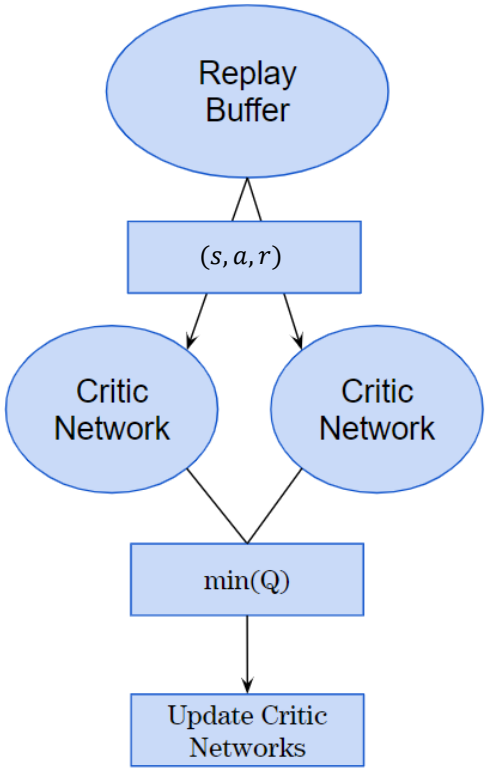


Figure 2.3 - TD3 Critic Update Diagram

The final distinction between the TD3 algorithm and its predecessors such as the Double DQN and Deep Deterministic Policy Gradient Algorithm is the target policy smoothing trick. Gaussian noise is added to the actions the target actor produces as shown in Figure 2.2. This random noise prevents the actor network from exploiting errors in the Q-function (Achiam, 2018; Fujimoto et al., 2018). An in-depth explanation of how the TD3 algorithm is implemented to catch objects is given is Section 4.4.

3. Experiment Overview

Artificial intelligence has already been shown to be capable of performing complex tasks with robots such as solving a Rubix cube (OpenAI et al., 2019). The goal of this experiment is to determine if artificial intelligent robots can be used to perform complex tasks in space. Background research has shown that deep reinforcement learning and the Twin Delayed Deep Deterministic Policy Gradients Algorithm (TD3) may be capable of controlling a robotic manipulator to perform a maneuver that requires real time adaptation. This work investigates methods that allows the real time adaptation to occur safely while the robotic arm is fixed to a rocket floating in space. In addition, deep reinforcement learning requires a large amount of training before the agent can perform a task sufficiently. Since training the agent in space would require the robotic arm to move around randomly and unpredictably, training the agent in space is impractical and unsafe. Therefore, the agent would need to train in an environment that closely simulates the environment it will experience in space. Training in a real-world Earth environment is difficult because a large amount of additional work is required to setup a training environment and this environment will not be able to simulate zero-gravity that will be experience in space.

(OpenAI et al., 2019) train their robotic hand in simulation and it is able to perform in the real-world due to a recurrent neural network in the agent's architecture allowing it to meta-learn. This work seeks to demonstrate a model-agnostic meta-learning method may allow a robotic agent to quickly learn to adapt to the environment of space without modifying the underlying neural network architecture (Finn et al., 2017; Rajeswaran et al., 2019).

The experiment to evaluate the artificial intelligent robotic agent's will be performed in space on-board a Terrier Improved Malemute Sounding Rocket launched from NASA's Wallops Flight Facility in August 2022. This experiment will include four balls that will be thrown and caught between two robotic arms while at altitudes between 100 and 150 km above the Earth. The artificial intelligent agent will utilize the iMAML algorithm to learn how to adapt to the space environment after each catch attempt. The TD3 algorithm has already been proven to catch balls in a simulated zero-gravity environment (see Section 4.4), but this experiment will determine if meta-learning can allow robots to adapt to the real-world zero-gravity environment while demonstrating how this can be performed safely.

4. Artificial Intelligence Subsystem

4.1. Subsystem Overview

The artificial intelligence subsystem is responsible for determining the position of the ball, predicting the balls position at a set time in the future, determining the ideal grasp configuration to catch the ball using a deep reinforcement learning algorithm and then learning how to perform the catch better at the level of gravity currently being experienced. Figure 4.1 shows a block diagram of the subsystem.

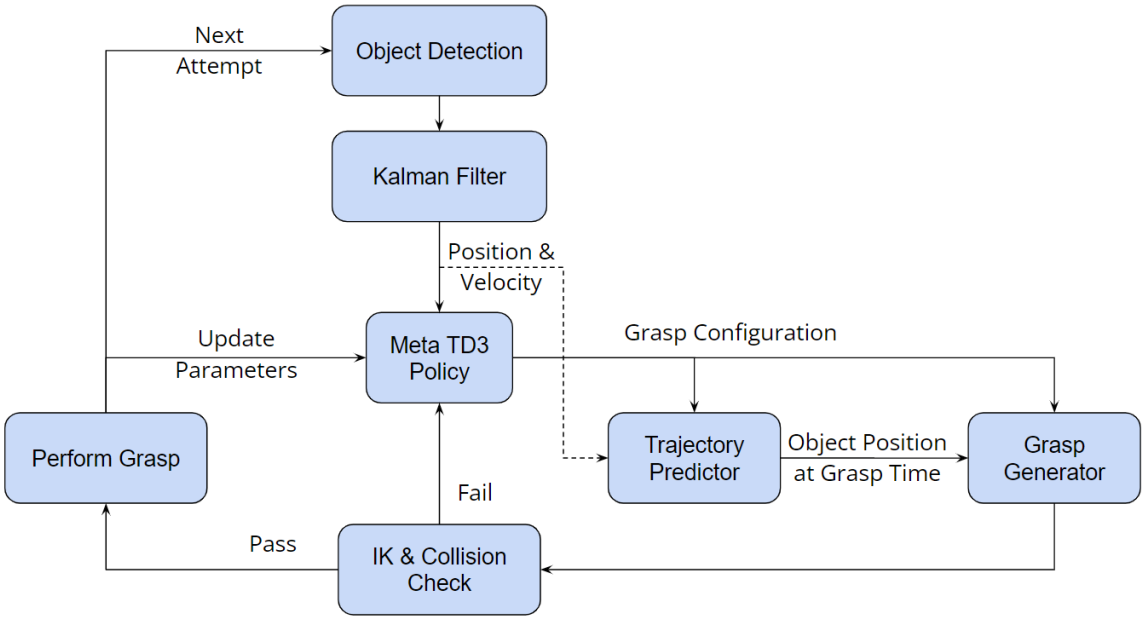


Figure 4.1 - AI Subsystem Flow Diagram

The loop begins by detecting the position of the object using computer vision. The object detection system is discussed in Section 4.2. After the position of the object has been determined, a Kalman filter determines the position and velocity of the object with noisy measurements from the object detection system. The Kalman filter is discussed in more detail in Section 4.3. The position and velocity are used as the observation for the TD3 deep reinforcement learning algorithm. The TD3 algorithm outputs the grasp configuration as the action.

The grasp configuration includes the grasp approach vector, trajectory slice, and grasp time offset. The grasp approach vector is comprised of spherical coordinates relative to the center of the ball. The gripper will move along this vector to catch the ball. The trajectory slice $x_n \in (0, 1)$ corresponds to a point along the section of the object's trajectory that intersects with the robotic arm's workspace (Discussed in Section 4.4). Finally, the grasp time offset is an offset from the predicted time the object will arrive at the trajectory slice. This offset allows for the grasp vector to originate at a point slightly ahead or behind the location of the object (Discussed in Section 4.4).

The trajectory predictor converts the normalized trajectory slice into the position and time the ball is at the agent's chosen position as discussed in Section 4.5. Once the future position of the ball is determined, the grasp generator converts the spherical coordinates relative to the object into positions relative to the base of the catching arm (Discussed in Section 4.6). Next, inverse kinematics and motion planning are performed by the ROS MoveIt library to determine the trajectory of each joint required to attempt the catch. A collision check is also performed to ensure the robotic arm does

not collide with any part of the rocket. If the motion planning or collision checking fail, the TD3 policy is queried with an updated object position so it will generate a new grasp configuration. Once a successful motion plan has been determined, the grasp is attempted, and the performance is recorded. In between attempts while the robotic arms are reloading, the Meta TD3 policy adjusts its parameters to quickly adapt and perform a better grasp on the next attempt as discussed in Section 4.7.

4.2. Object Detection

The object detection system is responsible for determining the position of the ball relative to the depth camera. The depth camera is an Intel RealSense D435 that features stereo imagers, and infrared projector, and an RGB Module. The current object detection method uses a color filter that generates a mask of the object. Since the ball is red, it is easy to differentiate the ball from the blue, green, and brown Earth or the blackness of space that will be present in the background. In addition, the ball is lit up by a light emitting diode (LED) on the payload to ensure consistent lighting. The object detection system can detect a moving ball even in front of a background of a similar color as demonstrated in Figure 4.2.

After the position of the ball in an RGB image is determined, the depth of the object is determined using the pixel to point function provided by the Intel RealSense software development kit. Once the position of the ball is determined relative to the camera, ROS parameters are used to transform that position to a point relative to the desired origin.

Figure 4.2 - Object Detection with Similar Background

4.3. Kalman Filter

The Kalman Filter is a method of estimating states and future states from inaccurate and uncertain measurements (Becker, 2022; Kalman, R. E., 1960). The filter takes the measurements and then predicts the next position and velocity. This prediction is then compared to the next measured state. This allows it to learn how much to trust the measurements compared to the underlying kinematic equation. Marsha's Kalman filter takes measurements until the state of the ball is known to a predetermined precision. This precision can be adjusted with a hyperparameter that adjusts the tradeoff between precision and how quickly the position and velocity of the object is determined.

A Multidimensional Kalman Filter is used to estimate the position and velocity of the object despite being given only a few noisy measurements. The Kalman Filter

iteratively updates the estimate of the ball's position (x_t, y_t, z_t), velocity $(\dot{x}, \dot{y}, \dot{z})$ and the uncertainty of these values (the covariance matrix, P_t) then predicts what these values will be in the next time step; Where t is incremented after each time step. The filter is initialized with the position of the gripper when the ball is released and a velocity of 0.75 inches per second toward the catching arm as due to constraints from NASA that is the desired release velocity. Once the ball is released, the filter begins polling the object detection's measured position until the uncertainty of the position is less than half of the width of the gripper. This ensures the predicted position is precise enough to fall withing the grasp range of the gripper. Marsha's Kalman Filter is largely based off the tutorial provided by (Becker, 2022) who includes the block diagram shown in Figure 4.3 that demonstrates this process.

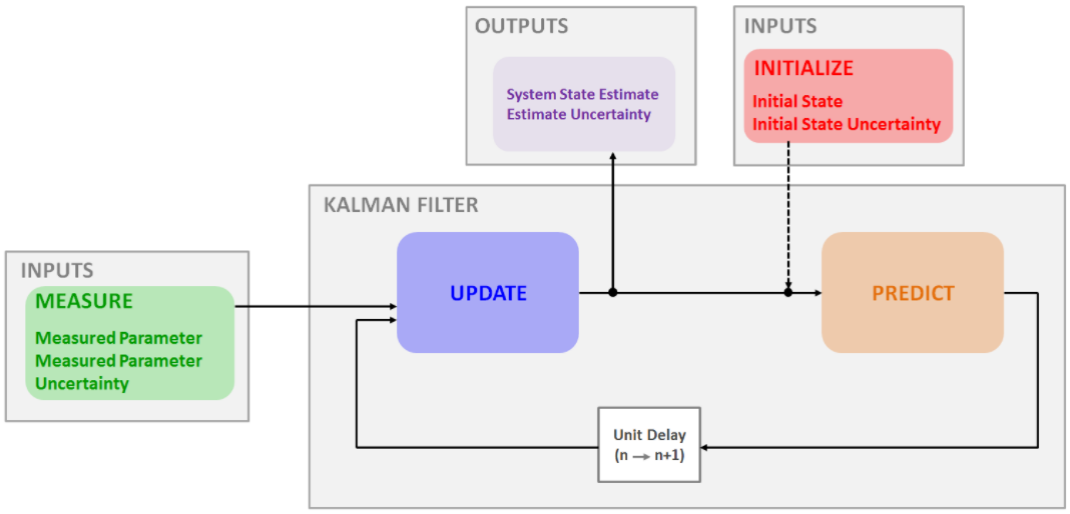


Figure 4.3 - Kalman Filter Block Diagram

The Kalman Filter designed for Marsha uses vector $\overrightarrow{z_t}$ (Equation 4.1) as the measured value and Rn as the uncertainty or covariance of the position measurement which will be found experimentally after the object detection neural network is trained. The process is begun with initial estimates of the position and velocity state $\widehat{x_t}$ (Equation 4.2) and uncertainty matrix P_t.

$$\overrightarrow{z_t} = \begin{bmatrix} x_t \\ y_t \\ z_t \end{bmatrix} \tag{4.1}$$

$$\widehat{x_t} = \begin{bmatrix} x_t \\ y_t \\ z_t \\ \dot{x}_t \\ \dot{y}_t \\ \dot{z}_t \end{bmatrix} \tag{4.2}$$

The next state is predicted with Equation 4.3.

$$\hat{x}_{t+1} = F\widehat{x_t} + G\overrightarrow{u_t} \tag{4.3}$$

Where F is a (6×6) state transition matrix as shown in Equation 4.4 that provides a more efficient way to simultaneously calculate the position and velocity along each axis. It is derived from the kinematic equations for the displacement of a moving object and the velocity of a moving object with constant acceleration as shown in Equations 4.5 and 5.6 respectively.

$$F = \begin{bmatrix} 1 & 0 & 0 & \Delta t & 0 & 0 \\ 0 & 1 & 0 & 0 & \Delta t & 0 \\ 0 & 0 & 1 & 0 & 0 & \Delta t \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.4}$$

$$\overrightarrow{x_t} = \overrightarrow{x_0} + \vec{v}_0\Delta t + \frac{1}{2}\vec{a}\Delta t^2 \tag{4.5}$$

$$\vec{v}_t = \vec{v}_0 + \vec{a}\Delta t \tag{4.6}$$

Continuing with Equation 4.3, $\overrightarrow{u_t}$ is the input variable which in this case is the acceleration as measured by an accelerometer. The acceleration input variable allows for predicting the trajectory in gravity and if the rocket still has some angular velocity. $\overrightarrow{u_t}$ is shown in Equation 4.7 and G, the control matrix responsible for calculating the change in position due to acceleration is shown in Equation 4.8.

$$\overrightarrow{u_t} = \begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} \tag{4.7}$$

$$G = \begin{bmatrix} \frac{1}{2}\nabla t^2 & 0 & 0 \\ 0 & \frac{1}{2}\Delta t^2 & 0 \\ 0 & 0 & \frac{1}{2}\Delta t^2 \\ \Delta t & 0 & 0 \\ 0 & \Delta t & 0 \\ 0 & 0 & \Delta t \end{bmatrix} \tag{4.8}$$

While the next state is being predicted, the uncertainty of the next state can be predicted with Equation 4.9.

$$P_{t+1} = FP_tF^T + G\sigma_a^2G^T \tag{4.9}$$

Where P_t and P_{t+1} are the uncertainties of the current and next state respectively; F is the state transition matrix shown in Equation 4.4; G is the control matrix shown in Equation 4.8; σ_a^2 is the process noise variance (the variance of the acceleration measurements); finally, F^T and G^T represent the transpose of the original matrix. The total predicted uncertainty is provided by the sum of the state estimate covariance (FP_tF^T) and the process noise covariance ($G\sigma_a^2G^T$).

After predicting the next state, the estimate of the current state is updated. The update begins by measuring the state with the object detection system that gives state

measurement $\overrightarrow{z_t}$ shown in Equation 4.1. Next the Kalman gain is updated which is a (6×3) matrix calculated with Equation 4.10 that seeks to minimize the estimated variance. The Kalman gain equation derivation is given by (Becker, 2022).

$$K_t = P_{t-1}H^T(HP_{t-1}H^T + R)^{-1} \tag{4.10}$$

Where H is a (6×3) matrix given in Equation 4.11 that relates the position measurement $\vec{z}_t$ to the state estimate $\widehat{x_t}$ which contains the estimated position and velocity; R is a (3×3) matrix of the measurement uncertainty. R is shown in Equation 4.12 but is simply the variance σ_m^2 of the object detection system's measurement for each axis which can be found experimentally.

$$H = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} \tag{4.11}$$

$$R = \begin{bmatrix} \sigma_m^2 & 0 & 0 \\ 0 & \sigma_m^2 & 0 \\ 0 & 0 & \sigma_m^2 \end{bmatrix} \tag{4.12}$$

After the Kalman gain K_t is found, the state estimate is updated with Equation 4.13.

$$\widehat{x_t} = \hat{x}_{t-1} + K_t(\overrightarrow{z_t} - H\hat{x}_{t-1}) \tag{4.13}$$

Finally, the uncertainty estimate P_t is updated with Equation 4.14 and the process begins again by using the estimated state $\hat{x}_t$ and uncertainty P_t to predict the next state $\hat{x}_{t+1}$ and uncertainty P_{t+1} as given in Equations 4.3 and 4.9 respectively.

4.4. Catching Agent

Open AI developed an interface for reinforcement learning algorithms called OpenAI Gym (Brockman et al., 2016). A custom gym environment was developed for the task of catching the ball. This custom gym environment is an interface that translates between the artificial intelligent agent and the ROS system which collects the percepts and performs the actions for the agent. OpenAI Gym is an open-source library that has become standard for AI researchers as it is easy to test different reinforcement learning algorithms on the same environment.

The Stable-Baselines3 implementation of the Twin Delayed Deep Deterministic Policy Gradients algorithm (TD3) is used to train the catching agent (Raffin et al., 2021). The catching agent collects the position and velocity percepts from the object detection system and then outputs the action tuple as shown in Figure 4.4.

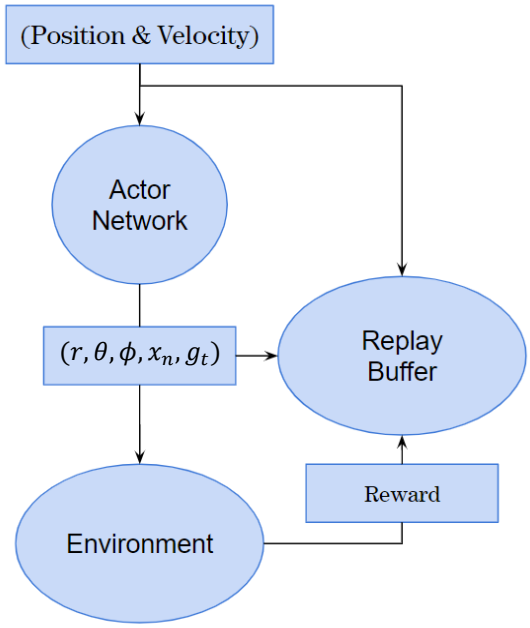


Figure 4.4 - Catching Agent Exploration Diagram

The action tuple is composed of spherical coordinates (r, θ, ϕ) whose origin is located at the center of the ball. The grasp generator system then converts the spherical coordinates to pre-grasp and post-grasp positions relative to the base of the robotic arm so MoveIt's inverse kinematics library can determine the motion path for each joint. The grasp generator system is discussed further in Section 4.6.

The next index in the action tuple is trajectory slice $x_n \in (0, 1)$ which corresponds to a point along the section of the object's trajectory that intersects with the robotic arm's workspace (Discussed in Section 4.5). Finally, the grasp time offset is an offset from the predicted time the object will arrive at the trajectory slice. This offset allows the for the grasp vector to originate at a point slightly ahead or behind the location of the object.

Reinforcement learning uses a reward function to give the agent feedback on which actions performed well and should be repeated for the percepts that it received. The reward function that produced the best results for Marsha is given in Equation 4.14.

$$r = \begin{cases} -4, & if\ pre\ grasp\ planning\ fails \\ -2, & if\ grasp\ planning\ fails \\ -1 * d_{grasp}, & if\ catch\ unsuccessful \\ +10, & if\ catch\ successful \end{cases} \tag{4.14}$$

The reward is determined by the state of the environment after the catch is attempted. In the first state, a reward of -4 is produced if pre-grasp planning fails. This negative reward punishes the agent for attempting a pre-grasp position that is not kinematically possible. In the second state, a reward of -2 is produced if the grasp planning fails. This is produced if the MoveIT rapidly-exploring random tree (RRT) motion planning

algorithm can not find a path from the pre-grasp position to the grasp position. This could be because there is an unavoidable obstacle in the way, or the grasp position is kinematically impossible. A negative reward is given so the agent learns to avoid positions that result in failed motion planning. If motion planning fails, the agent is queried again with an updated position in an attempt to find an action tuple that results in successful motion planning. Figure 4.5 shows the graph of a training session that demonstrated the agent's ability to learn how to decrease the number of motion planning attempts.

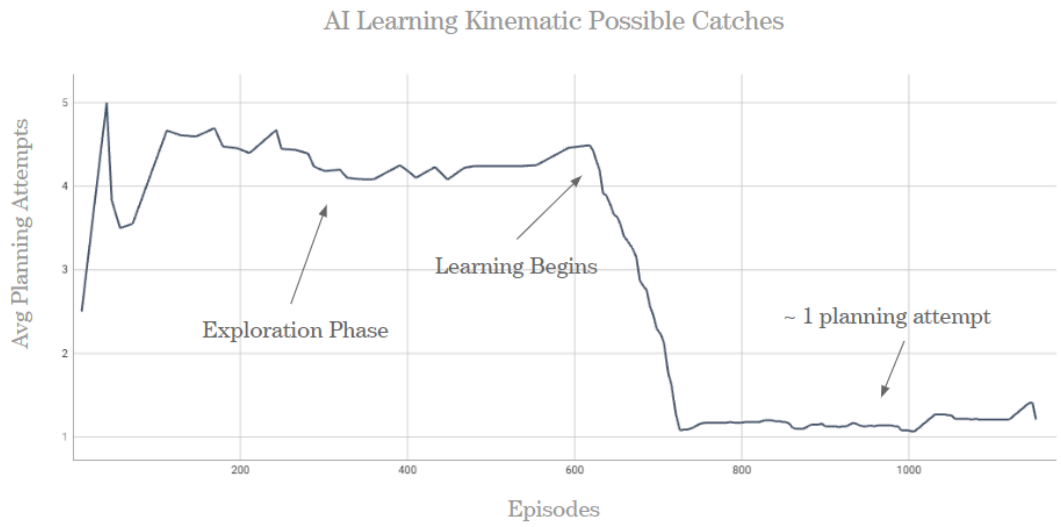


Figure 4.5 - AI Learning Kinematic Possible Catches

In Figure 4.5, actions are sampled from a uniform random distribution for the first six hundred episodes in order to improve exploration (Achiam, 2018; Raffin et al., 2021) (Figure 4.5 is smoothed to improve readability). After episode seven hundred, the agent is able to return actions that result in kinematically possible grasps in almost one planning attempt each time. Each episode corresponds to a single ball throw. The episode ends if motion planning is successful, the ball moves out of reach (occurs after

approximately five planning attempts), or twenty failed motion planning attempts have occurred.

Continuing with Equation 4.14, if motion planning is successful, but the gripper does not close around the ball, a reward is produced that is negatively proportional to the distance between the object and the end effector at the time the gripper closes. The final case occurs if everything goes well and the catch is successful. A successful catch results in a large positive reward of +10. The catch AI has shown promising results as Figure 4.6 demonstrates.

Figure 4.6 - Average Reward During Training Session; Light gray displays the exact reward received during training (10 represents a catch); Dark gray displays a moving average for the reward indicating an increase in catch frequency as training continues.

4.5. Trajectory Prediction

The trajectory prediction system is responsible for determining the position and time of the object at the agent's desired trajectory slice. The trajectory slice $x_n \in (0, 1)$ is centered around the base of the robotic arm so $x_n = 0.5$ is the point along the trajectory that is closest to the base of the robotic arm. When $x_n = 0$ or 1 the agent will attempt the grasp at a point along the trajectory at the end of the reach of the robotic arm. This ensures any trajectory slice the agent selects will be within the workspace of the robotic manipulator.

Multiple trajectory prediction methods were developed for Marsha, but the best performing one was named the second order distance minimization with binary search method as it is efficient and will work with or without gravity. The goal of this method is to denormalize the trajectory slice x_n that is given by the TD3 agent. Therefore, the time the object is at $x_n = 0, x_n = 0.5$, and $x_n = 1$ must be found which is demonstrated in Figure 4.7.

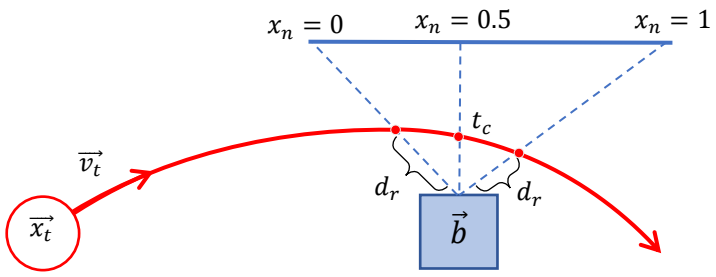


Figure 4.7 - Trajectory Slice Denormalization

The first step is finding the time the object is at the point on its trajectory closest to the base of the robotic arm as denoted by t_c in Figure 4.7. Since the object follows the kinematic equation for the displacement of a moving object as given in Equation 4.5,

the distance between the ball and the base of the robotic arm at a given time can be calculated with Equation 4.15. Equation 4.15. assumes the values of $\overrightarrow{x_0}$, $\overrightarrow{v_0}$, and $\vec{a}$ have been found with the Kalman filter and the base of the robotic arm is at the origin. Equation 4.16 references the individual indexes of the vectors used in Equation 4.5 which are given in Equation 4.15.

$$\overrightarrow{x_t} = \overrightarrow{x_0} + \overrightarrow{v_0}\Delta t + \frac{1}{2}\vec{a}\Delta t^2 \tag{4.5}$$

$$\overrightarrow{x_0} = \begin{bmatrix} x_0 \\ y_0 \\ z_0 \end{bmatrix} \quad \overrightarrow{v_0} = \begin{bmatrix} \dot{x}_0 \\ \dot{y}_0 \\ \dot{z}_0 \end{bmatrix} \quad \vec{a} = \begin{bmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{bmatrix} \tag{4.15}$$

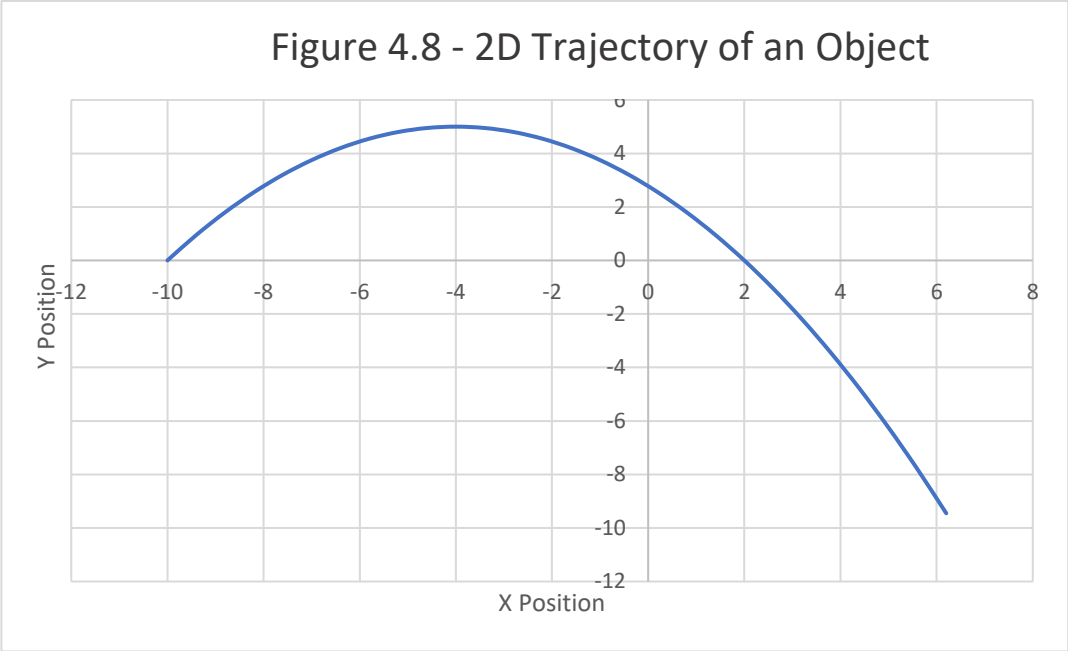
$$d(t) = \sqrt{(x_0 + \dot{x}_0\Delta t + \ddot{x}\Delta t^2)^2 + (y_0 + \dot{y}_0\Delta t + \ddot{y}\Delta t^2)^2 + (z_0 + \dot{z}_0\Delta t + \ddot{z}\Delta t^2)^2} \tag{4.16}$$

To find the time t_c when the ball is closest to the base of the robotic arm, the function $d(t)$ would need to be minimized. The minimum could be found by finding the time when $d'(t) = 0$ and $d''(t) > 0$. Since the derivative could be calculated beforehand, the program would only need to find the roots for a third and fourth order equation with known coefficients. While this could be possible by finding the approximate solution with the Newton-Raphson method, a simpler approach can be used (Raphson, 1697; Smit, 2021).

The simpler approach stems from the binary search algorithm commonly used to search for a value in a sorted array (Lin, 2019). While, the binary search algorithm is normally used for a discrete set of values, a variant of the algorithm can be used to approximate the second order d(t) function. This variant includes recursively finding the time the object is closest to the base of the arm with a given range and then reducing the range. The algorithm is shown below in c++:

```
typedef Eigen::Matrix<float, 3, 1> Vector3f;

 float distance(float t) {
  Vector3f obj_pos = x_0 + v_0*t + 0.5*a*t*t;
  Vector3f robot_base(0, 0, 0);
  // x_n = 0 and x_n = 1 can be found by setting desired_dist
  // to the reach of the arm
  return eigen_dist(robot_base, obj_pos) - desired_dist;
}

float binarySearch(float before_time, float after_time) {

  float mid_time = before_time + (after_time - before_time) / 2;
  float mid_dist = distance(mid_time);
  float after_dist = distance(after_time);
  float before_dist = distance(before_time);

  // if not much change occurs local minimum has been reached
  if (abs(after_dist - before_dist) < EPSILON) {
    return mid_time;
  }
  if (mid_dist < before_dist and mid_dist < after_dist) {
    if (before_dist < after_dist) {
      return binarySearch(before_time, (mid_time + after_time) / 2);
    }
    else {
      return binarySearch((mid_time + before_time) / 2, after_time);
    }
  }

  // mid_dist is also less than before_dist
  if (mid_dist > after_dist) {
    return binarySearch((mid_time + before_time) / 2, after_time);
  }

  if (mid_dist > before_dist) {
    return binarySearch(before_time, (mid_time + after_time) / 2);
  }
 }
```

To evaluate the validity of the algorithm, multiple unit tests were performed as well as an experiment in excel. Figures 4.8 and 4.9 demonstrate a 2D version of the algorithm.



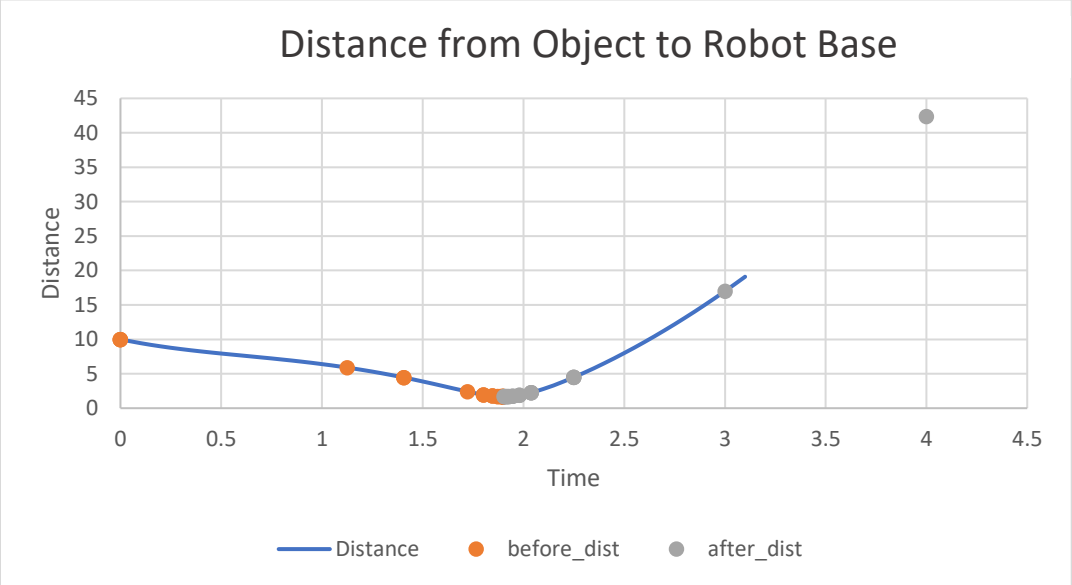


Figure 4.8 shows the trajectory of an object that follows Equations 4.16 and 4.17.

$$x = x_0 + v_x * t \qquad x_0 = -10, \quad v_x = 6 \tag{4.16}$$

$$y = y_0 + v_y * t + a_y * t^2 \qquad y_0 = 0, \quad v_y = 10, \; a_y = -10 \tag{4.17}$$

The distance between the object and the base of the arm (assuming the base is at the origin) was calculated with Equation 4.15, but with the z axis excluded for simplicity. As Figure 4.9 shows, the orange and grey dots represent the binarySearch algorithm quickly converging on the minimum value.

4.6. Grasp Generator

The 6-DOF GraspNet (Mousavian et al., 2019) is a method to generate a grasp from a 3D point cloud of a non-uniform shaped object with a variational autoencoder. MARSHA's grasp generator is designed to be easily replaced by the 6-DOF GraspNet therefore enabling MARSHA to catch objects of varying shape (assuming the trajectory is properly learned or calculated). The grasp generator emulates the decoder by using spherical coordinates (r, θ, Φ) as the "latent space." Instead of the grasp evaluator iterating through generated grasps, MARSHA's grasp evaluator (aka the actor network) uses the latent space as actions that then generate the grasp. The grasp generator transforms these spherical coordinates to a quaternion that rotates the robot's orientation. The following algorithm performs the conversion:

```
geometry_msgs::Pose generate(float normalized_r, float theta, float phi)
{
  geometry_msgs::Pose pose;
  float radius = convert_radius(normalized_r);
  tf::Vector3 wrist_vector = polar_to_rect(radius, theta, phi);
  wrist_vector.normalize();

  tf::Vector3 z = tf::Vector3(0, 0, 1);


  tf::Vector3 rot_axis = wrist_vector.cross(z);
  float rot_angle = acos(wrist_vector.dot(z));
  tf::Vector3 other_axis = rot_axis.cross(z);

  float dot_product = other_axis.dot(wrist_vector);
  if (dot_product < 0) {
    rot_angle = M_PI - rot_angle;
  }

  return tf::Quaternion(rot_axis.normalized(), rot_angle);
}
```

4.7. Meta TD3 Algorithm

The artificial intelligence responsible for the catch is powered by an algorithm that combines the TD3 and iMAML algorithms. The individual algorithms are discussed in Sections 2.5 and 2.6 respectively. While OpenAI's Rubix cube solver (OpenAI et al., 2019) implements meta learning with a long short-term memory architecture (LSTM), this work uses a model-agnostic meta-learning (MAML) approach as it can be implemented with any model or algorithm (Finn et al., 2017). However, MAML is inefficient because it includes taking the gradient of the gradient descent algorithm, so MAML's successor, model-agnostic meta-learning with implicit gradients (iMAML) was implemented for Marsha (Rajeswaran et al., 2019).

Both MAML and iMAML algorithm learn a meta-model θ that is used to initialize multiple mesa-models ϕ_i. The mesa-models are initialized with the same weights as the meta-model, but then retrained to perform better on a specific task. After the mesa models have been trained the meta-model is updated using the results from each training session. This is repeated for multiple tasks or environments until the meta-model learns parameters that can be trained with a few shots to succeed in an unseen task or environment.

For Marsha, the Stable-Baselines3 implementation of the TD3 algorithm is used to train the mesa-models. The iMAML algorithm was implemented in python to train the meta-model. The iMAML algorithm is shown in Algorithm 4.9.

Algorithm 1 Implicit Model-Agnostic Meta-Learning (iMAML)

Require: Distribution over tasks $P(\mathcal{T})$, outer step size η, regularization strength λ,
while not converged **do**
  Sample mini-batch of tasks $\{\mathcal{T}_i\}_{i=1}^{B} \sim P(\mathcal{T})$
  for Each task $\mathcal{T}_i$ **do**
    Compute task meta-gradient $\boldsymbol{g}_i = \texttt{Implicit-Meta-Gradient}(\mathcal{T}_i, \boldsymbol{\theta}, \lambda)$
  end for
  Average above gradients to get $\hat{\nabla} F(\boldsymbol{\theta}) = (1/B) \sum_{i=1}^{B} \boldsymbol{g}_i$
  Update meta-parameters with gradient descent: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \hat{\nabla} F(\boldsymbol{\theta})$ *// (or Adam)*
end while

Algorithm 2 Implicit Meta-Gradient Computation

Input: Task $\mathcal{T}_i$, meta-parameters $\boldsymbol{\theta}$, regularization strength λ
Hyperparameters: Optimization accuracy thresholds δ and δ'
Obtain task parameters ϕ_i using iterative optimization solver such that: $\|\phi_i - \mathcal{A}lg_i^{\star}(\boldsymbol{\theta})\| \leq \delta$
Compute partial outer-level gradient $\boldsymbol{v}_i = \nabla_{\boldsymbol{\phi}} \mathcal{L}_{\mathcal{T}}(\boldsymbol{\phi}_i)$
Use an iterative solver (e.g. CG) along with reverse mode differentiation (to compute Hessian vector products) to compute $\boldsymbol{g}_i$ such that: $\|\boldsymbol{g}_i - \left(\boldsymbol{I} + \frac{1}{\lambda} \nabla^2 \hat{\mathcal{L}}_i(\boldsymbol{\phi}_i)\right)^{-1} \boldsymbol{v}_i\| \leq \delta'$
Return: $\boldsymbol{g}_i$

Algorithm 4.9 - iMAML Algorithm (Rajeswaran et al., 2019)

The outer step size η is the gradient descent step size for the meta-model; The regularization strength λ is a hyperparameter that represents how close the mesa-parameters will be to the meta-parameters; B represents the number of tasks; T_i

represents a single task; θ represents the meta-parameters; $\widehat{\nabla}F(\theta)$ is the gradient the will produce the updated meta-parameters when applied with gradient descent; The optimization accuracy thresholds δ and δ' represent how close the mesa-parameters and meta-parameters should get to the optimal solution; $\mathcal{L}_{\mathrm{T}}(\phi_i)$ represents the test loss from mesa-training; $\boldsymbol{I}$ is the identity matrix; and $\hat{\mathcal{L}}_{\mathrm{T}}(\phi_i)$ represents the training loss from mesa-training.

To use iMAML with the TD3 algorithm, the three TD3 networks (actor and two critics) need to be meta-learned. However, each network has a different way of calculating the loss functions. The actor network loss is calculated with Equation 4.18 (Fujimoto et al., 2018).

$$\mathcal{L}_{actor} = -1 * avg(Q_{critic1}, Q_{critic2}) \tag{4.18}$$

Where the Q values are produced by the main critic networks not the target networks. The loss for the critic networks is calculated with Equation 4.19 (Fujimoto et al., 2018).

$$\mathcal{L}_{critic} = MSE\left(Q1_{target}, Q1_{main}\right) + MSE(Q2_{target}, Q2_{main}) \tag{4.19}$$

Where MSE is the mean squared error, $Q1$ is the Q-value produced by the first TD3 critics, and $Q2$ is the Q-value produced by the second TD3 critic. Figure 4.10 gives a visual for how the losses relate to the different networks.

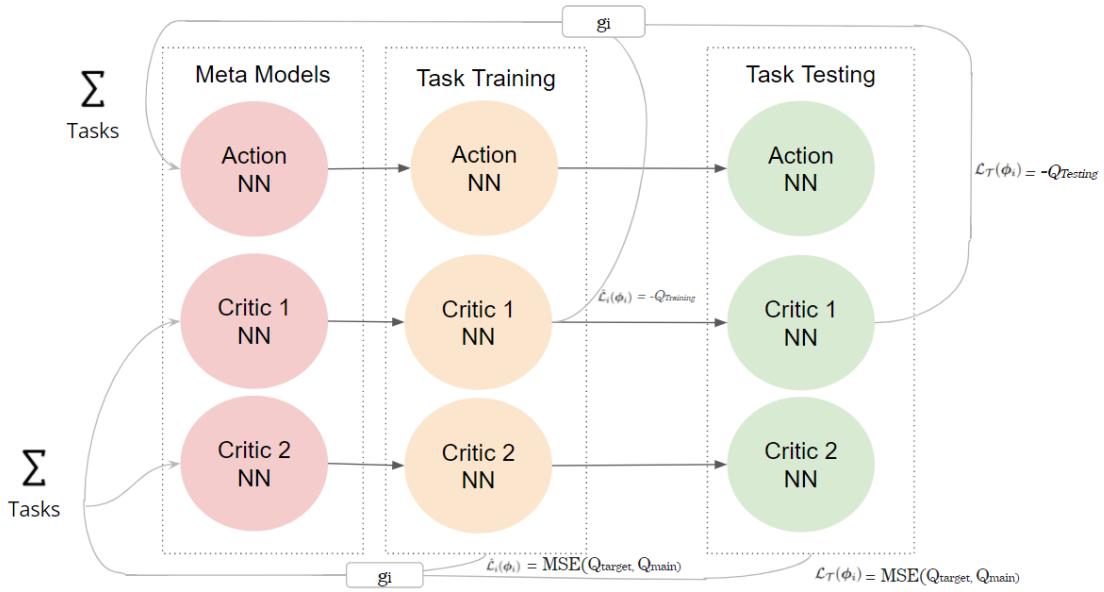


Figure 4.10 - Meta-TD3 Training

The appendix shows the entire python file that was written for Marsha that combines the iMAML and TD3 algorithms.

5. Discussion

This work showed that the TD3 algorithm could successfully control a robotic arm to catch a ball in a simulated zero-gravity environment and learn kinematically possible grasp configurations. Current tests resulted in a successful catch occurring 69% of the time after training for one thousand episodes. However, there was little work put into adjusting hyperparameters to receive a better catch rate. This was a large project so other sections of the project required immediate attention. It is highly likely this success rate can be improved.

This work also produced the second order distance minimization with binary search algorithm which quickly calculates the minimum distance between a moving object

and a point in space. The Meta-TD3 algorithm was also developed and to my knowledge has not been done before. The effectiveness of the Meta-TD3 algorithm will continue to be tested over the summer of 2022 and a final test will be conducted from 150km above the Earth on-board a Terrier Improved Malemute Rocket.

For the artificial intelligence system, I developed a custom OpenAI gym environment to interface the agent with the rest of the Marsha system. OpenAI gym is the standard for AI researchers to develop reinforcement learning algorithms that interact with different environments (Brockman et al., 2016). I developed the Marsha robotic subsystem so it can easily be adapted to any robot (as demonstrated by implementing it on the AR3 and two arm2d2 robots). This means AI researchers experienced with the gym interface would not need to have experience with ROS to test new reinforcement learning algorithms on robots that have the Marsha system installed. On the other hand, ROS engineers would not need to have experience with AI to test new control methods for an AI controlled robotic system. Currently, I have integrated Marsha on the AR3 that Chris Annin donated to Northwest Nazarene University and the dual Arm2d2 arms that have been developed by the Rocksat team. However, it may be difficult for Northwest Nazarene University to continue the work I began researching adaptive Astro-robotics. It is currently difficult for students at NNU to gain the knowledge I was able to learn to complete such a project. While I attempted to create a level of abstraction so computer science and engineering students could collaborate on the software without needing to learn the other discipline, the two disciplines may have become too intertwined in the modern age to work in similar

fields without knowledge in both disciplines. If NNU wishes to continue researching innovative technology in adaptive astro-robotics and produce students who excel in similar fields, a better path must be created for students to gain knowledge in both disciplines. In addition, it may be beneficial to develop curriculum for advanced artificial intelligence techniques such as deep reinforcement learning. Although there is not a standard textbook on the subject, the OpenAI Spinning Up repository is a great resource (Achiam, 2018).

Future versions of Marsha should implement the following features that will allow intelligent astro-robotic arms to catch non-uniform shaped objects such as tools, components, or spacecraft. The first feature that should be implemented is the 6-DOF grasp net (Mousavian et al., 2019). The TD3 catch agent's action should be the latent space for the grasp net's decoder. Therefore, the catch agent will learn to grasp different points on the object. A different gripper will also be required to perform this. Two-fingered grippers are commonly used to perform such grasps, however it may be difficult to get the precision required to grasp a moving object in space with a two-fingered gripper. Perhaps a soft robotic gripper could be used to grasp moving objects with less precision. The soft robotic gripper could also feature AI for soft robotic sensing or actuation (Kim et al., 2021). The final addition required for Marsha to catch non-uniform shaped objects, is an updated trajectory predictor system. The current trajectory predictor system assumes the object is a point, however non-uniform shaped objects can rotate. The Kalman filter and trajectory slice denormalizers will still be useful, but an additional system will need to predict how a specific point on the object

will move as it rotates through space. Perhaps the PointNet++ architecture could be combined with a recurrent neural network to predict the future rotation of a 3D-point cloud object. Meta-learning will also play an important role in learning how different objects move and react to touch in zero-gravity.

Implementing iMAML was a difficult task as the MAML paper provided instructions for reinforcement learning, but the iMAML paper did not. I had to come up with a method to incorporate the iMAML paper with reinforcement learning and the TD3 algorithm, which to my knowledge has not been done before. Additionally, the Stable-baselines3 implemented TD3 with PyTorch, but I only had experience with Tensorflow at the time I worked on the Meta-TD3 algorithm. I was able to convert between PyTorch and Tensorflow, but it would be much better to re-write the Meta-TD3 algorithm in PyTorch.

I also believe that future attempts at creating safe AI controlled Astro-robots should use the TD3 algorithm or similar algorithms. The essential part of the TD3 algorithm is its ability to be deterministic. Therefore, the AI agent will produce the same action every time it receives the same perception. This allows the agent to be properly tested before it performs in a safety-critical application. In addition, I found that using AI to control the robot in a high level was important. Using AI to determine how the agent grasped the ball performed much better than creating an agent that is responsible for controlling the individual joints or the position of the end effector. This allowed for inverse kinematics and collision checking to still be used.

References

Achiam, J. (2018). *Spinning Up in Deep Reinforcement Learning*. https://spinningup.openai.com/en/latest/index.html

Barto, A. G. (2007). Temporal difference learning. *Scholarpedia*, *2*(11), 1604. https://doi.org/10.4249/scholarpedia.1604

Becker, A. (2022). *Online Kalman Filter Tutorial*. https://www.kalmanfilter.net/

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., & Zaremba, W. (2016). OpenAI Gym. *ArXiv:1606.01540 [Cs]*. http://arxiv.org/abs/1606.01540

Cutler, M., & How, J. P. (2015). Efficient reinforcement learning for robots using informative simulated priors. *2015 IEEE International Conference on Robotics and Automation (ICRA)*, 2605–2612. https://doi.org/10.1109/ICRA.2015.7139550

Finn, C., Abbeel, P., & Levine, S. (2017). Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks. *ArXiv:1703.03400 [Cs]*. http://arxiv.org/abs/1703.03400

Fujimoto, S., van Hoof, H., & Meger, D. (2018). Addressing Function Approximation Error in Actor-Critic Methods. *ArXiv:1802.09477 [Cs, Stat]*. http://arxiv.org/abs/1802.09477

Gao, Y., Tebbe, J., & Zell, A. (2021). Optimal Stroke Learning with Policy Gradient Approach for Robotic Table Tennis. *ArXiv:2109.03100 [Cs]*. http://arxiv.org/abs/2109.03100

Kalman, R. E. (1960). *A new approach to linear filtering and prediction problems.*

Kim, D., Kim, S.-H., Kim, T., Kang, B. B., Lee, M., Park, W., Ku, S., Kim, D., Kwon, J.,

Lee, H., Bae, J., Park, Y.-L., Cho, K.-J., & Jo, S. (2021). Review of machine learning methods in soft robotics. *PLOS ONE, 16*(2), e0246102. https://doi.org/10.1371/journal.pone.0246102

Koos, S., Mouret, J.-B., & Doncieux, S. (2010). Crossing the reality gap in evolutionary robotics by promoting transferable controllers. *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*, 119–126. https://doi.org/10.1145/1830483.1830505

Lin, A., & al. (2019). Binary search algorithm. *WikiJournal of Science*, *2*(1), 5. https://doi.org/10.15347/WJS/2019.005

Mahmood, A. R., Korenkevych, D., Vasan, G., Ma, W., & Bergstra, J. (2018). Benchmarking Reinforcement Learning Algorithms on Real-World Robots. *ArXiv:1809.07731 [Cs, Stat]*. http://arxiv.org/abs/1809.07731

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing Atari with Deep Reinforcement Learning. *ArXiv:1312.5602 [Cs]*. http://arxiv.org/abs/1312.5602

Mousavian, A., Eppner, C., & Fox, D. (2019). 6-DOF GraspNet: Variational Grasp Generation for Object Manipulation. *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, 2901–2910. https://doi.org/10.1109/ICCV.2019.00299

OpenAI, Akkaya, I., Andrychowicz, M., Chociej, M., Litwin, M., McGrew, B., Petron, A., Paino, A., Plappert, M., Powell, G., Ribas, R., Schneider, J., Tezak, N., Tworek, J., Welinder, P., Weng, L., Yuan, Q., Zaremba, W., & Zhang, L. (2019). Solving Rubik's Cube with a Robot Hand. *ArXiv:1910.07113 [Cs, Stat]*.

http://arxiv.org/abs/1910.07113

Raffin, A., Hill, A., Gleave, A., Kanervisto, A., Ernestus, M., & Dormann, N. (2021). *Stable-Baselines3: Reliable Reinforcement Learning Implementations*. 8.

Rajeswaran, A., Finn, C., Kakade, S., & Levine, S. (2019). Meta-Learning with Implicit Gradients. *ArXiv:1909.04630 [Cs, Math, Stat]*. http://arxiv.org/abs/1909.04630

Raphson, J. (1697). *Analysis Eequationum Universalis*. https://archive.org/details/bub_gb_4nlbAAAAQAAJ

Smit, A. (2021). *Program for Newton Raphson Method*. https://www.geeksforgeeks.org/program-for-newton-raphson-method/

van Hasselt, H., Guez, A., & Silver, D. (2015). Deep Reinforcement Learning with Double Q-learning. *ArXiv:1509.06461 [Cs]*. http://arxiv.org/abs/1509.06461

Watkins, C. J. C. H., & Dayan, P. (1992). Q-learning. *Machine Learning*, *8*(3), 279–292. https://doi.org/10.1007/BF00992698

Appendix

The code for the project is available in a public Github repository:

https://github.com/aborger/Marsha

The repository is split into two branches (Auxiliary-Platform and Embedded-Platform). The Auxiliary-Platform features programs and launch files to control the experiment from an auxiliary computer in a lab setting. The Embedded-Platform features the program and launch files that are stored on the Jetson Nanos and is used in the final flight code. Within these branches there are several packages. The following section gives an overview of the important files within the marsha-ai package.

AI_node.py

The AI node is the top level program to run the TD3 algorithm.

https://github.com/aborger/Marsha/blob/Embedded-Platform/marsha_ai/nodes/ai_node

gym_env.py

The gym environment file controls the Open AI gym environment.

https://github.com/aborger/Marsha/blob/Embedded-Platform/marsha_ai/src/marsha_ai/catch_bandit/gym_env.py

catch_interface.py

The catch interface provides an interface between the gym environment and the Move Interface that controls the robot. This interface is largely responsible for performing the catch.

https://github.com/aborger/Marsha/blob/Embedded-Platform/marsha_ai/src/marsha_ai/catch_bandit/catch_interface.py

trajectory_predictor_kalman.cpp

This trajectory predictor uses the Kalman filter and the second order distance minimization with binary search algorithm to estimate the position and velocity and predict the future position of the ball.

https://github.com/aborger/Marsha/blob/Embedded-Platform/marsha_ai/nodes/trajectory_predictor_kalman.cpp

kalman_filter.h

The Kalman filter file uses the c++ eigen library to implement a Kalman filter.

https://github.com/aborger/Marsha/blob/Embedded-Platform/marsha_ai/include/marsha_ai/kalman_filter.h

object_dynamics.h

The object dynamics file finds the position along the object's trajectory that is closest to the base of the robotic arm and the positions at the end of the reach of the arm. This file includes two methods, the Newton Raphson Method, and the second order distance minimization with binary search algorithm method that is far less complex and gave better results.

https://github.com/aborger/Marsha/blob/Embedded-Platform/marsha_ai/include/marsha_ai/object_dynamics.h

grasp_generator.cpp

The grasp generator file converts spherical coordinates from the TD3 agent into grasp positions.

https://github.com/aborger/Marsha/blob/Embedded-Platform/marsha_ai/nodes/grasp_generator.cpp

meta_td3.py

The meta TD3 file combines the iMAML and TD3 algorithms to allow the catching agent to continue learning in space.

https://github.com/aborger/Marsha/blob/Embedded-Platform/marsha_ai/src/marsha_ai/meta_td3.py