# DEEP REINFORCEMENT LEARNING FOR THE VELOCITY CONTROL OF A MAGNETIC, TETHERED DIFFERENTIAL-DRIVE ROBOT

A Dissertation
Presented to
The Academic Faculty

By

Devarsi Rawal

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in Computer Science
College of Computing

Georgia Institute of Technology

December 2022

**DEEP REINFORCEMENT LEARNING FOR THE VELOCITY CONTROL OF A MAGNETIC, TETHERED DIFFERENTIAL-DRIVE ROBOT**

Thesis committee:

Dr. Cédric Pradalier
College of Computing
School of Interactive Computing
*Georgia Institute of Technology*

Dr. Sehoon Ha
College of Computing
School of Interactive Computing
*Georgia Institute of Technology*

Dr. Mackenzie Lau
College of Engineering
School of Aerospace Engineering
*Georgia Institute of Technology*

Date approved: December 6, 2022

They know enough who know how to learn.

*Henry Adams*

To my loving parents and brother

## ACKNOWLEDGMENTS

First and foremost, I would like to express my deepest gratitude to my advisor Dr. Cédric Pradalier for providing me the opportunity to break into the robotics field through two exceptional classes and a year of engaging research. Your wisdom and expertise have been invaluable, and you truly helped me find a new passion. I would also like to thank the members of the thesis committee, Dr. Sehoon Ha and Dr. Mackenzie Lau, for lending their knowledge and taking the time to review the research done in this thesis.

I would also like to thank my graduate mentor Dr. Antoine Richard for all the help you provided over the past year. From fighting through bugs to hosting the others and I at the SpaceR lab in Luxembourg, I would not trade those experiences for anything.

To Luis Batista, Stéphanie Aravecchia, Pete Schroepfer, Salim Khazem, Mehran Adibi, Dr. Othmane Ouabi, and the other members of the DREAM lab, thank you for the wonderful conversations, advice, and support. You all made a once-in-a-lifetime experience in France even more unique.

Special thanks go out to Ithan Velarde for helping set up SEED RL to run on the lab computers. You were a great help, and it was a pleasure working with you for the semester.

Finally, I would like to thank my father, for encouraging me to do a thesis and always pushing me to strive for my best, my mother, for her unconditional love and support, and my brother, for being my best friend and a great role model. I would not be the person I am today without you and consider myself incredibly blessed.

## TABLE OF CONTENTS

## LIST OF TABLES

## LIST OF FIGURES

## LIST OF ACRONYMS

**A2C** advantage actor-critic

**A3C** asynchronous advantage actor-critic

**AC** actor-critic

**BPTT** back-propagation through time

**DDR** differential drive robot

**DL** deep learning

**DNN** deep neural network

**DOF** degree of freedom

**DQN** deep Q-network

**DRL** deep reinforcement learning

**GLIE** Greedy in Limit with Infinite Exploration

**GPI** generalized policy iteration

**ICC** instantaneous center of curvature

**IID** independent and identically distributed

**IMU** inertial measurement unit

**IVK** inverse velocity kinematics

**KL** Kullback-Leibler

**LSTM** long-short term memory

**MC** Monte Carlo

**MDP** Markov decision process

**MP** Markov process

**MRP** Markov reward process

**MSE** mean-squared error

**ONNX** Open Neural Network Exchange

**PID** proportional-integral-derivative

**PPO** proximal policy optimization

**ReLU** rectified linear unit

**RL** reinforcement learning

**RMSE** root mean-squared error

**RNN** recurrent neural networks

**ROS** Robot Operating System

**TD** temporal-difference

**TGS** Temporal Gauss-Siedel

**TRPO** trust-region policy optimization

**UAV** unmanned aerial vehicle

**UGV** unmanned ground vehicle

**URDF** Unified Robot Description Format

**UUV** unmanned underwater vehicle

**UWB** ultrawide band

## SUMMARY

The ROBOPLANET Altiscan crawler is a magnetic-wheeled, differential-drive robot being explored as an option to aid, if not completely replace, humans in the inspection and maintenance of marine vessels. Velocity control of the crawler is a crucial part in establishing trust and reliability amongst its operators. However, thanks to the crawler's elongated, magnetic wheels and umbilical tether, it operates in a complex environment rich with nonlinear dynamics which makes control challenging. Model-based approaches for the control of a robot that aim to mathematically formalize the physics of the system require an in-depth knowledge of the domain.

Reinforcement learning (RL) is a trial-and-error-based approach that can solve control problems in nonlinear systems. To accommodate for high-dimensionality and continuous state spaces, deep neural networks (DNNs) can be used as nonlinear function approximators to extend RL, creating a method known as deep reinforcement learning (DRL). DRL coupled with a simulated environment provides a way for a model to learn physics-naive control. The research conducted in this thesis explored the efficacy of a DRL algorithm, proximal policy optimization (PPO), to learn the velocity control of the Altiscan crawler by modeling its operating environment in a novel, GPU-accelerated simulation software called Isaac Gym. The approaches evaluated the error between measured base velocities of the crawler as a result of the actions provided by the DRL model and target velocities in six different environments. Two variants of PPO, standard and reccurent, were compared against the inverse velocity kinematics model of a differential-drive robot. The results show that velocity control in simulation is possible using PPO, but evaluation on the real crawler is needed to come to a meaningful conclusion.

# CHAPTER 1
# INTRODUCTION AND BACKGROUND

The use of mobile robots for inspection, maintenance, and repair in commercial and industrial settings has grown in recent years given their versatility and safety in hazardous environments. Unmanned aerial vehicles (UAVs), unmanned ground vehicles (UGVs), and unmanned underwater vehicles (UUVs), have been used to monitor a variety of infrastructure such as cell towers, bridges, and nuclear plants [1, 2]. Mobile robots have also been used in more complex environments that demand atypical features; instead of using traditional locomotion techniques, robots that are inserted into pipes can use wheels that attach magnetically to the pipage or attain a different kind of locomotion using bio-inspired movement [3]. Control of these robots can be difficult considering the multitude of external disturbances present in these unpredictable environments. In this thesis, the main focus is a magnetic, differential-drive crawling robot used for the inspection of large, ferromagnetic structures. This chapter provides the motivation for using such a robot, the problem this thesis aims to solve, the framework for a potential solution to this problem, and an outline for the rest of the thesis.

## 1.1 Motivation

The inspiration for this thesis comes from the BugWright2 Project [4], a multinational pilot venture funded by the European Union's Horizon 2020 research and innovation program for the autonomous inspection and cleaning of marine vessels. It is known that container ships face exposure to abrasive elements while at sea which can corrode their outer hulls. These corrosion patches can affect a ship's hydrodynamics and increase its fuel consumption, which increases the operating costs for marine commerce and contributes to environmental pollution. Currently, detailed inspections are performed manually by humans, which is a

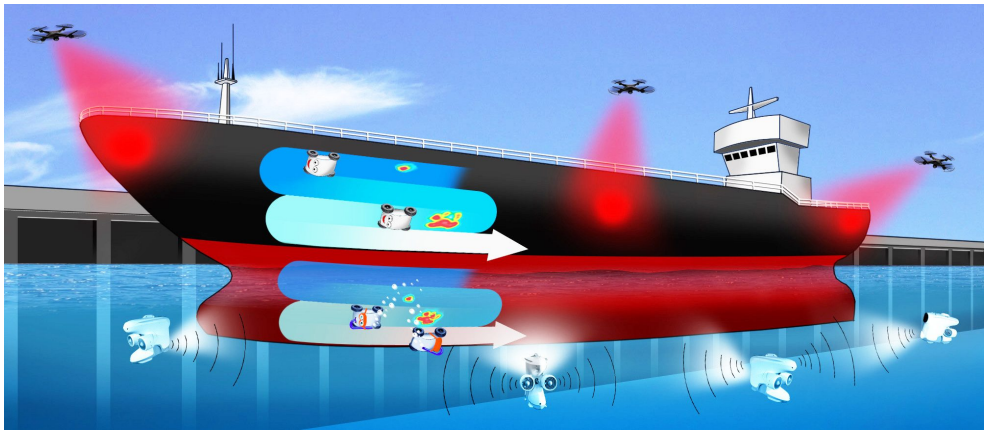


Figure 1.1: An illustration of the multi-robot interaction concept in the BugWright2 project. The coordinated scanning by UAVs, UUVs, and crawlers enable inspection of the hull of a ship without dry-docking [4].

time-consuming, expensive, and laborious process, particularly considering the immense length of large vessels (>50m). The BugWright2 initiative aims to take a proactive stance on detecting external defects by deploying a multi-robot network of UAVs, UUVs, and magnetic-wheeled crawling robots in order to minimize human intervention while maintaining inspection efficiency. Each robot is equipped with an array of visual, acoustic, and laser sensors. For such a critical part of inspection for ships, it is necessary to develop a framework that allows for the precise control of a robot to ensure accurate self-localization, detection of defects, and coordination between robots.

## 1.2 ROBOPLANET Altiscan Crawler

The magnetic-wheeled crawler used in the BugWright2 project is the primary focus of this thesis. The Altiscan by ROBOPLANET is a differential-drive mobile robot with two front wheels actuated by brushless DC motors and a passive caster wheel that provides magnetic support and balance [5]. Each wheel is comprised of a cylindrical electromagnet which provides 135N of magnetic force and enables the crawler to traverse vertical, ferromagnetic surfaces. Rotary wheel encoders capture the position of these wheels and a Roboclaw hardware motor controller provides a voltage proportional-integral-derivative (PID) con-

troller for their velocities. The carbon steel body of the robot gives it a spatial footprint of 35x35x20cm and a mass of 7kg. The crawler has been modified from its manufactured version with sensors that aid in autonomy. Several off-the-shelf sensors have been rigged to the frame of the crawler such as a 6-DOF inertial measurement unit (IMU), an ultrawide band (UWB) receiver, and an RGB camera. A forward-facing 3D LiDAR sensor is also attached and can provide point clouds of the surrounding environment. A contact piezoelectric transducer is included for the detection of corrosion patches and landmarks used in localization. On-board compute is provided by an embedded Axiomtek CAPA310 computer. A picture of the crawler can be found in Figure 1.2. In its current state, the crawler is a well-equipped but resource intensive machine which renders teleoperation infeasible. Hence, an umbilical tether that attaches to the crawler from some base station is required. The 30m long tether includes data and power cables, as well as a tube for water supply to ensure adequate contact between the piezoelectric transducer and the surface. The crawler can be controlled autonomously or manually using a standard joystick and can reach a top speed of 20cm/s. Manual control relies on the inverse velocity kinematics of a differential drive robot detailed later in Section 2.1. The autonomy package of the crawler relies on probabilistic state-estimation in the form of a mesh-constrained particle filter that incorporates IMU, UWB, and encoder (odometry) readings [6]. The control, sensing, and computing done by the crawler is conducted using the Robot Operating System (ROS) middleware [7].

### 1.3 Problem Definition

During field-testing of the crawler, operators qualitatively noted inconsistencies in its motion when traversing large areas non-proximal to the ground station. Specifically, during vertical transection, the crawler struggles to maintain a base linear velocity that matches the linear velocity commanded by the operator. The same issues are faced during horizontal transection. During rotation, the crawler tends to increase its base angular velocity when

Figure 1.2: An image of the ROBOPLANET Altiscan in action on a vertical surface. The cord attached to the chassis of the crawler is the tether that contains a bundle of transfer cables for data, power, and water supplies.

turning towards the ground and base station and decrease its base angular velocity when turning away to reach a higher point. With these locomotion artifacts, control of the crawler becomes difficult and a liability when working in a coordinated setting. One hypothesis for these inconsistencies is that the dynamics of the crawler and environment are highly nonlinear. The kinematics model of a differential drive robot operates on many assumptions detailed in Section 2.1 which may be impossible to abide by in the real world. The crawler has long, magnetic wheels, which add to the nonlinearity by introducing wheel slippage and friction into the equation. Second, the constant force from gravity can influence the motion of the crawler on vertical surfaces. Finally, the umbilical attached to the crawler can add a significant amount of weight. Since the tether is relatively free-moving aside from its two anchor points, the rope forces that act on the crawler can be difficult to model. Therefore, the objective of this thesis is to develop a controller to handle the nonlinear dynamics of the crawler's environment to achieve robust velocity control. Physics model-based control methods based on classical and modern control theory exist, but require a deep knowledge of the domain dynamics. Reinforcement learning (RL)—more specifically deep reinforcement learning (DRL)—has gained popularity in the past decade as a physics-naive approach to control systems. DRL has been used in autonomous vehicles to achieve smooth and efficient velocity control to optimize driving performance [8]. This thesis will explore the effectiveness of using DRL for the velocity control of a magnetic, tethered differential-drive robot.

## 1.4 Research Goal

The research goals of this thesis are to develop a simulation capable of capturing the dynamics of the crawler's environment, train a DRL velocity controller capable of tracking velocity better than a standard kinematics model, and provide a framework to test this model on a real system.

## 1.5 Outline

This thesis is structured as follows:

- Chapter 2 provides a comprehensive overview of the concepts examined in this thesis. Section 2.1 explains the forward and inverse kinematics model of a differential-drive robot. Section 2.2 details the basics of RL including policy optimization. Section 2.3 introduces the notion of using deep neural networks as nonlinear function approximators. Finally, Section 2.4 combines the previous two ideas to explain the state-of-the-art used in this thesis, including the implementation of proximal policy optimization (PPO).

- Chapter 3 surveys the various methods used for the control of differential-drive robots and other robots with nonlinear dynamics.

- Chapter 4 presents the simulation software Isaac Gym and details how the environment and RL task are setup for learning.

- Chapter 5 proposes different environments and metrics to evaluate the performance of a DRL velocity controller for the crawler.

- Chapter 6 evaluates the effectiveness to a DRL velocity controller for the proposed methods in comparison to the baseline kinematics model currently used in the crawler.

- Chapter 7 presents a summary of the work, detailing the strengths and weaknesses of the approach used, and provides direction for future work.

# CHAPTER 2

# FUNDAMENTALS

## 2.1 Differential Drive Robot Kinematics

The differential drive robot (DDR) is a versatile class of mobile robots characterized by a set of fixed standard wheels placed coaxially on opposite sides of a chassis. These wheels are typically powered by motors that provide the necessary torque to command the robot to drive in a certain direction at a specific velocity. The relatively cheap design requirements for a DDR make it suitable for a wide variety of use cases such as autonomous home vacuum cleaners [9], pharmaceutical warehousing [10], and storage tank inspection [6]. The velocity control of a DDR can be determined by its kinematics model under a set of critical assumptions:

- Movement occurs across a horizontal surface
- Wheels contact the surface at a single point
- Wheels are non-deformable
- Wheels are *pure rolling*, so no slipping, skidding, or sliding occurs
- No rotational friction around the point of contact
- Steering axis of wheels is orthogonal to the surface
- Wheels are connected by a rigid frame (chassis)

Under these constraints, the instantaneous change in position (linear velocity) is always in the steering direction of the DDR. The velocity in the frame of the robot body $v^{body}$ is

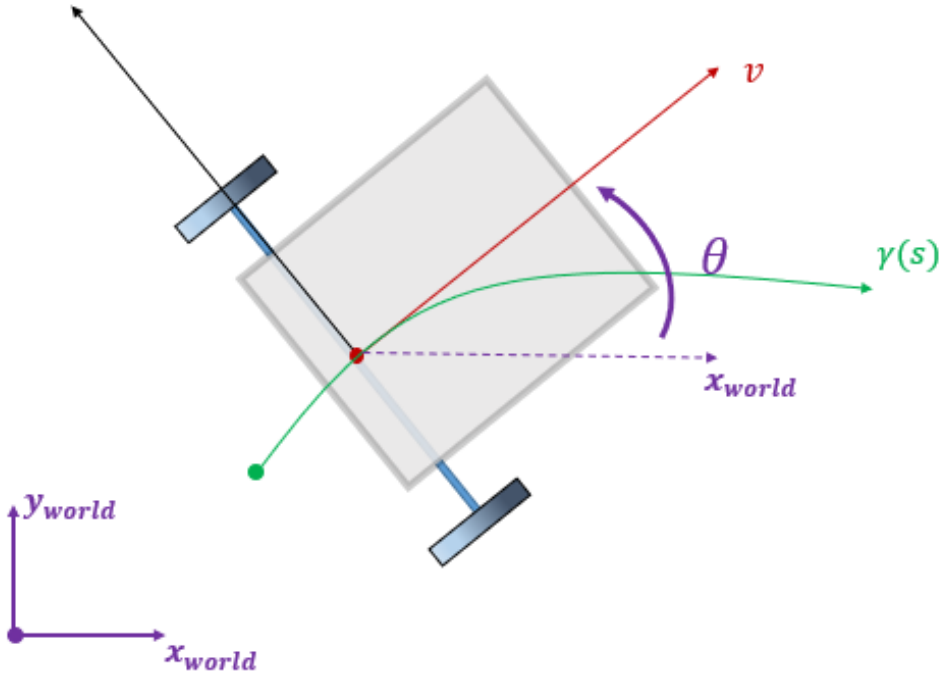



Figure 2.1: A representation of the velocity of a DDR. For a DDR that follows some path $\gamma(s)$ where $s$ defines its parameters, the velocity $v$ in the frame of the robot is tangent to the path with a rotational velocity derived from the change in its heading $\theta$ [11].

formalized as

$$v^{body} = \begin{bmatrix} v_x \\ 0 \\ \dot{\theta} \end{bmatrix}, \tag{2.1}$$

where $v_x$ is the velocity in the $x$-direction and $\dot{\theta}$ is the angular velocity of the base of the robot. It is important to note that the velocity in the $y$-direction $v_y$ is 0. Though a DDR cannot move freely along its lateral axis, it is able to perform turning maneuvers by rotating its wheels in opposite directions. To go straight, the angular velocity of both wheels $\dot{\phi}_{left}, \dot{\phi}_{right}$ must be equal to each other. The linear velocity of a wheel can be computed by multiplying its radius $r$ by its angular velocity

$$v_{wheel} = r\dot{\phi}_{wheel}. \tag{2.2}$$

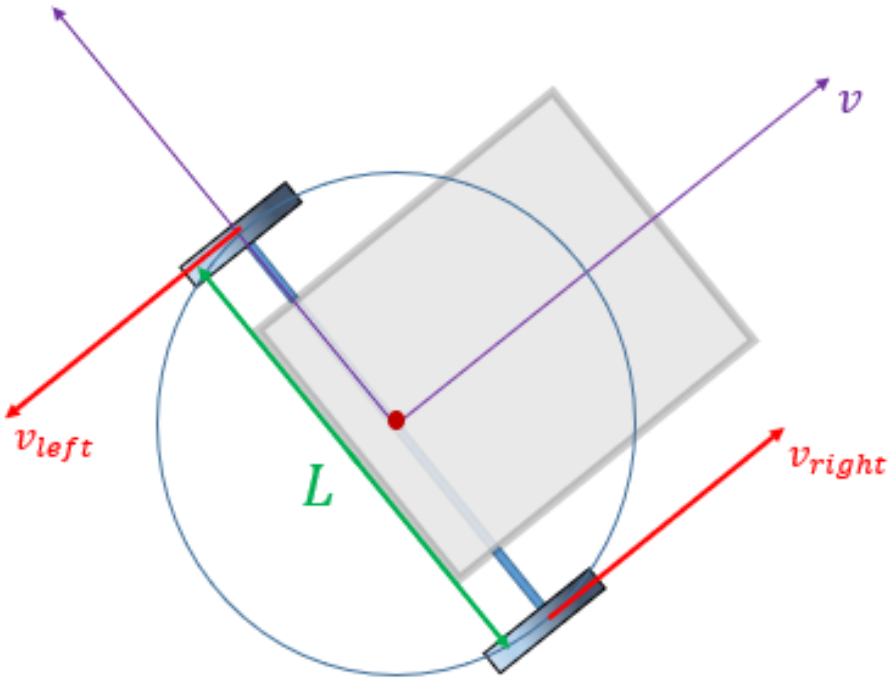



Figure 2.2: A representation of pure rotation for DDR. For an axle of length $L$, the instantaneous center of curvature (ICC) lies at the point $\frac{L}{2}$ along it. The linear velocities of each wheel are exact opposite, such that $-v_{left} = -r\dot{\phi}_{left}$ and $v_{right} = r\dot{\phi}_{right}$ [11].

By the pure rolling constraint, no energy is lost to the surrounding environment, so the velocities of each wheel are exactly equal to the base linear velocity $v_{wheel} = v_x$. By substitution, the angular velocity of each wheel is

$$\dot{\phi}_{wheel} = \frac{v_x}{r}. \tag{2.3}$$

While turning in place, the ICC for the DDR occurs exactly at the midway point along its axle. Point rotation is achieved by turning both wheels in opposite directions at the exact same speed, as shown in Figure 2.2. By applying the equations of circular motion, the

angular rotation of each wheel during point rotation is

$$\dot{\phi}_{left} = -\frac{L\dot{\theta}}{2r} \tag{2.4}$$

$$\dot{\phi}_{right} = \frac{L\dot{\theta}}{2r}. \tag{2.5}$$

The inverse velocity kinematics (IVK) of a differential drive robot can be found by combining the pure translation and pure rotation models.

$$\dot{\phi}_{left} = \frac{v_x}{r} - \frac{L\dot{\theta}}{2r} \tag{2.6}$$

$$\dot{\phi}_{right} = \frac{v_x}{r} + \frac{L\dot{\theta}}{2r} \tag{2.7}$$

IVK is useful for computing target wheel velocities in order to reach some desired base velocity. Forward velocity kinematics, on the other hand, determine the expected base velocity given the angular velocities applied to each wheel, which can be found by algebraically rearranging the terms in Equation 2.6 and Equation 2.7, respectively:

$$v_x = \frac{r}{2}(\dot{\phi}_{right} + \dot{\phi}_{left}) \tag{2.8}$$

$$\dot{\theta} = \frac{r}{L}(\dot{\phi}_{right} - \dot{\phi}_{left}). \tag{2.9}$$

## 2.2 Reinforcement Learning

Reinforcement learning (RL) is a subset of machine learning that provides a learner the means to maximize a numerical feedback signal through an iterative trial-and-error decision-making process. The RL dichotomy, as seen in Figure 2.3, consists of the learner, traditionally referred to as the *agent*, and the *environment*. The agent decides what actions to take based on the observations it extracts from the environment. Executing an action allows an agent to interact with its environment, which in turn gives a *reward*, a scalar value that quantifies how good a particular action is in an instant. This reward is accumulated over

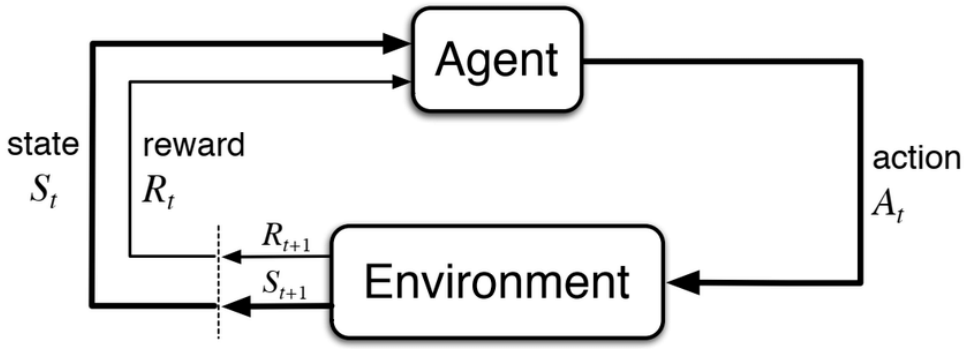



Figure 2.3: The agent-environment interaction in reinforcement learning. An agent exists in state $S_t$ and takes an action $A_t$ at each timestep $t$. Based on the dynamics of the environment, the agent will end up in state $S_{t+1}$ with a reward $R_{t+1}$ [12].

time after a sequence of actions has been performed. The goal of an RL agent is to balance exploration and exploitation to find an optimal mapping between states and actions, also known as a *policy*, that maximizes the expected sum of rewards. The following section is based on Sutton and Barto's work [12] and will look at the formalization of the RL problem, the difference between approaches to model-based and model-free problems, and optimizations to these approaches.

### 2.2.1 Markov Decision Processes

A Markov process (MP) is a stochastic process that satisfies the memorylessness of the Markov property where the current state, $S_t$, encompasses all information needed to make an adequate prediction of future outcomes. MPs are typically simplified to a Markov Chain with a discrete number of states defined by the tuple $\langle \mathcal{S}, \mathcal{P}_{ss'} \rangle$, where:

- $\mathcal{S}$: the finite set of states, or *state space*, where $s \in \mathcal{S}$
- $\mathcal{P}_{ss'}$: $(\mathcal{S} \times \mathcal{S}) \rightarrow [0, 1]$, the state transition matrix defining the probability of reaching state $s'$ from $s$

By interacting with the environment, an agent is given feedback in the form of a reward. This is formalized by the Markov reward process (MRP). An MRP is a 4-tuple $\langle \mathcal{S}, \mathcal{P}_{ss'}, \mathcal{R}_s, \gamma \rangle$, where $\mathcal{S}$ and $\mathcal{P}_{ss'}$ are the same as before, and $\mathcal{R}_s$ and $\gamma$ define:

- $\mathcal{R}_s$: $(\mathcal{S} \times \mathcal{S}) \to \mathbb{R}$, the expected reward over all possible states reachable from $s$

$$\mathcal{R}_s = \mathbb{E}[R_{t+1} | S_t = s]$$

- $\gamma$: the reward discount factor, $\gamma \in [0, 1]$

One limitation of MRPs is the inability to model sequential decision-making through actions. The Markov decision process (MDP) addresses this by extending the MRP to include a set of finite actions. MDPs are formalized by the 5-tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}^a_{ss'}, \mathcal{R}^a_s, \gamma \rangle$, where:

- $\mathcal{S}$: the set of states, or *state space*, where $s \in \mathcal{S}$
- $\mathcal{A}$: set of actions, or *action space*, where $a \in \mathcal{A}$
- $\mathcal{P}^a_{ss'}$: $(\mathcal{S} \times \mathcal{S} \times \mathcal{A}) \to [0, 1]$, the state transition function

$$\mathcal{P}^a_{ss'} = p(s'|s, a) = p(S_{t+1} = s' | S_t = s, A_t = a)$$

- $\mathcal{R}^a_s$: $(\mathcal{S} \times \mathcal{A} \times \mathcal{S}) \to \mathbb{R}$ the expected reward over all possible states reachable from $s$ by taking action $a$

$$\mathcal{R}^a_s = r(s, a, s') = \mathbb{E}[R_t | S_t = s, A_t = a, S_{t+1} = s']$$

- $\gamma$: the reward discount factor, $\gamma \in [0, 1]$

### 2.2.2 Returns

By defining a reward signal, the agent knows the difference between what it should and should not achieve. Its objective is now to maximize the *expected return*, $G_t$, which can be formally defined as:

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + ... + R_T,$$

where $T$ is the time step in which the agent enters a *terminal state*. However, many RL problems are continuous tasks and do not enter a natural terminal state, so $T = \infty$. In this case, the agent must choose an action to maximize the *expected discounted return*:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.10}$$

### 2.2.3 Policy

A policy $\pi$ defines the behavior of an agent in a particular state. It can be modeled by a probability distribution $\pi(a|s)$ where $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$. In a stochastic policy, an action will be sampled from $\pi(\cdot|s)$, whereas in a deterministic policy $\pi(s)$, an action will be picked uniquely based on the state.

### 2.2.4 Value Functions

RL algorithms depend on an estimate of how good it is for an agent to be in its current state. If a policy $\pi$ is given, the *state-value function* $v_\pi(s)$ defines the expected return of a given state $s$, provided the agent follows the actions sampled from the policy:

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s\right], \forall s \in \mathcal{S} \tag{2.11}$$

This formulation gives rise to a recursive relation known as the *Bellman Equation for* $v_\pi$. This relation becomes apparent by expanding the previous equation:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_t|S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s] \\ &= \sum_{a\in\mathcal{A}} \pi(a|s) \sum_{s'\in\mathcal{S}} \mathcal{P}^a_{ss'} \Big( \mathcal{R}^a_s + \gamma \mathbb{E}_\pi \big[ G_{t+1}|S_{t+1} = s' \big] \Big) \\ &= \sum_{a\in\mathcal{A}} \pi(a|s) \Big( \mathcal{R}^a_s + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}^a_{ss'} v_\pi(s') \Big) \end{aligned} \tag{2.12}$$

Similarly, the *action-value function* $q_\pi(s, a)$ defines the expected return of being in a given state $s$ and taking a certain action $a$, provided the agent follows the actions sampled from the policy thereafter:

$$\begin{aligned} q_\pi(s, a) &= \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \\ &= \mathbb{E}_\pi \Big[ \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s, A_t = a \Big], \forall s \in \mathcal{S}, \forall a \in \mathcal{A} \end{aligned}$$

There also exists a *Bellman Equation for* $q_\pi$

$$\begin{aligned} q_\pi(s, a) &= \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s, A_t = a] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1})|S_t = s, A_t = a] \\ &= \mathcal{R}^a_s + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}^a_{ss'} \sum_{a'\in\mathcal{A}} \pi(a'|s') q_\pi(s', a') \end{aligned} \tag{2.13}$$

### 2.2.5 Optimality

By formally defining the value functions, a partial ordering over policies becomes apparent. A policy $\pi$ is considered equivalent to or better than another policy $\pi'$ if:

$$v_\pi(s) \geq v_{\pi'}(s), \forall s \in \mathcal{S}$$

An *optimal policy* $\pi_*$ is a policy that is equivalent to or better than every other policy. The *optimal state-value function* $v_*(s)$ is defined as the state-value function given an optimal policy, or formally:

$$v_*(s) = \max_\pi v_\pi(s), \forall s \in \mathcal{S}$$

Likewise, the *optimal action-value function* $q_*(s, a)$ can be defined formally as:

$$q_*(s, a) = \max_\pi q_\pi(s, a), \forall s \in \mathcal{S}, \forall a \in \mathcal{A}(s)$$

### 2.2.6 Monte Carlo Methods

In RL, if the model of the environment is given (the full MDP is known), then a generalized policy iteration (GPI) framework consisting of a policy evaluation and policy improvement routine can be used to find the optimal value function and optimal policy after some time with dynamic programming. In practice, MDPs that capture all the dynamics of an environment are difficult to create. For example, without prior knowledge of the state-transition function $\mathcal{P}$, the Bellman optimality equations cannot be solved. Monte Carlo (MC) methods aim to address this problem by learning from simulated experience through episodic tasks.

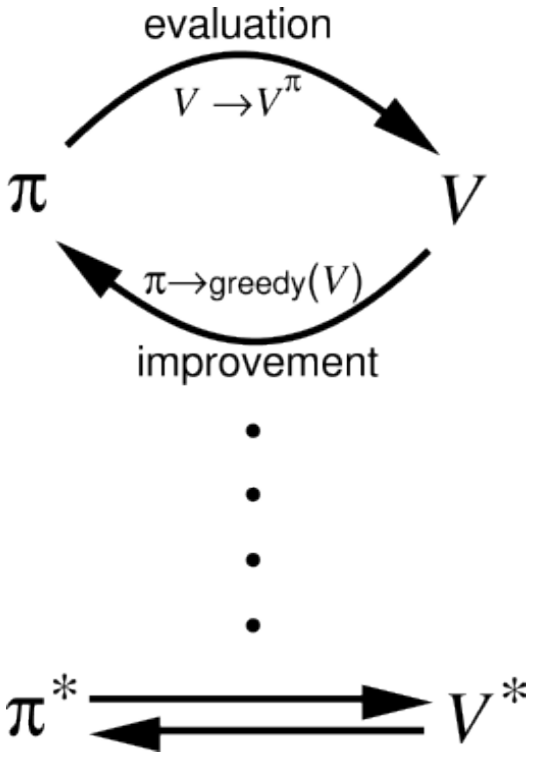



Figure 2.4: An illustration of the GPI flow. Every policy evaluation is subsequently followed by a policy improvement step. After many iterations, the agent will converge to an optimal policy and optimal value function [12].

*Prediction*

The first part of GPI is policy evaluation. To evaluate a policy $\pi$ using MC, an agent must sample from a generic policy to create an episode of $S_0, A_0, R_1, S_1, ..., S_{T-1}, A_{T-1}, R_T$. The estimated value of a state $V(S_t)$ can be updated incrementally over each episode, such that

$$N(S_t) = N(S_t) + 1$$

$$V(S_t) = V(S_t) + \frac{1}{N(S_t)}(G_t - V(S_t)) \tag{2.14}$$

$$= V(S_t) + \alpha(G_t - V(S_t)) \tag{2.15}$$

where $N(S_t)$ represents the number of times the state is visited. $\alpha$ is a constant step-size parameter . This update can be applied the first time (*first-visit*) or every time (*every-visit*)

a state is seen in an episode. $V(s)$ will converge to $v_\pi(s)$ as $N(s) \to \infty$. While an estimate of the state-value function is useful when the transition model is known, an estimate of the state-action value function captures more information.

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha(G_t - Q(S_t, A_t)) \tag{2.16}$$

Special consideration must be given to ensure that all state-action pairs are explorable. The MC prediction utilizes *exploring starts* which specifies that each episode starts with a state-action pair and that each pair has a non-zero probability of being selected. Over an infinite number of sampled episodes, all pairs will be visited and $Q(s, a)$ will converge to $q_\pi(s, a)$.

*Control*

The second part of GPI is policy improvement—a way to generate a new policy $\pi'$ such that $\pi' \geq \pi$. Using $Q(s, a)$, a greedy policy improvement scheme looks like:

$$\pi'(s) = \underset{a \in \mathcal{A}}{\operatorname{argmax}} Q(s, a) \tag{2.17}$$

Unfortunately, the exploring starts assumption is unrealistic. A more approachable way to ensure all state-action pairs are visited is to use an *on-policy* (improving the policy used to make the decisions) MC control method called $\epsilon$-*greedy*. In this method, when updating a policy $\pi$, all actions begin with a minimum probability $\frac{\epsilon}{|\mathcal{A}(s)|}$. The greedy action receives an added incentive of $1 - \epsilon$ of being chosen. Formally, a policy with $\epsilon$-greedy exploration is

$$\pi(a|s) = \begin{cases} \frac{\epsilon}{|\mathcal{A}(s)|} + 1 - \epsilon & a^* = \operatorname{argmax}_{a \in \mathcal{A}} Q(s, a) \\ \frac{\epsilon}{|\mathcal{A}(s)|} & \text{otherwise} \end{cases} \tag{2.18}$$

Over an infinite number of episodes, an $\epsilon$-greedy exploration policy will converge to an optimal policy $\pi_*$ by the Greedy in Limit with Infinite Exploration (GLIE) definition [13].

### 2.2.7 Temporal Difference Learning

Like MC methods, temporal-difference (TD) methods do not require a full MDP and use the GPI framework to evaluate and improve a policy. TD also uses raw experience to learn, but rather than estimating the value function by sampling entire episodes, it learns from incomplete episodes by *bootstrapping* the target value.

*Prediction*

To reiterate, the goal of the prediction stage is to evaluate the value function given a policy $\pi$. Previously, $V(s)$ was updated based on the true return $G_t$ calculated from the trajectory of the sampled episode. TD learning algorithms no longer have this ability and must shift the value function towards an estimated return called the *TD target*. The new state-value function update looks like

$$V(S_t) = V(S_t) + \alpha\big[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\big], \tag{2.19}$$

where $R_{t+1} + \gamma V(S_{t+1})$ is the TD target and $R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$ is the *TD error* for a one-step look-ahead. This TD target can also be adapted for further look-aheads using $n$-step returns.

*Control*

TD control can be split up into two distinct and commonly used methods: *Sarsa* and *Q-Learning*. As with MC control, TD control uses the action-value function to account for an incomplete MDP.

Sarsa is an on-policy TD control method that uses 5 sequential events $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$ to update the estimated action-value function $Q(S_t, A_t)$. The action $A_{t+1}$ taken from the next state is chosen directly from the current policy $\pi$.

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha\big[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\big] \tag{2.20}$$

Q-Learning is a TD control method that also uses bootstrapping to estimate the action-value function, but unlike Sarsa, the action taken in the subsequent state depends on the action that gives the maximum value from the current action-value function. By removing reliance on the current policy to determine the next action, this method is considered off-policy.

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha\big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\big] \tag{2.21}$$

### 2.2.8 Value Function Approximation

Reinforcement learning can be used to solve problems of varying complexity. For relatively simple problems with small state spaces, a tabular representation of the value function suffices to capture all possibilities for the MDP. However, as the problem grows increasingly complex, such as in robotic control tasks with a continuous state and action space, a very-large—sometimes infinite—tabular solution is needed, which can render policy optimization computationally expensive or impossible. Instead, parameterized functions are used to approximate the value of a state of state-action pair. Linear approximators, such as polynomials and Fourier basis functions [14], and non-traditional approximators, such as decision trees [15] and nearest neighbor methods [16], have been used. Most notably, however, artificial neural networks have been widely used as nonlinear value function approximators.

## 2.3 Deep Learning

Deep learning (DL)—in contrast to shallow learning methods such as linear regression, support vector machines, decision trees, etc.—is a machine learning paradigm used to approximate nonlinear functions by layering computation flows between an input and output

[17]. These models have been used extensively in medical imaging, autonomous driving, and robotic controls [18, 19]. For the purposes of this thesis, deep learning will be used as a tool to approximate the value functions in RL for complex tasks. The upcoming section is largely based on the work by Goodfellow et. al [20] and Bishop [21].

### 2.3.1 Perceptrons

Neural networks are the basis behind DL. They are mathematical representations that loosely model the neuron in the animal brain. The neuron works by accumulating some impulses from its input branches, called dendrons, until the polarization reaches a certain threshold, after which it propagates an action potential down its output branches, called axons. Similarly, neural networks consist of single neurons, called *perceptrons*, that sum together an input $x$ balanced by some weight and a bias $b$ to produce an output $z$.

$$z = \sum_i w_i x_i + b \tag{2.22}$$

### 2.3.2 Activation Functions

After the perceptron receives an input, it defines a threshold at which the signal can be propagated. In neuroscience, this threshold is thought of as all-or-nothing; however, the perceptron allows for some granular control by specifying an *activation function* to produce an output. These activation functions give rise to the nonlinear behavior desired by neural networks. Some commonly-used activation functions include:

1. Sigmoid: The sigmoid activation function limits the range of the output between 0 and 1.

$$f(z) = \frac{1}{1 + e^{-z}} \tag{2.23}$$

2. Hyperbolic tangent (tanh): The tanh activation function limits the range of the output

between -1 and 1.

$$f(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{2.24}$$

3. Rectified linear unit (ReLU): The ReLU activation function zeros all negative inputs while keeping the true values for all non-negative inputs.

$$f(z) = \begin{cases} 0 & \text{when } z \leq 0 \\ z & \text{when } z > 0 \end{cases} \tag{2.25}$$

### 2.3.3 Deep Neural Networks

A perceptron can have many inputs, and when those inputs are shared across multiple perceptrons, a fully-connected layer emerges. A deep neural networks (DNNs) extends this structure to include *hidden layers*, layers that take the outputs of a previous layer as inputs. DNNs have the capability to model more complex data.

### 2.3.4 Training

For a DNN to model a nonlinear function, its parameters must be fine-tuned to line up with the given data. Let $\theta$ represent the parameters of the DNN. For some input $x$, a forward pass through a DNN will give an estimated value $\hat{y}$. In order for the network to learn, a target value $y$ for the input $x$ must be provided. To judge the quality of the predicted value, a loss function $\mathcal{L}$ can be used. A commonly used loss function is the $p$-norm:

$$\mathcal{L}_p = \|y - \hat{y}\|_p^p = \sum_{i=1}^{N} |y_i - \hat{y}_i|^p \tag{2.26}$$

The goal of training a DNN is to update its weights $\theta$ such that the loss $\mathcal{L}$ is minimized. Theoretically, $\nabla_\theta \mathcal{L} = 0$ is the ideal solution; however, the complexity of the DNN prevents

an exact solution. Instead, an algorithm called *back-propagation* uses *stochastic gradient descent* to recursively update the weights and biases of each layer in the DNN such that:

$$\theta_{t+1} = \theta_t - \alpha \nabla_{\theta_t} \mathcal{L}, \tag{2.27}$$

where $\alpha$ controls the learning rate.

### 2.3.5 Recurrent Neural Networks

Recurrent neural networkss (RNNs) are a special type of neural network that operate on data that do not necessarily abide by the Markov property. DNNs have trouble generating sequential data such as stock prices, temperature, and natural language, but having knowledge of their previous states can be useful in predicting future states. In a feed-forward DNN, the flow between input to output is unidirectional. To give the sense of memory, an RNN adds a looping mechanism that takes the output of the hidden layer as an input to itself. Let $h_t$ be the output of the current hidden state and $h_{t-1}$ be the output of the previous hidden state. The current output can be defined as:

$$h_t = f(W[h_{t-1}; x_t] + b), \tag{2.28}$$

where $W$ is the matrix of weights, $x_t$ is the input vector concatenated to $h_{t-1}$, $b$ is the bias term, and $f$ is function that adds nonlinearity, most often the sigmoid or tanh activation functions. To train this type of network, a method called back-propagation through time (BPTT) is used which adapts the back-propagation algorithm to update the weights between the input and the hidden layer, inside the hidden layers themselves, and between the hidden layers of each time series. However, RNNs suffer from the *exploding gradient* and *vanishing gradient* problems during BPTT. The exploding gradient problem occurs when the magnitude of the eigenvalues of $W$ are greater than 1 which causes the output $h$ to increase exponentially and unbounded. The vanishing gradient problem suffers from the

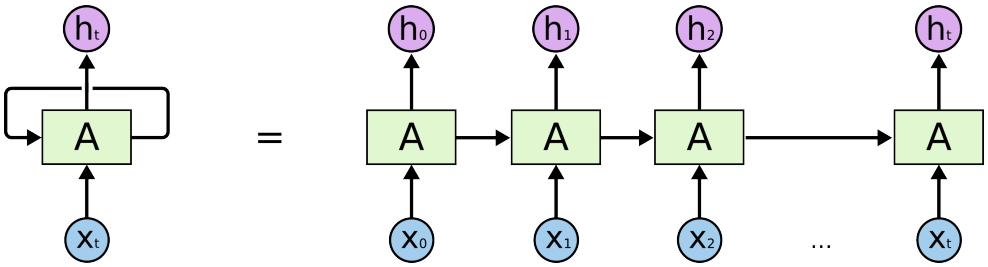



Figure 2.5: An illustration of a simple RNN. The recursive flow of a RNN gives it the ability to learn across a related sequence of inputs [22].

opposite where the magnitude of the eigenvalues of $W$ are less than 1 which causes the output $h$ to approach 0 if the input sequence is long enough.

### 2.3.6 Long-Short Term Memory Networks

Long-short term memory (LSTM) networks [23] are an extension to RNNs that solve the exploding and vanishing gradient problems. Rather than relying on a single activation function to transform the combination of inputs and weights, a series of gates are used to preserve information across an entire sequence of inputs as seen in Figure 2.6. The long-term memory of an LSTM is stored in the cell-state $C_t$. The short-term memory, or hidden state memory, is stored in $h_t$. The control gates in an LSTM cell are:

- Forget gate: The forget gate uses the sigmoid activation function to determine whether the previous cell state $C_{t-1}$ should be forgotten based on the current input $x_t$ and the previous hidden layer output $h_{t-1}$.

$$f_t = \sigma(W_f[h_{t-1}; x_t] + b_f) \tag{2.29}$$

- Input gate: The input gate also uses a sigmoid activation function but uses it to determine what the new information to store in the cell state.

$$i_t = \sigma(W_i[h_{t-1}; x_t] + b_i) \tag{2.30}$$

- Output gate: The output gate takes the input and passes it through another sigmoid activation function but will eventually be combined with the current cell state to produce a memory-balanced output.

$$o_t = \sigma(W_o[h_{t-1}; x_t] + b_o) \tag{2.31}$$

The LSTM network also contains two update steps:

- Cell state update: The cell state update weights the previous cell state $C_{t-1}$ by the output of the forget gate $f_t$ and adds to it a candidate cell state weighted by the output of the input gate.

$$C_t = f_t \times C_{t-1} + i_t \times \tanh(W_C[h_{t-1}; x_t] + b_C) \tag{2.32}$$

- Hidden state update: The hidden state update gives the cell output and the short-term memory to be used in the next cell state. This update takes current updated cell state and weights it by the output of the output gate $o_t$.

$$h_t = o_t \times \tanh(C_t) \tag{2.33}$$

## 2.4 Deep Reinforcement Learning

### 2.4.1 Introduction

Deep reinforcement learning (DRL) has garnered much interested among RL researchers in recent years. Classical RL assumes tables are used to store the values for the state-value and action-value functions. This, however, is only suitable for problems with small state spaces; more complex problems are defined by larger state spaces. For example, board games, while seemingly simple, consist of a large amount of states—backgammon has about $10^{20}$

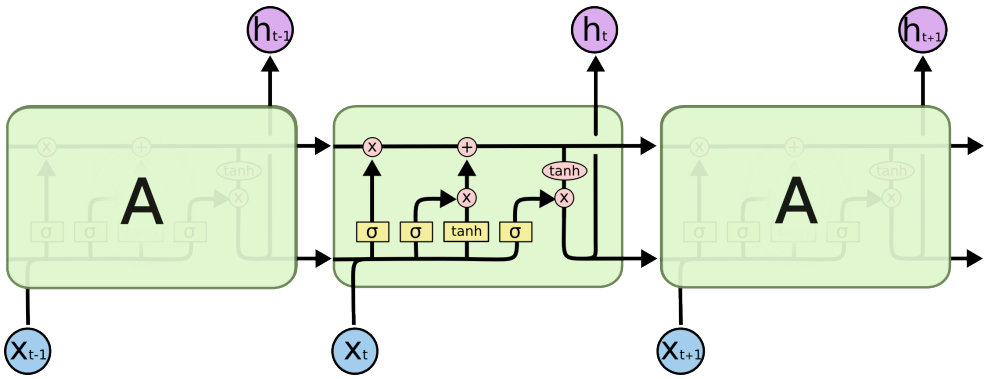



Figure 2.6: An illustration of the cell configuration of an LSTM network. The top line acts as a highway for the cell state which retains long-term memory across all cells. The bottom line represents the short-term memory dependent only on the output of the hidden layer from the previous cell [22].

states, chess has about $10^{40}$ states, and Go has about $10^{170}$ states. The compute power required to handle finding an optimal policy scales with a problem's complexity. DNNs are nonlinear function approximators that can act directly has value functions and/or policies. They have been deployed successfully to create agents that achieve human-level play in the aforementioned games [24, 25, 26]. Control tasks typically exist in environments with continuous state and/or action spaces. DRL can also be used to provide optimal control to maximize a relevant goal in this domain, such as smart grid energy optimization [27] and robotic manipulation [28].

### 2.4.2 Value-Based Methods

Q-Learning provided an intuitive solution to tabular methods of off-policy classical RL problems by bootstrapping the value function and choosing the next action based on the action-value from the $Q$-table. With DRL, the action-value function used is parameterized as the *$Q$-network* $Q_\theta(s, a)$. Rather than looking to update cells in a table, the objective is to update the parameters of the DNN backing the $Q$-network. To do this, a loss function must be defined to guide the parameters to optimality by way of gradient descent. An example

mean-squared error (MSE) loss function is

$$\mathcal{L}(\theta) = \mathbb{E}_{s,a,s',r\sim\pi}\left[(r + \gamma \max_{a'} Q_\theta(s', a') - Q_\theta(s, a))^2\right] \tag{2.34}$$

The previous function acts as an optimization objective between the approximate action-value function $Q_\theta(s, a)$ and an estimate of the true action-value function $r+\gamma \max_{a'} Q_\theta(s', a')$, or the target. This loss function, however, suffers from two major drawbacks: the target is always changing because it depends on $Q_\theta$ itself and the data used as inputs to the DNN are not independent and identically distributed (IID) random variables. This causes instability during value iteration and can prevent convergence to even a local minimum [14].

*Deep Q-Network*

The target-chasing and IID problems raised before can be addressed by introducing the deep Q-network (DQN) [29] with two key ideas. The first idea is to employ the use of an *experience replay* [30]. Over the course of an episode, the agent experiences its current state, current action, reward, and next state at every timestep summarized by the 4-tuple $(S_t, A_t, R_t, S_{t+1})$. These experiences are stored in the replay buffer $\mathcal{D}$, typically capped at a size of $N$, and randomly sampled in mini-batches for the control update. This sampling reduces the correlation between sequential experiences and thus minimizes most problems caused by IID data. The second idea is to separate the target and estimated state-value functions using a *target network*. This specialized network delays the update of the target network parameters $\theta^-$ by synchronizing with the estimated state-value function after $C$ timesteps. Formally, the MSE of a DQN is:

$$\mathcal{L}(\theta) = \mathbb{E}_{s,a,s',r\sim\mathcal{D}}\left[(r + \gamma \max_{a'} Q_{\theta^-}(s', a') - Q_\theta)^2\right] \tag{2.35}$$

Together, the experience replay and target network of a DQN help improve stability and reduces oscillation in the $Q$-network during training.

### 2.4.3 Policy Gradient Methods

DQN parameterizes the value function by using a DNN to determine the appropriate values given some state and action. To create an optimal policy, a traditional $\epsilon$-greedy policy can be layered above the $Q$-network to determine a noisy best action. However, value-based methods still fail to accommodate larger, continuous action spaces. *Policy gradient* methods abstract the policy above a DNN. By parameterizing the policy to $\pi_\theta$, an agent is able to optimize the policy directly, which gives rise to better convergence properties. The objective is to find a parameter set $\theta$ that maximizes some objective function $J$.

$$\theta^* = \underset{\theta}{\operatorname{argmax}}\, J(\theta) \tag{2.36}$$

This objective function reflects the agent's goal to maximize the cumulative discounted reward across a trajectory $\tau$

$$J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}[R(\tau)] \tag{2.37}$$

$$= \mathbb{E}_{\tau\sim\pi_\theta}\big[\sum_{t=0}^{T}\gamma^t R_t\big] \tag{2.38}$$

Two main methods exist to optimize the policy parameters: *gradient-free* and *gradient-based*. Gradient-free methods use cross-entropy loss to update the parameters, whereas gradient-based methods take the gradient of the objective function $J$ and perform *gradient-ascent* to push the policy in a direction that maximizes the discounted cumulative reward. Gradient ascent is outlined below.

$$\theta = \theta + \alpha\nabla_\theta J(\theta) \tag{2.39}$$

By way of the log-derivative trick and the notion of reward-to-go, where the cumulative reward is only computed after an action $A_t$, the gradient of the objective function is

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(A_t|S_t) R(\tau)\right] \tag{2.40}$$

$$= \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(A_t|S_t) \sum_{t'=t}^{T} R_{t'}\right] \tag{2.41}$$

This vanilla policy gradient is referred to as *REINFORCE* [31].

### 2.4.4 Actor-Critic Methods

The policy update depends directly on the update rule seen in Equation 2.26. However, this update suffers from high variance because the gradient of the objective function depends on an entire trajectory, similar to MC methods. Actor-critic (AC) [32, 33] methods are similar to TD-learning and work by bootstrapping the estimated parameterized value function. The *actor* is the parameterized policy $\pi_\theta$ and the critic is the parameterized value function $V_\psi^{\pi_\theta}$, where $\psi$ is the parameter set for the DNN backing the value function. Practically, $\theta$ and $\psi$ do not need to be separate parameter sets; the network can be shared across the policy and value function and still works as a nonlinear function approximator for both. The AC algorithm combines gradient descent and gradient ascent such that the parameter updates are:

$$\psi = \psi - \alpha_\psi \nabla_\psi \mathcal{L}_{V_\psi^{\pi_\theta}}(\psi) \tag{2.42}$$

$$\theta = \theta + \alpha_\theta \nabla_\theta J(\theta), \tag{2.43}$$

for a loss function $\mathcal{L}_{V_\psi^{\pi_\theta}}(\psi)$ that calculates the loss between a target and estimated value and an objective function $J(\theta)$. To accommodate the high variance from REINFORCE, an advantage estimation $\phi_t$ can be used in place of the cumulative reward to reduce variance and improve optimization capabilities without affecting the overall expected value, given

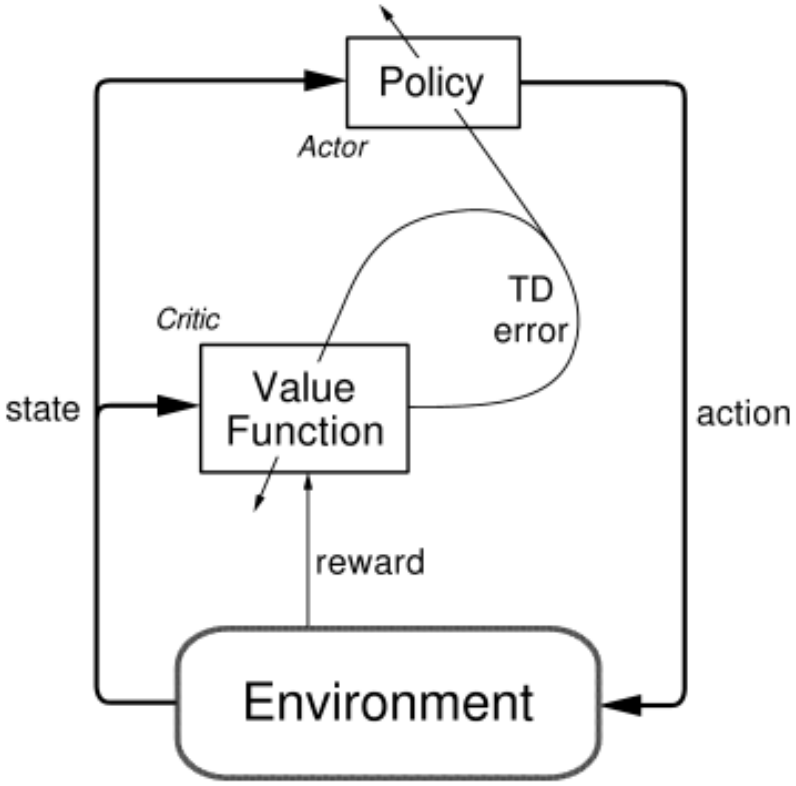



Figure 2.7: A diagram of the actor-critic architecture. [12]

that the baseline only depends on the current state $S_t$. This can be proven by the *Expected Grad-Log-Prob Lemma*, which states

$$\mathbb{E}_{x\sim P_\theta}\big[\nabla_\theta \log P_\theta(x)\big] = 0, \tag{2.44}$$

for a parameterized probability distribution $P_\theta$ over a random variable $x$. Consequentially, for some baseline $b$ dependent on state $s_t$, if:

$$\phi_t = \sum_{t=0}^{T} R_t - b(s_t), \tag{2.45}$$

then

$$\mathbb{E}_{\tau\sim\pi_\theta}\big[\nabla_\theta \log \pi_\theta(A_t|S_t)b(S_t)\big] = 0. \tag{2.46}$$

A common advantage estimation function is the *advantage* itself which describes the value of taking a certain action $A_t$ from state $S_t$ compared to the overall intrinsic value of being in state $S_t$.

$$\phi_t = A^{\pi_\theta}_\psi(S_t, A_t) = Q^{\pi_\theta}_\psi(S_t, A_t) - V^{\pi}_\psi(S_t) \tag{2.47}$$

Thus, a simple objective function for an advantage actor-critic (A2C) that reduces the overall variance is

$$J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\big[A^{\pi_\theta}_\psi(S_t, A_t)\big] \tag{2.48}$$

The critic loss function can be defined as

$$\mathcal{L}_{V^{\pi_\theta}_\psi}(\psi) = \mathbb{E}_{\tau\sim\pi_\theta}\big[A^{\pi_\theta}_\psi(S_t, A_t)^2\big] \tag{2.49}$$

### 2.4.5 Trust Region Policy Optimization

Policy gradient methods suffer from a few issues that can severely affect performance. First, the first-order gradient computed by $\nabla_\theta J(\theta)$ can lead to undesirably large steps if the objective function has a dynamic curvature to it. Second, the trajectories used to update $\theta$ are sampled directly from $\pi_\theta$ and place a heavy dependence on the tuning of the step size parameter $\alpha_\theta$. A solution to both of these problems can be found in the trust-region policy optimization (TRPO) [34] algorithm. The idea behind TRPO stems from [35] where a new policy $\pi_{\theta'}$ is updated such that it improves on the current policy $\pi_\theta$. The objective function that details this improvement is:

$$J(\theta') = J(\theta) + \mathbb{E}_{\tau\sim\pi_{\theta'}}\left[\sum_{t=0}^{\infty}\gamma^t A^{\pi_\theta}(S_t, A_t)\right] \tag{2.50}$$

$$\tag{2.51}$$

However, the bonus term takes a trajectory over an infinite horizon from $\pi_{\theta'}$ which cannot be sampled from. So, TRPO uses *importance sampling* to sample states from the state space of the old policy and actions from the old policy given that state. In order to do this, the advantage function must be weighted by the action probabilities under the new policy. The bonus term can then be represented as:

$$\mathcal{L}_{\pi_\theta}(\pi_{\theta'}) = \mathbb{E}_{\tau\sim\pi_{\theta'}}\left[\sum_{t=0}^{\infty}\gamma^t A^{\pi_\theta}(S_t, A_t)\right] = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{\infty}\gamma^t \frac{\pi_{\theta'}(A_t|S_t)}{\pi_\theta(A_t|S_t)} A^{\pi_\theta}(S_t, A_t)\right], \tag{2.52}$$

where $\mathcal{L}_{\pi_\theta}(\pi_{\theta'})$ as a loss function represents the surrogate advantage function of $\pi_{\theta'}$ with importance sampling. Luckily, a theoretical constraint emerges from the approximation error and can be expressed in terms of the worst-case Kullback-Leibler (KL) divergence $D_{KL}^{max}$ between the old and new policies.

$$|J(\theta') - J(\theta) - \mathcal{L}_{\pi_\theta}(\pi_{\theta'})| \leq C\, D_{KL}^{max}(\pi_\theta \| \pi_{\theta'}) \tag{2.53}$$

By way of Lagrangian duality, an optimal policy can be found using

$$\begin{aligned} \max_{\pi_{\theta'}} \quad & \mathcal{L}_{\pi_\theta}(\pi_{\theta'}) \\ \text{subject to} \quad & \mathbb{E}_{s\sim\rho_{\pi_\theta}}\left[D_{KL}(\pi_\theta \| \pi_{\theta'})\right] \leq \delta, \end{aligned} \tag{2.54}$$

where $\mathbb{E}_{s\sim\rho_{\pi_\theta}}\left[D_{KL}(\pi_\theta \| \pi_{\theta'})\right]$ is the trust-region function subjected to the constraint $\delta$. Solving this constraint optimization problem can be done using the *natural gradient*, which includes the second-order derivative, and gives an analytic form to the new policy parameter update:

$$\theta' = \theta + \sqrt{\frac{2\delta}{g^\top H^{-1} g}} H^{-1} g \tag{2.55}$$

where $g = \nabla_\theta \mathcal{L}_{\pi_\theta}(\pi_{\theta'})$ and $H$ is the Hessian matrix of the trust-region function.

### 2.4.6 Proximal Policy Optimization (PPO)

In reality, the natural gradient is computationally expensive and can be replaced by regularized and clipped versions of the optimization function using a method called proximal policy optimization (PPO) [36]. The first approach defines a new objective $\mathcal{L}^{KLPEN}_{\pi_\theta}(\pi_{\theta'})$ where the original objective is regularized by penalizing the trust-region constraint.

$$\mathcal{L}^{KLPEN}_{\pi_\theta}(\pi_{\theta'}) = \mathcal{L}_{\pi_\theta}(\pi_{\theta'}) - \lambda \mathbb{E}_{s\sim\rho_{\pi_\theta}}\left[D_{KL}(\pi_\theta \| \pi_{\theta'})\right] \tag{2.56}$$

This is done to ensure that new policies tend to stay closer to the old distribution. The regularization coefficient $\lambda$ also adapts according to the constraint depending on if it meets a particular target. The update of this term is defined by

$$\lambda = \begin{cases} \frac{\lambda}{b} & \text{if } d < \frac{d_{target}}{a} \\ \lambda b & \text{if } d > a \times d_{target} \end{cases}, \tag{2.57}$$

where $a$ and $b$ are hyperparameters typically defined as 1.5 and 2, respectively, $d$ is the trust-region constraint, and $d_{target}$ is the target value for the expected KL divergence. While $\mathcal{L}_{\pi_\theta}^{KLPEN}(\pi_{\theta'})$ offers an ease-of-use advantage over TRPO, the authors of the original PPO paper noted that $\mathcal{L}_{\pi_\theta}^{CLIP}(\pi_{\theta'})$ performs better. The clipped surrogate objective does not rely on any penalty term for the optimization function; rather, it provides a range for the probability ratio between $\pi_{\theta'}(A_t|S_t)$ and $\pi_\theta(A_t|S_t)$ which in turn scales the advantage function. The objective function can be formally defined as

$$\mathcal{L}_{\pi_\theta}^{CLIP}(\pi_{\theta'}) = \mathbb{E}_{\tau\sim\pi_{\theta'}}\big[\min(r_t(\theta)A^{\pi_\theta}(S_t,A_t), \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon))A^{\pi_\theta}(S_t,A_t)\big], \tag{2.58}$$

where $r_t(\theta) = \frac{\pi_{\theta'}(A_t|S_t)}{\pi_\theta(A_t|S_t)}$. Clipping the ratio acts as another way of penalizing a policy update that takes too big a step.

Recently, popularity among recurrent PPO algorithms has increased due to the supposed increase in performance. One common implementation of this type of algorithm is PPO-LSTM which includes a layer of LSTM nodes stacked before or after the actor/critic networks. This way, the hidden layers will adjust their weights according to the state observations as well as the weights of the hidden layers themselves.

All models discussed to this point have their strengths and weaknesses in various use cases. With all algorithms considered, variants of PPO will be evaluated due to their ability to handle continuous action and observation spaces, stability in dynamic reward structures, and ease of implementation. PPO also works well with simulation RL tasks due to the dense reward structure and lax requirements of sample efficiency. Therefore, PPO and PPO-LSTM will be used as the DRL algorithms for velocity control of the crawler.

# CHAPTER 3

# RELATED WORKS

The control of dynamical systems, or systems that can be defined by some ordinary differential equations, is a common problem that has been studied extensively in the field of robotics. Control theory concerns the development of models that map inputs to outputs governed by a set of feedback laws. This provides a way to create predictive models using data gathered from an environment. The methods of control theory are typically divided into two subdomains: classical and modern control. This chapter will explore previous research related to classical control, modern control, and reinforcement learning control with an emphasis on velocity control and differential drive robots.

## 3.1 Classical Control

Classical control refers to the methods of controlling a closed-loop system, or plant, where inputs are determined by the sensor feedback outputs. The controller of the system calculates an error between the target and actual outputs and tries to minimize that error in order to match a desired signal, or reference. Classical control methods operate on the assumption that the system is linear, meaning the additive response of two inputs can be achieved by applying each input individually, and time-invariant, meaning the output of the system will be identically no matter when the input was applied. The most common method in this subdomain is the proportional-integral-derivative (PID) controller. The PID controller is model-free and requires no knowledge of the system. As the name suggests, there are three components: a proportional, integral, and derivative part. If the error term is defined by $e(t)$ at a time step $t$, then the output can be given by

$$u(t) = K_P e(t) + K_I \int e(t)dt + K_D \frac{d}{dt} e(t), \tag{3.1}$$

where $K_P$, $K_I$, and $K_D$ represent tunable gain parameters for each term. The proportional term drives the output higher as the error term increases, and vice versa. The derivative term acts as a damping term and can prevent overshoot from the proportional term. The integral term exists to eliminate bias that may exist in the system [37].

PID control has been used to control state in differential drive robots. Trajectory control, or the control of a mobile robot to track a path over time, has been implemented using nonlinear and fuzzy PID controllers [38, 39]. Cruise control functionality in many automobiles use variants of PID controllers to maintain velocity despite hills, curves, or bumps in the road [40, 41]. PID control was also the basis of velocity tracking in the form of a steering strategy for a differential drive forklift [42].

## 3.2 Modern Control

Modern control theory, on the other hand, encompasses a wider range of problems than classical control theory. This theory caters to complex systems with multiple inputs and multiple outputs, where the time-invariant and linearity requirements do not hold. Some methods of modern control engineering include robust control, adaptive control, and optimal control. Robust control concerns models that are designed to handle uncertainty in a system. A linear-quadratic Gaussian motion-planning model [43] for DDRs has been designed to track trajectories with motion uncertainty and imperfect state information. Adaptive controllers maintain a steady state and guide actions to an optimal level by adjusting the parameters for the controller to compensate for system dynamics. A model-reference adaptive controller was designed for a DDR with a passive, lagging caster wheel to handle actuator and friction dynamics [44]. Optimal control strategies aim to maximize or minimize an objective function in relation to the dynamics of a system. The linear quadratic regulator (LQR) is an optimal control method where the system dynamics can be defined as a set of differential equations and the objective function is characterized by a quadratic. LQRs have also been used for DDR trajectory tracking [45].

## 3.3 Reinforcement Learning

RL also falls under the umbrella of modern control theory. It can be thought of as an optimal control method where the dynamics of the state are captured in its transition function and the objective is defined by the value function. Problems with well-defined MDPs could be solved using dynamic programming [12]. However, in complex environments, DNNs can be used as nonlinear function approximators to accommodate more states and actions, as seen in Section 2.4. A DRL model, specifically the deep deterministic policy gradient algorithm, was shown to outperform a non-RL modern control method in the velocity control of an automobile for adaptive cruise control [8]. DRL is also capable of disturbance, rejection control in nonlinear, uncertain, dynamical environments [46]. For robotic control, DRL models transferred from simulation to reality have also been proven effective. Robust locomotion control has been achieved on legged robots such as the Cassie bipedal robot [47] using a simulator with a gait-library plugin and the Anymal C for dynamic locomotion in unstructured environments [48]. The methods of experimentation in this thesis will expand on [48] for the velocity control of the Altiscan in a dynamic environment with disturbances.

# CHAPTER 4
# EXPERIMENTAL SETUP

This chapter presents the setup required to train a DRL velocity controller for the crawler. Section 4.1 covers the use of a novel simulation software called Isaac Gym. Section 4.2 and Section 4.3 provide specific implementation details for the physics and task parameters used in simulation. Variations of the reward function is given in Section 4.3.1 which will ultimately be used to update the parameters in the PPO model defined in Section 4.4. Finally, a framework to export a trained model to the real robot is given in Section 4.5.

## 4.1 Simulation Environment

For a DRL agent to learn, it must experience situations where it is tasked to make a decision. For robotic control tasks, this decision is typically described as the actuation of a particular set of joints. Experiences can be collected by recording data directly from a robot operating in the real world, but such scenarios are time-consuming and costly, particularly if a robot makes a mistake. To reduce the time and cost overhead of training a DRL agent, robotics researchers use physics simulators to build an approximate model of a real-world scenario so that the actors are able to fail freely while pursuing the trial-and-error learning style inherent to RL. Some popular physics simulators include PyBullet [49], MuJoCo [50], RaiSim [51], and Drake [52]. Many of these simulators perform calculations of contact physics entirely on the CPU, which becomes a performance bottleneck as the simulation environment grows increasingly complex. On the other hand, tensor-based DNN training can be accelerated on GPUs, which have a greater propensity for parallelization. However, context switching between CPU and GPU cores is inefficient because observation and action data must be passed between the two to complete the agent-environment interaction loop. Isaac Gym [53] is an end-to-end GPU accelerated contact physics sim-

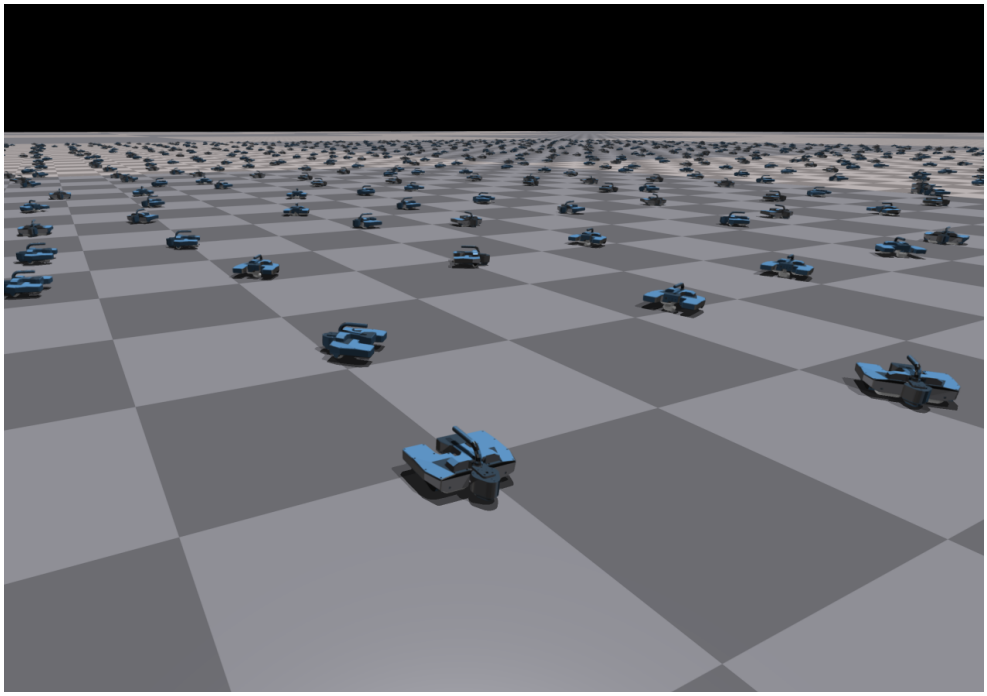


Figure 4.1: A picture of the parallelization capabilities of Isaac Gym. Multiple actors can be spawned in the same scene thanks to the parallel computation capabilities of GPUs.

ulator developed by NVIDIA™ that solves the data transfer problem by delegating the computation of observations, rewards, and actions entirely to the GPU. The speedup Isaac Gym provides for training models that achieve performant control reduces the real-world training time to an order of minutes. Isaac Gym provides substantial speedup for training models and is able to achieve performant control that reduces training time to an order of minutes. Isaac Gym is also capable of massive parallelization of actors in the same scene, as seen in Figure 4.1, as a result of the physics calculations occurring on the GPU.

Python is the language of choice for the Isaac Gym API, as it offers an abstraction over the core C++ and CUDA backend. Raw data buffers provided from the simulation are wrapped in tensors from the machine learning framework PyTorch [54] which aid in the speedup as they can be stored on either the CPU or GPU. Robots in Isaac Gym can be modeled using the Unified Robot Description Format (URDF). URDF models consist of a chain of links, or rigid bodies, connected by joints, degrees of freedom (DOFs) that can be fixed, revolute, prismatic, or spherical. The API provides state tensors consisting of a

position, orientation (in quaternions), linear velocity, and angular velocity for every actor root and rigid body in the scene. The position and velocity of every joint is also readily available through a DOF state tensor. For revolute joints, the velocity is given in terms of radians per second, and for prismatic joints, the velocity is given in terms of meters per second. In order to manipulate an actor, physics control tensors are provided. The DOF control tensors can be separated into three distinct methods: actuation force, positional, and velocity. The first DOF control method, effort control, applies a torque or linear force to the joint. The second method, positional control, uses a PD controller to move a joint to a given position. The last method, velocity control, also uses a PD controller to adjust the forces applied to the joint in order to reach a certain linear or angular velocity. The API also provides an interface to apply forces and torques to rigid bodies relative to their center of mass. Isaac Gym is built on top of NVIDIA's PhysX physics engine which relies on a Temporal Gauss-Siedel (TGS) solver [55]. The tunable parameters for this solver can be found in Table 4.1.

## 4.2 Scene Setup

A scene can be described as a simulation setup where actors interact with each other. An actor is a controllable or uncontrollable object for which contact physics can be calculated. It should be noted that the use of actor here differs from the use of actor in DRL methods in DRL. In the upcoming experiments, the actors are individual instances of the Altiscan crawler, spawned using the assets of the crawler which are proprietary URDF and mesh models provided by Roboplanet. The model includes a base link and base frame that act as parent links to two coaxial front wheels, defined as cylindrical collision meshes with a length of 5.5cm and a radius of 2.5cm. Each wheel is offset in the $y$-axis by 7cm and in the $z$-axis by 3cm. A revolute joint oriented on the $y$-axis represents the motors for the front wheels. The rear wheel is a caster wheel mechanism simulated by a revolute joint oriented along the $z$-axis for the base or swivel and a revolute joint oriented along the $y$-axis for the

Table 4.1: This table, adapted from [53], represents the tunable physics parameters for Isaac Gym available to the user.

| **Parameter** | **Description** |
|---|---|
| Time Delta (dt) | Controls time-step size. |
| Substeps | Number of physics iterations between each time step. |
| Gravity | Controls the gravity in the scene. |
| Collision Filtering | Filters collisions between shapes. |
| Position Iterations | Biased (velocity + positional error correcting solver iterations). |
| Velocity Iterations | Unbiased (velocity error only correcting) solver iterations. |
| Max Bias Coefficient | Limits the magnitude of position error bias friction. |
| Restitution | Controls bounce. |
| Static/Dynamic Friction | Static and dynamic friction coefficients. |
| Bounce Threshold | Relative normal velocity limit below which restitution is ignored. |
| Rest Offset | Distance at which shapes are held separated. Default is 0 but can be increased to hold objects at gap. Useful for thin objects. |
| Friction Offset Threshold | Distance at which friction anchors are discarded (static friction depends on friction anchor caching). |
| Solver Offset Slop | An $\epsilon$ value used to correct for round-off errors in contact gen. Corrects small skew effects with rolling spheres or capsules. |
| Friction Correlation Distance | Distance at which contacts are merged into a single friction constraint. |
| Max Force | Per-body and per-contact force limits. |
| Drive Stiffness | Positional error correction coefficient of a PD controller. |
| Drive Damping | Velocity error correction coefficient of a PD controller. |
| Joint Friction | Per joint frictional term. Simulates dry friction in a joint. |
| Joint Armature | Per joint armature term. Simulates motor intertia. |
| Body/Link Damping | World-space linear/angular damping on each body/link. |
| Max Velocity | Linear/angular velocity limits per body. |

wheel itself. The swivel is offset by 13cm along the $x$-axis behind the center of the crawler. The caster wheel is defined by a slightly smaller cylindrical collision mesh with a length of 4cm and a radius of 2cm with an arm offset of 2cm laterally and 3.5cm downwards in the $z$-direction. Only the front wheels are actuated, so in the DOF property parameters, the drive modes are set to velocity control with a maximum effort of 1000Nm. Since velocity control is determined by a PD controller in Isaac Gym, stiffness, the positional error coefficient, is set to 800 units and damping, the velocity error coefficient, is set to 200 units. The swivel and caster wheel joints are set as free-moving joints with stiffness and damping set to 0 units. Each actor is then spawned in its own environment, defined by an upper and lower boundary for its starting position. Actors can be tiled along the $xy$-plane in a grid-like fashion. For the experiments performed in this thesis, a total of 1024 actors are spawned in parallel. The ground plane can also be defined by controlling its normal vector. This could be useful for simulating walls or vertical surfaces, similar to surfaces best suited for the crawler to traverse. However, during sandbox testing of Isaac Gym for the crawler, inconsistencies in the physics computation occurred when using a ground plane normal to either the $x$- or $y$-planes. This may have been because Isaac Gym does not allow for tiling actors in the $z$-direction; when spawning a large number of actors along a vertical surface, their initial positions extend past the size of the ground plane, which may cause computational issues. To solve the problem of simulating a vertical plane, a horizontal plane with a modified gravity vector is used. Instead of the gravity vector pointing in the $-z$-direction in the global frame, it points in the $-y$-direction. To simulate magnetism, a force could be applied directly to the center of mass of the front and caster wheels in a direction normal to the surface; however, inconsistencies were found during testing that caused a drifting behavior in the crawler when applying forces directly to the wheels. Instead, a global force of 135N normal to the surface is applied generally across all crawlers in the scene. More rigorous testing is necessary to conclude the true cause of these inconsistencies.

### 4.3 Task Setup

In order for an actor to observe and act, a task must include a workflow to initialize an actor, step through simulation, and end cleanly. OpenAI provides an RL toolkit called Gym [56] that neatly packages the previous framework into an easy-to-use Python library. While this library is universally applicable, it has been specifically adapted to Isaac Gym in a repository called IsaacGymEnvs[1].

In order to create a velocity controller for the Altiscan crawler derived from a DRL model, a continuous observation space $\mathcal{S}$ and action space $\mathcal{A}$ must be defined. The observation space consists only of the measured base linear $x$-velocity ($\hat{v}$) and angular $z$-velocity ($\hat{\omega}$) of the crawler and its corresponding commanded velocity ($v, \omega$). The action space will provide outputs for the angular velocity control of the left and right wheels ($\dot{\phi}_{left}, \dot{\phi}_{right}$).

$$\mathcal{S} = \begin{bmatrix} \hat{v} \\ \hat{\omega} \\ v \\ \omega \end{bmatrix} \qquad \mathcal{A} = \begin{bmatrix} \dot{\phi}_{left} \\ \dot{\phi}_{right} \end{bmatrix}$$

In simulation, the base velocity can be found by transforming the actor root velocity vector into the actor's base frame orientation. The commanded velocity was given by sampling random linear velocity commands between $[-0.2, 0.2]$m/s and angular velocity commands between $[-1.0, 1.0]$rad/s. Gaussian noise and sinusoidal bias was added to the observed base velocities to emulate noisy sensor data. The standard deviation of the noise and period/frequency of the bias term are tunable. The action space outputs a continuous value between $[-1.0, 1.0]$ which can were scaled to apply an appropriate angular wheel velocity. Each episode lasted 3000 time-steps, which equates to 50 simulated seconds for a simulation running at 20Hz with 30 physics sub-steps. New commands were sampled every

[1] https://github.com/NVIDIA-Omniverse/IsaacGymEnvs

250 time-steps to encourage the RL agent to generalize. Each crawler was also initialized with a random heading direction and a random swivel position to increase the number of possible states observed and encourage robust control.

### 4.3.1 Reward Function

One of the many challenges in RL is defining a reward function that efficiently guides an agent towards some desired behavior. For the crawler, that desired behavior is accurately tracking some commanded linear and angular velocity. In other words, the goal was to minimize the error between the target and measured velocities. A few different reward structures were considered for the experiments.

The simplest reward structure considered was the sum of the absolute value between the target velocities and the measured velocities subtracted by 1, as seen in Equation 4.1.

$$r = 1 - (|\hat{v} - v| + |\hat{\omega} - \omega|) \tag{4.1}$$

However, this absolute value reward structure suffers from scaling issues when trying to find the error between velocities of different magnitude. To solve this, the error can be divided by the target to give a relative error that scales with the magnitude of the target velocity. An issue arises when dealing with 0 velocities, which is solved by eliminating the scaling for the special case. For example, the reward term for linear velocity would look like:

$$r_v = \begin{cases} 1 - |\frac{\hat{v}-v}{v}| & \text{if } v \neq 0 \\ 1 - |\hat{v} - v| & \text{if } v = 0 \end{cases}.$$

The reward term for angular velocity $r_\omega$ would look similar, where $r_v$ and $r_\omega$ are the summands for the overall reward $r$ . During preliminary sandboxing, the scaled absolute value reward function failed to learn a model that could sufficiently control the crawler, perhaps due to the flatness of the absolute value function. In this reward structure, as the agent

learns to track the commanded velocities, a change in error that minimizes the total error will result in the same increase in reward as a change in error that maximizes it. For velocity tracking, it is crucial that the agent is rewarded at a greater scale for small errors and actions that minimize the error produce a greater magnitude of reward than actions that increase the error.

The next reward structure considered was the Gaussian function. Gaussian functions take the form of a bell curve that peaks at an input of 0 and tapers off as the input approaches $\infty$ or $-\infty$. In this case, the input is treated as a velocity error. Two Gaussians define the reward for linear and angular velocity. The error terms are scaled by their respective variance values: $\sigma_v^2$ and $\sigma_\omega^2$. Here, the product of partial rewards is used instead of the sum. By multiplying the Gaussians together, a larger reward is produced at a particular timestep when the action taken minimizes both the error in linear and angular velocity. Equation 4.2 gives a formal definition of this Gaussian reward structure.

$$r = e^{\frac{(\hat{v}-v)^2}{\sigma_v^2}} \cdot e^{\frac{(\hat{\omega}-\omega)^2}{\sigma_\omega^2}} \tag{4.2}$$

While this new reward structure characterizes the improvements sought after from the scaled absolute value reward, learning also stalled during sandbox testing. The problem with the Gaussian reward structure is that its steepness, dictated by the variance, also controls the rate at which the bell curve tapers off. For large errors, the reward is minimal; actions that provide small, incremental increases in reward do not influence the agent enough to learn better controls. However, the steepness of the Gaussian is still desired because it rewards finer, optimal control. This leads to the final reward structure used for experimentation: the Gaussian mixture reward. The Gaussian mixture reward provides a path for poor models to perform better by increasing the change in reward for decreasing errors with a relatively large magnitude, while preserving the steepness around an error of 0. This is accomplished by summing two weighted Gaussians with a scaled variance. Formally, the

blueprint of the Gaussian mixture reward structure is:

$$r_v = \alpha e^{\frac{(\hat{v}-v)^2}{\sigma_v^2}} + (1-\alpha)e^{\frac{(\hat{v}-v)^2}{100\sigma_v^2}}$$
$$r_\omega = \alpha e^{\frac{(\hat{\omega}-\omega)^2}{\sigma_\omega^2}} + (1-\alpha)e^{\frac{(\hat{\omega}-\omega)^2}{100\sigma_\omega^2}}$$
$$r = r_v \cdot r_\omega. \tag{4.3}$$

Here, $\alpha$ represents the weighting factor that affects the mixing between the two Gaussians, and $r_v$ and $r_\omega$ represent the reward terms for linear and angular velocity. The right-hand summand in each term adds a slight incline to the bell curve as the error approaches 0 for larger error values. The graph in Figure 4.2 offers a visual comparison between the Gaussian reward structure to the Gaussian mixture reward structure. This reward structure produced the best results during sandbox testing, and was used for further testing. The hyperparameter values used are $\alpha = 0.9$, $\sigma_v^2 = 0.001$, and $\sigma_\omega^2 = 0.01$.

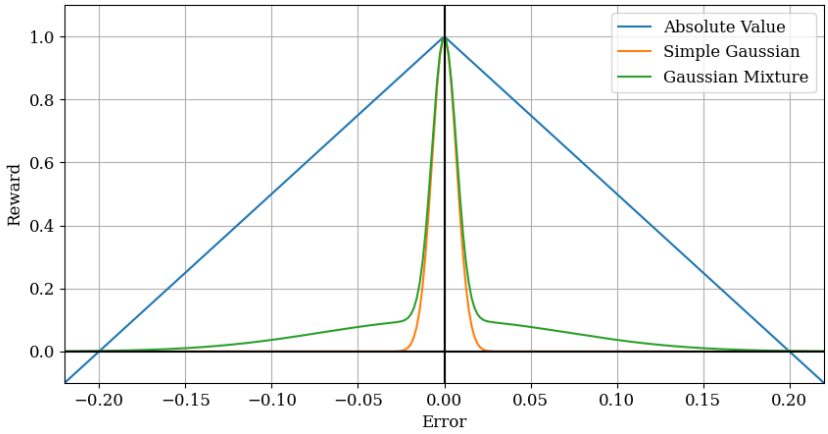



Figure 4.2: A comparison of the reward functions considered. Notice how the simple Gaussian reward function has less of a taper as the error increases compared to the Gaussian mixture reward function. Also, the absolute value reward uses a different reward scale, since the negative reward is unbounded.

## 4.4 Agent Training

As referenced in Section 2.4.6, PPO and PPO-LSTM were chosen as the DRL algorithms for the velocity control of the crawler. To use PPO with Isaac Gym, a single model must take in observations from each of the parallel actors in simulation and produce a respective action for each one. For this an asynchronous advantage actor-critic (A3C) [57] architecture can be used to consolidate the learning to an individual model while distributing the agent/environment interaction to parallel workers. Isaac Gym also claims end-to-end GPU-based learning, so any A3C with a PPO extension to it used with Isaac Gym must allow GPU acceleration. The algorithms used in this thesis are from rl-games [58] which provides an open-source implementation of a PyTorch-based DRL library and has been used in many DRL contexts with real-world transferability [53, 59, 60, 61]. The RL framework IsaacGymEnvs also allows for full interoperability with rl-games with ease of use by providing hyperparameter tuning through the open-source framework Hydra [62]. A full list of the hyperparameters used can be found in Appendix B.

The DNN architectures were not evaluated; rather, the architectures are borrowed from other models that have shown success in Isaac Gym. For the PPO algorithm, a DNN was used as a function approximator for the policy distribution used to produce actions from input observations. The DNN architecture for PPO, presented in Table 4.2, consists of 1 input layer, 3 hidden layers, and 1 output layer. The input layer received a vector of 4 continuous observations: the measured and target linear and angular velocities. The first and second hidden layers contain 256 nodes, while the last layer contains 128 nodes. Each node uses a ReLU activation function to propagate its signal. The output layer is described by the continuous action space vector of size 2. This output is clipped between -1 and 1 and later scaled to the appropriate left and right wheel velocities.

Rl-games also allows for RNN implementations such as LSTM and gated recurrent networks (GRUs). As mentioned in Section 2.3.5, RNNs are better at predicting outcomes

Table 4.2: This table shows the DNN architecture that supports the actor and critic for the implementation in rl-games.

| Layer | Type | Activation | Size |
|---|---|---|---|
| 1 | Input | - | 4 |
| 2 | Fully-Connected | ReLU | 256 |
| 3 | Fully-Connected | ReLU | 256 |
| 4 | Fully-Connected | ReLU | 128 |
| 5 | Fully-Connected | Linear | 2 |

Table 4.3: This table shows the PPO-LSTM DNN architecture that supports the actor and critic for the implementation in rl-games.

| Layer | Type | Activation | Size |
|---|---|---|---|
| 1 | Input | - | 4 |
| 2 | Fully-Connected | ReLU | 256 |
| 3 | Fully-Connected | ReLU | 256 |
| 4 | Fully-Connected | ReLU | 128 |
| 5 | LSTM | ReLU | 128 |
| 5 | Fully-Connected | Linear | 2 |

of input data that are sequential in nature. The LSTM networks introduced in Section 2.3.6 improve upon RNNs and allow for more stable gradient descent. PPO-LSTM incorporates an LSTM network into the DNN that serves the actor and critic. The architecture backing the PPO-LSTM algorithm used in this thesis can be found in Table 4.3. A single LSTM layer of size 256 is stacked before the output of the DNN, while the rest of the DNN remains the same. The efficacy of these algorithms will be evaluated in Chapter 5 and Chapter 6.

## 4.5 Sim-to-Real Transfer

After training the agent, the model would have been configured with a certain set of weights for an optimal policy. However, in this case, all training was done in simulation. True evaluation of the model must be grounded in the real world with the physical robot. Bridging the gap to reality by porting the knowledge gained during simulation is called *sim-to-real transfer* [63]. The model creation provided by rl-games relies on PyTorch and the weights are defined by .PTH files. These can be reconstructed to be used directly on the crawler;

however, Python is relatively computationally heavy in comparison to C++. To export the model to C++, a framework called Open Neural Network Exchange (ONNX) [64] can be used.

# CHAPTER 5
# EVALUATION METHODS

With the simulation environment, task, and DRL algorithm primed for training, the specific evaluation metrics and cases can be detailed. Since DRL has not been applied to the velocity control of ROBOPLANET Altiscan before, this methodology serves as a proof-of-concept for its effectiveness. The scenes modeled in simulation increase in complexity to highlight the strengths and weakness of the DRL velocity controller. This chapter will expand upon the specifics of the environment setups in Isaac Gym. First, an ideal environment will be evaluated. This ideal environment relies on the contact physics and friction models from PhysX to simulate crawler motion on a flat, horizontal plane with gravity acting as the sole external factor. Next, noise will be added to the observations to test the robustness of the controller. The crawler will then be evaluated on a vertical surface with a constant force of gravity applied parallel to the surface. Finally, simulated tether forces will be applied to the base of the crawler to test the crawler's ability to withstand nonlinear external disturbances.

## 5.1 Training Performance

The first stage in model creation is the training process. Proper evaluation during training can help the end user determine the effectiveness of the controller without waiting until the end of the entire process. Metrics such as loss in the actor and critic networks can provide insights to model performance; however, these values can sometimes be difficult to interpret and do not necessarily pertain to the defined task. Fortunately, reward is a task-specific signal calculated per timestep that is available to track during the training process. Rl-games is integrated with a machine learning experiment-tracking suite called Weights & Biases [65] which provides a dashboard interface to monitor statistics such as actor-critic

loss, rewards per timestep, and even system data like GPU temperature. Here, Weights & Biases is used to track the reward statistic over the course of training. At each timestep, every actor in simulation computes a reward based on the Gaussian mixture reward given in Equation 4.3 for the corresponding action given by the model. The average of this reward across all actors provides a broad overview of the performance of a model. The range of the Gaussian mixture reward ranges from 0 (exclusive) to 1 (inclusive). A reward closer to 0 indicates poor performance because it relates to a larger error. A reward of 1 indicates optimal velocity tracking because in order to reach this value, the error between the target and measured velocities must be 0. As a preliminary quantitative check to determine the potential of a model, the reward per timestep metric will be used to evaluate velocity control among different environments and between PPO and PPO-LSTM.

## 5.2 Runtime Performance

Isaac Gym promises significant speed-up compared to other simulators by using the massive parallelization capabilities of an end-to-end GPU pipeline. To evaluate the runtime performance of the training process, the total elapsed time was recorded. However, it should be noted that since the task for velocity control was not episode-length dependent, training always ended at a pre-specified number of timesteps. Since the computational load during training for different environments or algorithms does not significantly affect runtime, it is more notable to compare the performance of Isaac Gym to that of another simulator. A mirror simulation with a similar environment, task, and reward setup was constructed in PyBullet [49], a CPU-based contact physics engine built atop the Bullet physics backend. The model of the Altiscan crawler was reused, though the tunable physics parameters differ greatly between the two physics engines. Timestep and substep parameters remained the same while most others were not exposed through by Bullet's Python API. The simulation scene in PyBullet only supports one actor. Another difference exists in the implementation of the PPO DRL algorithm.

While Isaac Gym relies on IsaacGymEnvs and rl-games for its RL framework, PyBullet works nicely with the standard OpenAI Gym API and SEED RL [66]. SEED RL is an asynchronous DRL library that allows for the scalable distribution of actors/simulation environments across multiple machines with a centralized learner. SEED RL provides multiple DRL algorithms based on Tensorflow [67], but only the PPO implementation will be used. Runtime performance evaluation will be conducted for the velocity control of the crawler for simulations implemented in PyBullet[1] and Isaac Gym[2]. The scene consisted only of the basic, ideal, horizontal plane, and the task only concerned optimal velocity tracking without regard for robustness. The Isaac Gym-rl-games tandem ran on a machine with an NVIDIA RTX 3090 GPU, AMD Ryzen 9 3900X 12-core processor, and 128GB of DDR4 RAM. The learner and actors in SEED RL were distributed in Docker [68] containers across 4 machines with an NVIDIA Quadro P400 GPU, 16-core Intel i7-11700 processor, and 16GB of RAM. The efficiency of the two simulators was based on the maximum reward per timestep achieved and total training time.

### 5.3 Robustness Evaluation

For DRL models to be deployed on real-world systems, they must be robust enough to handle the natural complexities of their environment. The natural environment of the Altiscan consists of noisy localization measurements, constant and stochastic forces from gravity and the tether, and motion artifacts due to initial conditions. Modeling these situations in simulation provides more observations to the agent which enable it to extrapolate actions for states it has never seen. The metrics used for robustness evaluation will rely on the root mean-squared error (RMSE), a measure of a model's predictive capabilities, between the target velocity and measured velocity for an entire trajectory. These will then be averaged across target linear and angular velocities or environments.

[1] https://github.com/devarsi-rawal/crawler_gym
[2] https://github.com/devarsi-rawal/CrawlerGymIsaac

### 5.3.1 Environment Robustness

Four environments of increasing complexity were evaluated in this research. Velocity tracking in these environments was evaluated in comparison to the inverse velocity kinematics model given in Section 2.1.

*Ideal Environment*

The ideal environment is the most basic environment to be evaluated. The goal of learning in this environment is to show that velocity control of a differential drive robot is possible in simulation. The ideal environment consists of a flat, horizontal ground plane with exact observations (considering some simulation noise) and no external disturbances.

*Noisy Environment*

Self-localization of the Altiscan crawler relies on a particle filter to calculate position and velocity estimates. Particle filters are known to have an element of uncertainty due in part to the noisy IMU, UWB, and odometry sensor readings and constant resampling of particles. However, Isaac Gym provides accurate position and velocity readings through its exposed state-tensor API. As such, noise can still be introduced by abstracting a layer of stochasticity on top of the observations. While the target linear and angular velocities should be known since they are applied as inputs from a joystick, the other set of observations, measured base linear and angular velocities, can have simulated noise. This noise will be applied as a Gaussian distribution centered around the true measured velocity with a standard deviation respective to linear or angular velocity. A sinusoidal bias term is also included to simulate drifting behavior common in sensor readings. This environment is designed to evaluate the model's robustness to noisy observations.

*Noisy Environment with Constant Disturbance*

The defining feature of the crawler is its electromagnetic wheels which allow it to traverse vertical surfaces. As such, a constant disturbance is introduced as gravity no longer acts parallel to the $z$-axis of the base of the robot. In simulation, the forces on the crawler are applied using the universal gravity vector $[0.0, -9.81, -135]$. Notice that the vertical surface is simulated by changing the force of gravity to the $-y$-direction and applying the 135N magnetic force of the wheels globally in the $-z$-direction.

*Noisy Environment with Tether Disturbances*

The key question evaluated by this research was the velocity control of the crawler when influenced by the forces of a tether. This can be modeled in simulation by reusing the noisy environment with constant disturbance from before and adding a layer of complexity by simulating tether forces acting on the base of the crawler. This was done by defining the two anchor points of the tether. The first anchor is placed at the center of mass of the crawler and the second anchor placed at the global origin. The force direction vector points from the first anchor point to the second anchor point. This direction vector is normalized and scaled by a force magnitude value. Three methods of tether force application were explored:

- Constant tether force: At every timestep, a constant force is applied to the base of the crawler at its anchor point equal to the scaled force vector described before.
- Impulse tether force: Here, a sudden application of force to the crawler is simulated by increasing the magnitude of the force every 100 timesteps. This simulates situations where the tether may snag on protrusions present on the structure the crawler is inspecting.
- Sinusoidal tether force: A fluctuating tether force is also explored to simulate the free-moving nature of the suspended tether. The magnitude of the force applied to

the base of the crawler is scaled in a sinusoidal fashion at every timestep.

### 5.3.2 Robustness to Initial Conditions

The initial caster wheel swivel position and base orientation of the crawler can have an impact on the outcome of the trajectory it takes given certain commanded velocities. Though the caster wheel is assumed to be a passive wheel that is tangent to the crawler's trajectory and included for stabilization in DDRs, the forces of magnetism and mass attributed to the wheel make its affect on the system nonneglible. The initial orientation of the crawler can also impact its trajectory in environments with forces directed in a constant direction, such as those with vertical surface. To evaluate the effects of caster wheel position, the crawler will be initialized with four positions at $90^\circ$ increments and the trajectory of each will be recorded for a constant linear velocity of 0.1m/s. The effect of initial orientation on the crawler in a vertical setting will be evaluated by recording the trajectory and velocities at the same constant linear velocity for four orientations at $90^\circ$ increments, as well.

### 5.3.3 Model Resilience

Model resilience here will refer to the ability of the DRL model to adapt to environmental conditions it was never exposed to while training in simulation. Model resilience will be evaluated by testing velocity control in a noisy environment with constant tether disturbances using a model trained in an ideal environment.

# CHAPTER 6

# RESULTS

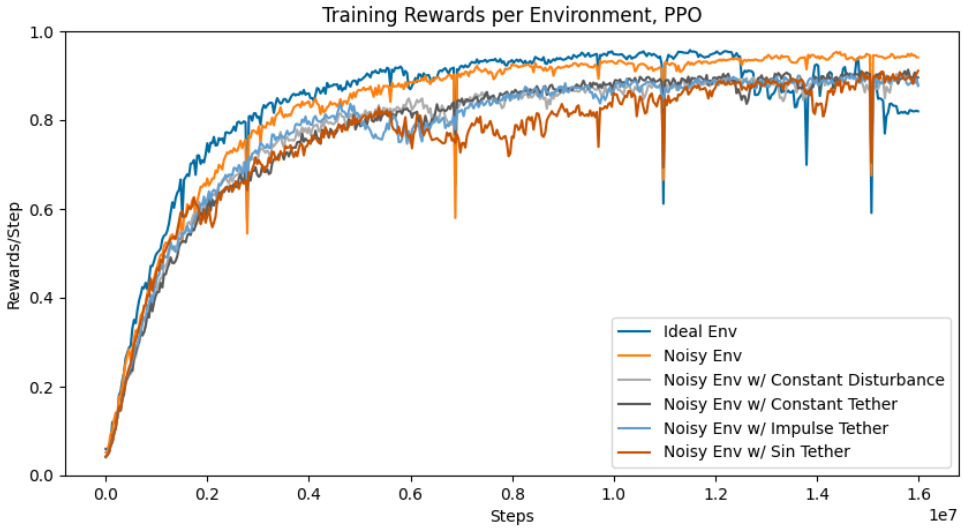
This section will explore the quantitative and qualitative results received from each test.

## 6.1 Training Performance

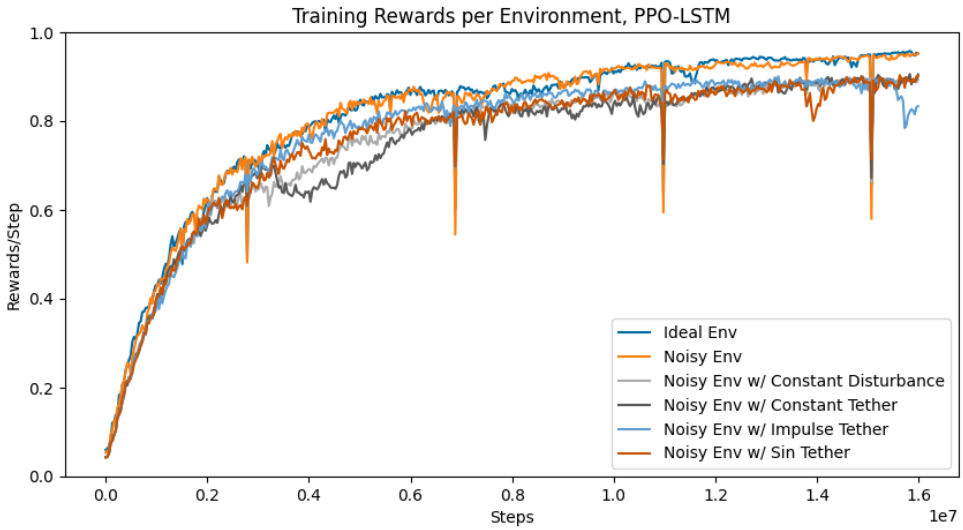
Training performance was evaluated by tracking the reward per timestep over the entire training period. 1024 actors were trained on a time horizon of 32 timesteps over 500 epochs, which gives about 16 million total training steps across all actors. Figure 6.1 shows the development of rewards across the entire training period, demonstrating how both PPO and PPO-LSTM increase performance in velocity tracking as training progresses. Table 6.1 provides detailed measures of performance by environment and algorithm. The optimal models were created in the ideal environment in which both PPO and PPO-LSTM achieve a reward of 0.958, while the next best model learned in the noisy environment which achieved a reward of 0.953 and 0.954 for the two algorithms. A notable drop in performance occurred, however, when introducing the simulated vertical surface and tether forces to the environment. PPO also learned better than PPO-LSTM for all of these external disturbances. The environment in which both models learned the worst was the environment with an impulse force in addition to gravity forces and noisy observation.

Table 6.1: Maximum Training Rewards.

| | PPO | PPO-LSTM |
|---|---|---|
| Ideal | 0.958 | 0.958 |
| Noisy | 0.953 | 0.954 |
| Noisy w/ Constant Disturbance | 0.908 | 0.898 |
| Noisy w/ Constant Tether Disturbance | 0.913 | 0.903 |
| Noisy w/ Impulse Tether Disturbance | 0.905 | 0.901 |
| Noisy w/ Sin Tether Disturbance | 0.911 | 0.906 |




(a) PPO




(b) PPO-LSTM

Figure 6.1: Training Rewards by Environment and PPO Variant

## 6.2 Runtime Performance

The runtime performance of training a DRL model using Isaac Gym and rl-games was evaluated against the distributed training with PyBullet and SEED RL for a noisy environment with constant tether disturbance. Both training methods ran for a total of about 16 million global timesteps. Table 6.2 provides the wall clock time, or real world time, to reach 16 million training timesteps. Isaac Gym ran significantly faster taking a wall clock time of 12 minutes and 10 seconds compared to PyBullet taking 17 hours, 3 minutes, and 16 seconds; however, it should be noted that Isaac Gym simulated 1024 actors while PyBullet had 4 distributed simualtions with 1 actor each. Figure 6.2 shows the reward per timestep between the two setups. The maximum reward per step achieved in Isaac Gym 0.913, whereas in PyBullet it reached a maximum of 0.681.

Table 6.2: Runtime performance.

| **Simulation Software** | **DRL Library** | **Training Time** |
|---|---|---|
| Isaac Gym | rl-games | 12m10s |
| PyBullet | SEED RL | 17h3m16s |

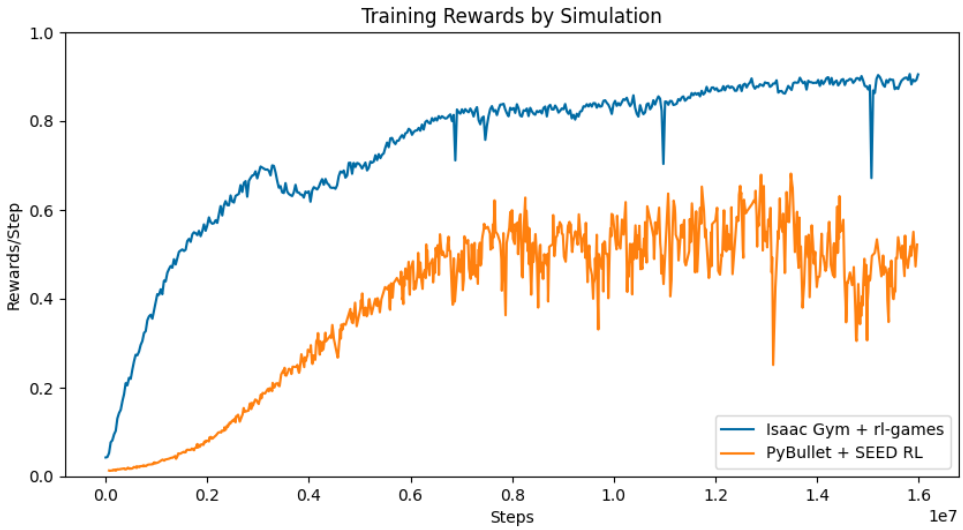
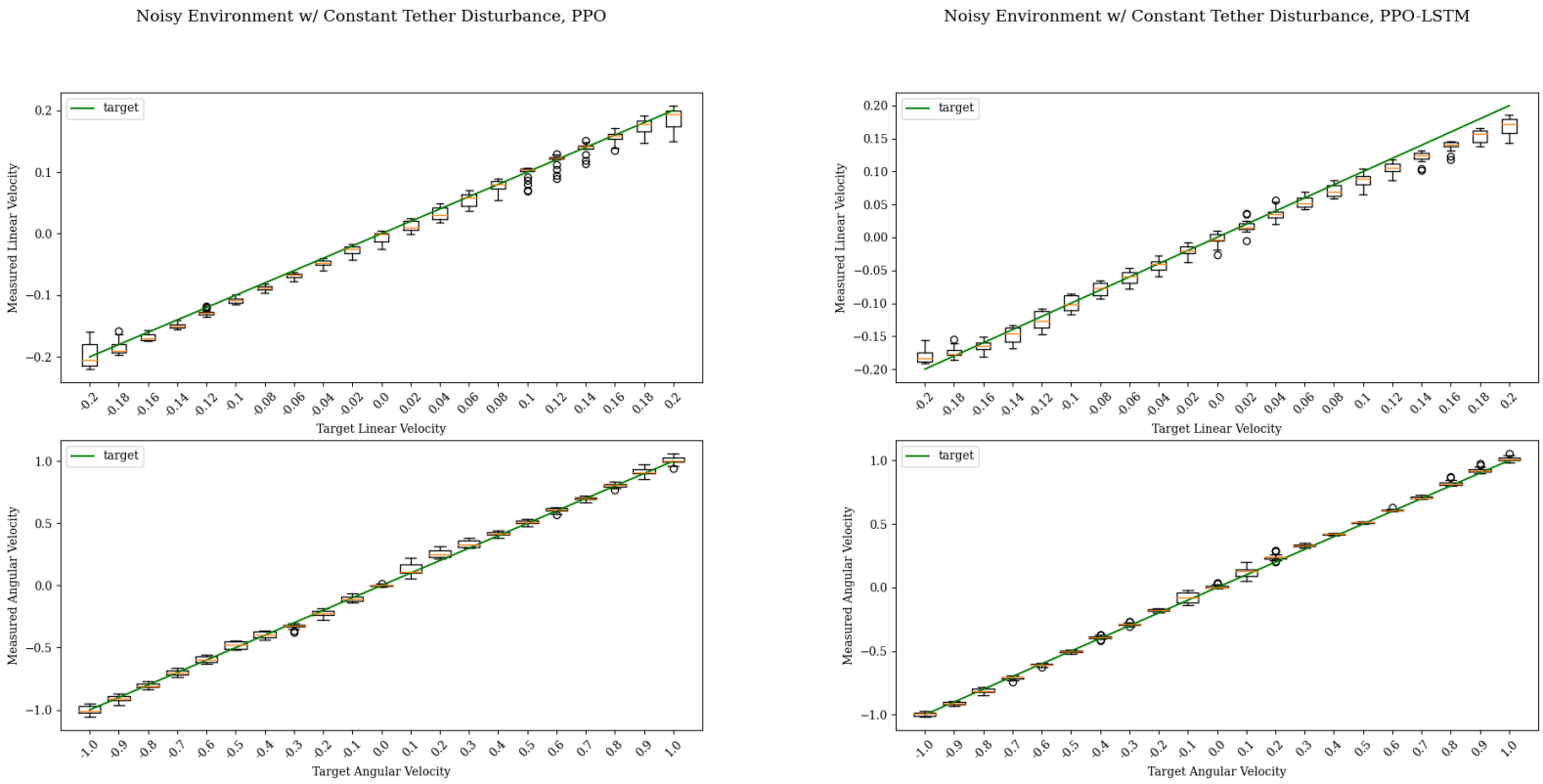



Figure 6.2: Training Reward for Isaac Gym and PyBullet

## 6.3 Robustness Evaluation

### 6.3.1 Environment Robustness

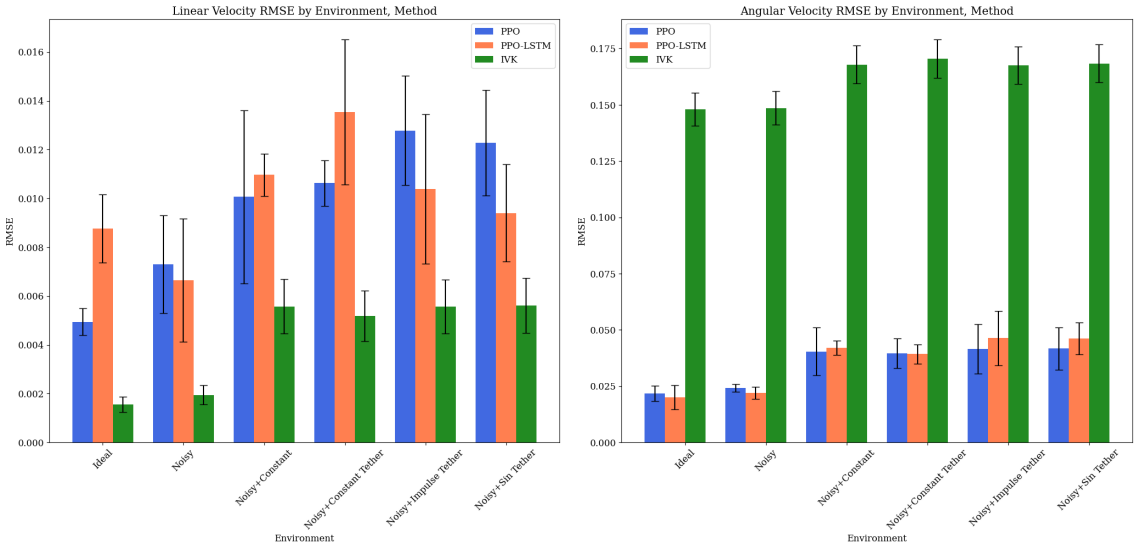
Six environments were evaluated to test a DRL model's ability to learn the dynamics of varying levels of external disturbances. Section 6.1 demonstrated that the models were able to learn velocity control generally over across environment. However, by analyzing the models in action, it is easier to highlight the strengths and weaknesses of velocity tracking by commanded velocity and by environment. For example, Figure 6.3a and Figure 6.3b show each PPO variant's ability to track linear and angularity velocity. There were 21 linear and 21 angular velocities evaluated by letting the crawler run for 200 timesteps, recorded after 50 steps to allow for stabilization. The mean linear and angular velocity was taken for each trajectory. Each target linear velocity captures a combination with all 21 angular velocities. In a noisy environment with a simulated vertical plane, both PPO and PPO-LSTM track velocity close to their target linear and angular velocities. Near the extremes of the commanded linear velocity range (-0.2m/s and 0.2m/s), the median measured linear velocity deviates from the target the most. This behavior is seen in both PPO and PPO-LSTM controllers.




(a) PPO

(b) PPO-LSTM

Figure 6.3: Linear and Angular Tracking per Velocity in a Noisy Environment with Constant Tether Disturbances




(a) Average Linear Velocity RMSE

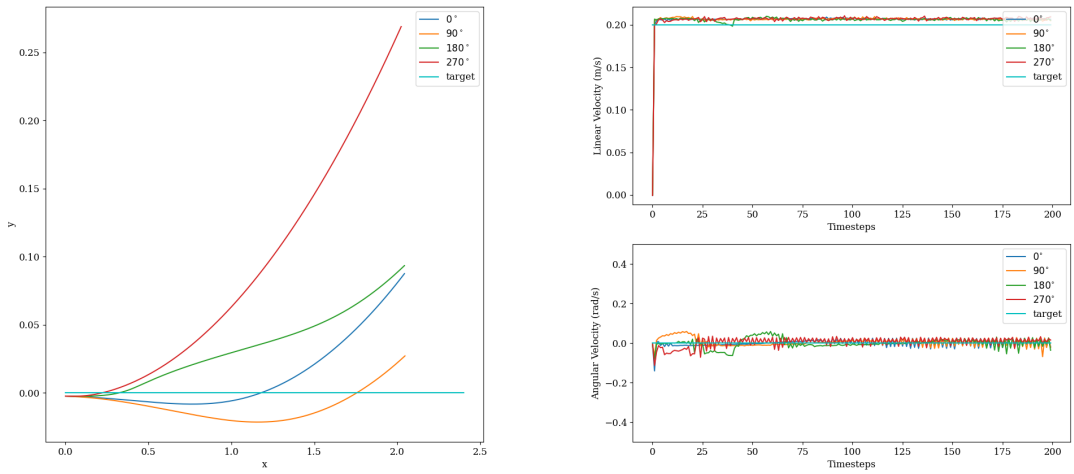
(b) Average Angular Velocity RMSE

Figure 6.4: Errors by Environment and PPO Variant

Figure 6.4 shows the average RMSE of linear and angular velocity across all six environments. Here, a comparison between the PPO, PPO-LSTM, and IVK velocity controllers is presented. The RMSE is calculated across the trajectory of the crawler and averaged over every linear or angular velocity, respective to the figure. The IVK gives an average linear RMSE of of 0.00371 across all environments, compared to 0.00833 for PPO and 0.00995 for PPO-LSTM. For angular velocity, the average RMSE was 0.0386 for PPO, 0.0399 for PPO-LSTM, and 0.162 for IVK across all environments.

### 6.3.2 Robustness to Initial Conditions

To evaluate the robustness to initial conditions, tests were performed with four caster wheel positions and four initial orientations in a noisy environment with constant tether disturbances. The four initial conditions were rotations, $0°$, $90°$, $180°$, and $270°$, about a central axis. For the caster wheel, $0°$ is the regular, lagging position along the $x$-axis of the crawler. $180°$ is also along the $x$-axis but is more unstable in forward motion. $90°$ and $270°$ are positions orthogonal to the trajectory of motion. Figure 6.5 shows the trajectory and velocity

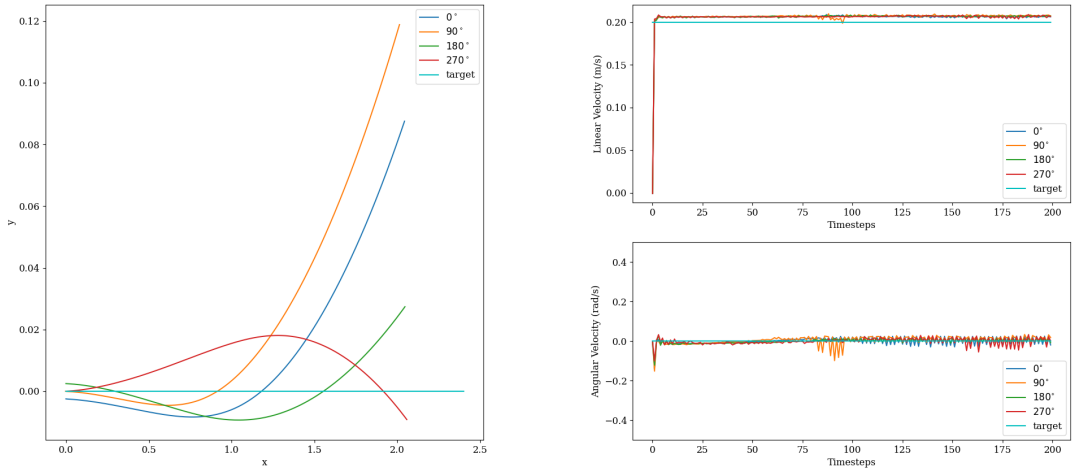



(a) Trajectory

(b) Velocities

Figure 6.5: Effect of Initial Caster Wheel Position on Crawler Trajectory and Velocity

tracking of the PPO model on 4 crawlers when given a linear velocity of 0.2m/s and no angular velocity. The measured linear velocity rises quickly past the target after which it stabilizes at a value around 0.21m/s. The measured angular velocity oscillates around a value of 0rad/s. Figure 6.6 shows the effect of orientation on crawler trajectory and velocity tracking the same environment. An initial orientation of $0°$ points in the positive $x$-direction. The following orientations turn in a clockwise rotation, where $90°$ points in the positive $y$-direction, $180°$ points in the negative $x$-direction, and finally $2700°$ points in the negative $y$-direction. For the simulated vertical surface, the force of gravity points in the negative $y$-direction. Similar to the position of the caster wheel, the velocity tracking amongst different orientations gives a stabilized linear velocity of 0.21m/s with oscillations in angular velocity around 0rad/s. The trajectories, however, deviate greatly from their target.




(a) Trajectory (b) Velocities

Figure 6.6: Effect of Initial Orientation on Crawler Trajectory and Velocity

### 6.3.3 Model Resilience

Model resilience was evaluated to measure a DRL model's ability to generalize velocity control in an environment never experienced in training. Specifically, a PPO model trained in an ideal environment was cross-evaluated in a noisy environment with a constant tether force and constant force of gravity. Figure 6.7 shows the velocity tracking for specific linear and angular velocities. Similar increases tracking errors occur near the bounds of the linear velocity as in the model with symmetric evaluation. Figure 6.8 gives a comparison between a symmetrically-evaluated model and the cross-evaluated model. The model trained in an ideal environment yields a lower linear velocity RMSE at 0.00913 than the symmetric model at 0.0120; however, for angular velocity, it yields an RMSE greater than the symmetric model at 0.0487 compared to 0.311.

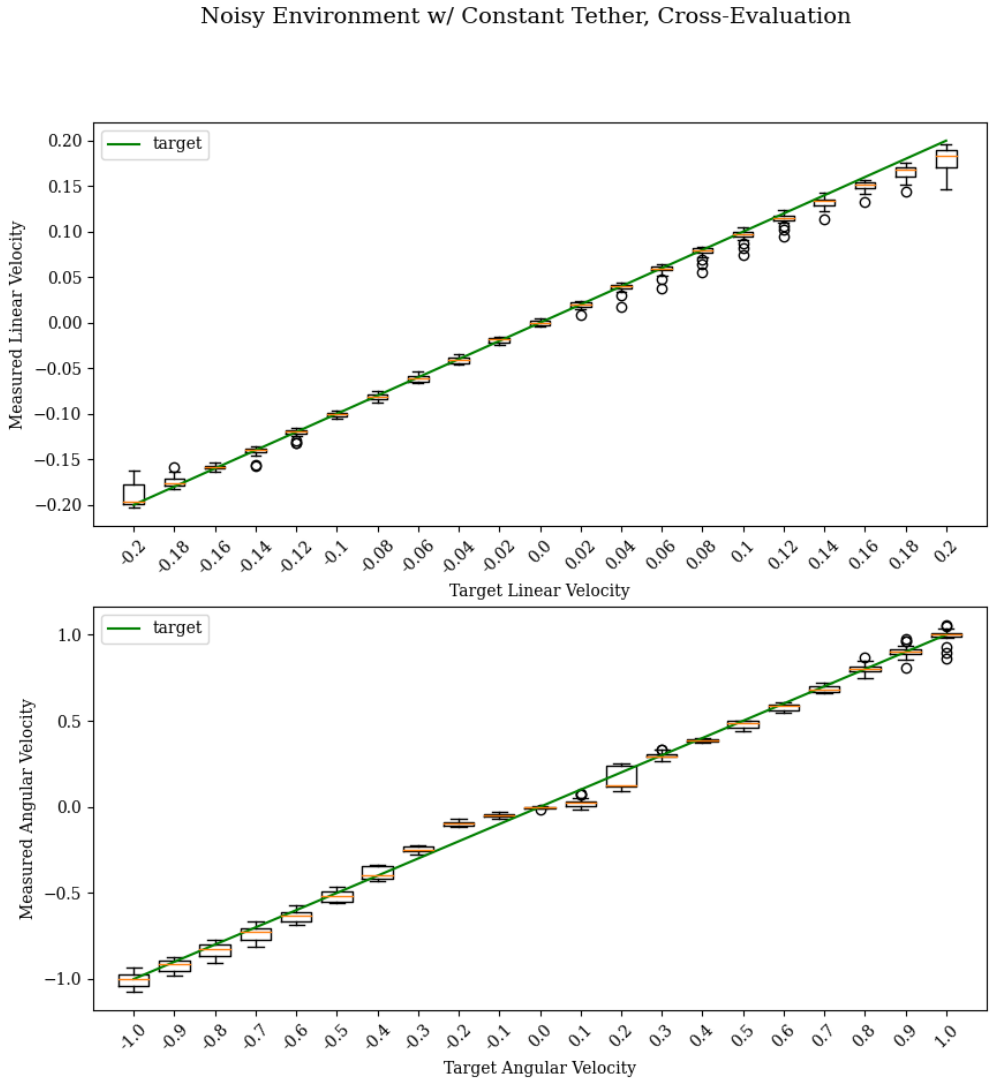



Figure 6.7: Cross evaluation of a PPO velocity control model. The PPO model was trained an in ideal environment and evaluated in noisy environment with constant tether disturbance.

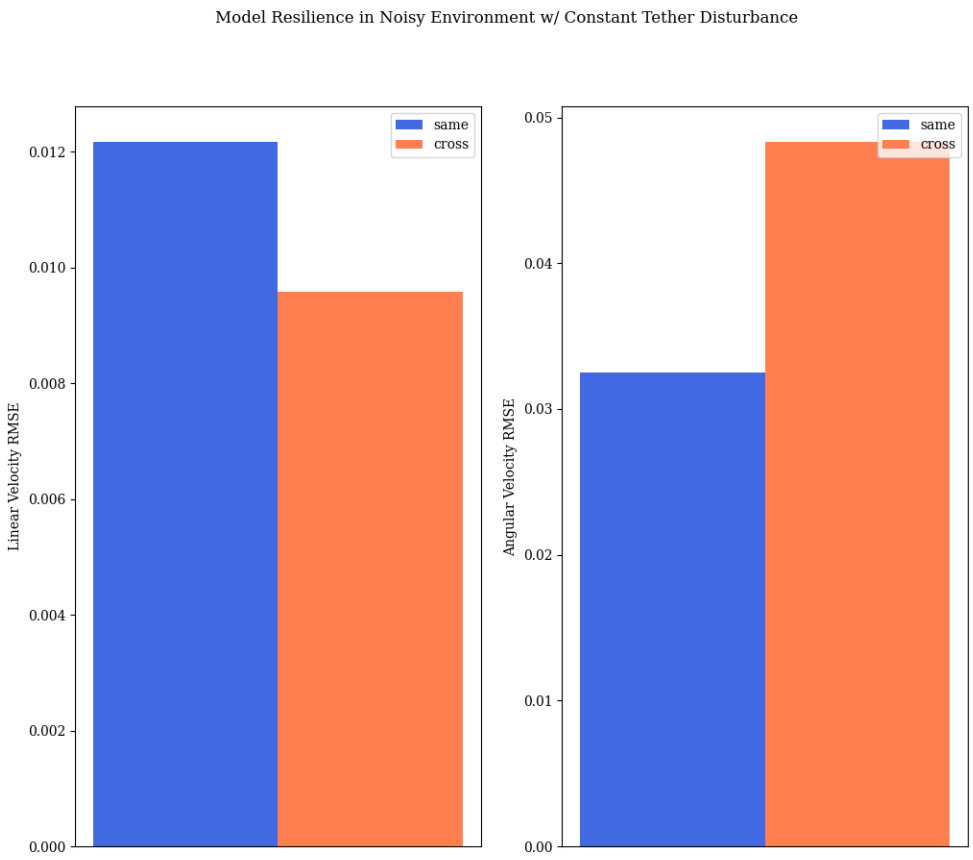



Figure 6.8: Cross-evaluated model errors compared to a symmetrically-evaluated model. ”Same” or symmetric refers to the model trained in the environment it was tested in. ”Cross” refers to the model trained in an ideal environment and tested elsewhere.

# CHAPTER 7
# DISCUSSION & CONCLUSION

Over the course of this thesis, deep reinforcement learning (DRL) velocity control of the ROBOPLANET Altiscan, a magnetic-wheeled, tethered differential-drive robot. PPO and PPO-LSTM were selected as two state-of-the-art DRL algorithms suitable for learning in simulation using the end-to-end GPU stack of Isaac Gym and rl-games.

First, it was shown that PPO and PPO-LSTM were able to learn the basic controls of a DDR. In Section 2.3.6, it was noted that LSTM networks have better predictive capabilities for sequential data than traditional DNNs and in Section 2.4.6, it was mentioned that PPO-LSTM could leverage these advantages by appending an LSTM layer to the actor-critic network. Surprisingly, PPO-LSTM performed worse in training than PPO as shown in Table 6.1, specifically in the environments with external disturbances. This fails to confirm the original hypothesis that PPO-LSTM would perform better when subject to forces of gravity and the tether, though tuning hyperparameters may give different results.

The second major finding was with regard to training speed performance. Table 6.2 shows the stark difference in real-world training time for the 16 million timesteps between the parallelized GPU pipeline with Isaac Gym and rl-games and the distributed CPU-based pipeline with PyBullet and SEED RL. However, the bias towards GPU training must be stated. First, the hardware used for GPU training far outclassed the processing power for CPU training. Second, 1024 actors were spawned in parallel in Isaac Gym compared to 4 actors across 4 instances of PyBullet. Given Figure 6.2, if we were to normalize reward by total wall clock time instead of total timesteps, the former duo would take an aggregated 204 hours to reach an optimal learning reward while the latter would take 68 hours. This means that Isaac Gym and rl-games is not necessarily more efficient in terms of learning time per actor than a similar CPU based setup. While PyBullet and SEED RL did not

quite reach the same maximum reward as its GPU-based counterpart, it is possible that training was stopped prematurely for comparison sake. This provides ample opportunity for further research. Some interesting tests could be to use the full capabilities of the CPU by distributing actors across multiple cores for each machine. This would allow the order of number of actors to more feasibly reach the parallelization capabilities of Isaac Gym. Deploying PyBullet and SEED RL on the AMD Ryzen 9 3900X 12-core processor may also provide better comparisons.

The third major finding considers the robustness of PPO variants for velocity tracking in environments of increasing complexity and its comparison to an IVK controller. The results found in Figure 6.3a and Figure 6.3b showed that velocity tracking near non-boundary target linear and angular velocities performs fairly well with minimal spread for both PPO and PPO-LSTM. However, larger interquartile ranges occur near the boundaries. The reason for this is because the maximum velocity of the crawler is 0.2m/s. This means that both actuated wheels must turn at a rate of 8rad/s to achieve this target. However, adding any angular velocity while maintaining this linear velocity means keeping the same speed in one wheel while increasing the speed in the second wheel. This is rendered physically impossible with the current crawler setup. A better evaluation method should only include physically possible target linear and angular velocities. Figure 6.4 gave a comparison of the average RMSE of velocity between PPO, PPO-LSTM, and IVK across all environments. These results are difficult to understand, especially when comparing PPO to PPO-LSTM. The two models trade-off RMSE values between environments. This may be caused due to PPO-LSTM implementation. PPO and LSTM networks are both susceptible to performance changes between implementation, so tuning hyperparameters may provide clearer results. The IVK controller also gives some inconclusive results. For linear velocity, the IVK yields a smaller error for all environments; however, for angular velocity, it yields errors about 3.5 times greater than either PPO variant. This could be attributed to simulation inconsistencies. While Isaac Gym is a promising simulation software with great possibili-

ties for speed-up, it is still in its alpha version and is prone to bugs especially in its contact physics calculations. Further testing should be done to verify the validity of the physics conducted in Isaac Gym and PhysX.

Similar simulation artifacts can be seen in the results from Section 6.3.2. Though initial conditions seem to have little effect on both linear and angular velocity tracking, a significant amount of drifting can be seen in the trajectories. This could be attributed to the oscillations in angular velocity experienced as a result of the actions provided from the PPO model. One solution to this could be to penalize the model for sharp changes in applied wheel velocities. This would encourage smooth control of the crawler and result in smaller, less frequent oscillations in base velocities.

The last major finding was the resilience of a PPO model when trained in two different environments. Findings showed that a model trained in an ideal environment provided similar tracking errors in a noisy environment with constant tether disturbances to a model trained in that exact same environment. However, this also yields inconclusive results as to whether modeling the external disturbances truly benefits real-world velocity tracking.

Unfortunately, due to time constraints and other restrictions, testing of the DRL model could not be done on the real Altiscan crawler. The author acknowledges that deploying the model on the physical robot is the greatest indicator of performance. However, the promising results that DRL can be used for the velocity control of a DDR opens the gate for improvements and future work, such as:

- Modeling of true noise and bias experienced from the particle filter localization of the crawler. In this thesis, these values were chosen without regard for the real-world values.

- Replacing simulated tether forces with a soft-body rope. Isaac Gym offers soft-body simulation with its Flex physics backend. This allows for fluid dynamics contact physics calculations, such that of a rope or tether. This may provide a more realistic representation of the forces that act of the crawler.

- Cross-checking physics in Isaac Gym to ensure they align with real-world data. The physics calculations computed in Isaac Gym and PhysX were taken as a ground truth; however, many of the simulation artifacts experienced weaken the strength of that assumption and encourage the validation of the contact physics.

- Hyperparameter tuning for PPO. PPO is susceptible to differences in implementation and hyperparameters. Further testing should be done to find an optimal set of parameters for this task.

- Evaluation of different DRL models. PPO and PPO-LSTM were the only DRL algorithms used in this thesis. However, PPO is a constantly evolving field and state-of-the-art algorithms are continuously discovered. Algorithms such as SAC [69], ACKTR [70], and DDPG [71] could be

- Explore trajectory tracking control. While velocity control is useful for manual control of a robot, trajectory tracking control has greater value for autonomous systems, as is the end goal for the crawler in the BugWright2 project.

In conclusion, the results found in this thesis are encouraging for the applicability of deep reinforcement learning to the velocity control of the ROBOPLANET Altiscan crawler. Deep neural networks proved useful in capturing the dynamics of a nonlinear system which included wheel slippage and tether forces. The rapid prototyping enabled by the speed-up offered from Isaac Gym makes DRL and robotic control research more approachable and allows researchers to learn by trial-and-error. Furthermore, the versatility of DRL can be leveraged to include sensory observations to provide a complete autonomy package for the crawler. Overall, there is a rich potential for future work that builds off the research conducted in this thesis.

# Appendices

# APPENDIX A

# SIMULATION PHYSICS PARAMETERS

Table A.1: Isaac Gym Simulation Parameters.

| **Parameter Name** | **Value** | **Description** |
|---|---|---|
| dt | 0.05 | Simulation time step |
| substeps | 30 | Number of physics iterations between each time step |
| up_axis | "z" | Up axis |
| gravity | $[0, 0, -9.81]$ for horizontal surfaces<br>[0, -9.81, -135] for vertical surfaces with magnetism | Gravity vector for simulation. A vertical plane is simulated by placing -9.81 in the $y$ position. Magnetism is simulated by placing -135 in the $z$ position |
| use_gpu_pipeline | "gpu" | Uses either GPU or CPU pipeline |

Table A.2: PhysX Parameters

| **Parameter Name** | **Value** | **Description** |
|---|---|---|
| bounce_velocity_threshold | 0.0 | A contact with a relative velocity below this will not bounce. |
| contact_collection | 2 | Collects contacts from all previous substeps. |
| contact_offset | 0.0025 | Shapes whose distance is less than the sum of their contact offset values will generate contacts. |
| default_buffer_size_multiplier | 5.0 | Default buffer size multiplier. |
| max_depenetration_velocity | 100.0 | Maximum velocity permitted to be introduced by the solver to correct the penetrations in contacts. |
| max_gpu_contact_pairs | 1048576 | Maximum number of contact pairs. |
| num_position_iterations | 6 | PhysX solver position iteration counts. |
| num_subscenes | 4 | Number of subscenes for multithreaded simulation |
| num_velocity_iterations | 2 | PhysX solver velocity iteration counts. |
| rest_offset | 0.0 | Distance for two objects to come to rest at. |
| solver_type | 2 | PhysX solver used. 1 for the Projected Gauss-Siedel (PGS) solver. 2 for the Temporal Gauss-Siedel (TGS). |

# APPENDIX B

# MODEL PARAMETERS

## B.1 PPO Parameters

Table B.1: Base Model Parameters

| **Parameter Section** | **Parameter Name** | **Value** | **Description** |
|---|---|---|---|
| base | seed | 42 | Random seed for PyTorch |
| algo | name | a2c_continuous | Base algorithm type |
| model | name | continuous_a2c_logstd | Base model type |
| network | name | actor_critic | Network type |
| | separate | False | Use separate networks for actor and critic |
| mlp | units | $[256, 256, 128]$ | Defines the structure of the DNN |
| | activation | relu | Activation function used for each layer |

Table B.2: PPO Paramters for rl-games

| **Parameter Name** | **Value** | **Description** |
|---|---|---|
| ppo | True | Use PPO for gradient stepping |
| normalize_input | True | Apply running mean and standard deviation normalization for input |
| normalize_value | True | Apply running mean and standard deviation normalization for value |
| scale_value | 0.01 | Scale for computed values |
| normalize_advantage | True | Apply running mean and standard deviation for advantage function |
| gamma | 0.99 | Value to discount future rewards by |
| tau | 0.95 | Lambda for the Generalized Advantage Estimate |
| learning_rate | 1e-3 | Learning rate for |
| lr_schedule | adaptive | Learning rate scheduler for each miniepoch |
| kl_threshold | 0.016 | KL threshold for adaptive learning rate scheduler |
| max_epochs | 500 | Maximum number of forward and backward passes of observation data |
| grad_norm | 1.0 | Gradient truncation value for learning stabilization |
| entropy_coef | 0.0 | Entropy coefficient |
| clip_loss | True | Apply PPO clip parameter |
| e_clip | 0.2 | Clip parameter for PPO loss |
| horizon_length | 32 | Horizon length for each actor |
| minibatch_size | 32768 | Minibatch size |
| mini_epochs | 8 | Number of miniepochs |
| critic_coef | 2 | Coefficient for critic loss |
| bounds_loss_coef | 1e-4 | Coefficient to the auxiliary loss for continuous space |

Table B.3: PPO-LSTM Recurrent Network Parameters for rl-games

| **Parameter Name** | **Value** | **Description** |
|---|---|---|
| name | lstm | Type of recurrent network layer |
| layers | 1 | Number of recurrent layers |
| units | 128 | Number of nodes in the recurrent layer |
| before_mlp | False | Append the recurrent layer before the DNN |

# REFERENCES


[1] L. E. Parker and J. V. Draper, "Robotics applications in maintenance and repair," *Handbook of industrial robotics*, vol. 2, pp. 1023–1036, 1998.

[2] D. Lattanzi and G. Miller, "Review of robotic infrastructure inspection systems," *Journal of Infrastructure Systems*, vol. 23, no. 3, p. 04 017 004, 2017.

[3] I. N. Ismail *et al.*, "Development of in-pipe inspection robot: A review," in *2012 IEEE Conference on Sustainable Utilization and Development in Engineering and Technology (STUDENT)*, IEEE, 2012, pp. 310–315.

[4] C. Pradalier, *Bugwright2: Autonomous inspection and maintenance on ship hulls*, [Online; accessed November 27, 2022], 2022.

[5] ROBOPLANET, *Altiscan v1.3*, [Online; accessed November 27, 2022], 2022.

[6] G. Chahine, P. Schroepfer, O.-L. Ouabi, and C. Pradalier, "A magnetic crawler system for autonomous long-range inspection and maintenance on large structures," *Sensors*, vol. 22, no. 9, p. 3235, Apr. 2022.

[7] M. Quigley *et al.*, "Ros: An open-source robot operating system," in *ICRA Workshop on Open Source Software*, 2009.

[8] M. Zhu, Y. Wang, Z. Pu, J. Hu, X. Wang, and R. Ke, "Safe, efficient, and comfortable velocity control based on reinforcement learning for autonomous driving," *Transportation Research Part C: Emerging Technologies*, vol. 117, p. 102 662, 2020.

[9] N. E. M. Joseph L. Jones and P. E. S. David M. Nugent, *Autonomous floor-cleaning robot*, U.S. Patent 6883201B2, 2005.

[10] P. Fiorini and D. Botturi, "Introducing service robotics to the pharmaceutical industry," *Intelligent Service Robotics*, vol. 1, pp. 267–280, Jan. 2008.

[11] F. Dellaert and S. Hutchinson, *Introduction to perception and robotics*, [Online; accessed November 16, 2022], 2022.

[12] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[13] S. P. Singh, T. Jaakkola, M. Littman, and C. Szepesvári, "Convergence results for single-step on-policy reinforcement-learning algorithms," *Machine Learning*, vol. 38, no. 3, pp. 287–308, 2000.

[14] J. Tsitsiklis and B. Van Roy, "An analysis of temporal-difference learning with function approximation," *IEEE Transactions on Automatic Control*, vol. 42, no. 5, pp. 674–690, 1997.

[15] L. Pyeatt and A. Howe, "Decision tree function approximation in reinforcement learning," Jul. 2001.

[16] D. Shah and Q. Xie, "Q-learning with nearest neighbors," *CoRR*, vol. abs/1802.03900, 2018. arXiv: 1802.03900.

[17] M. Leshno, V. Y. Lin, A. Pinkus, and S. Schocken, "Multilayer feedforward networks with a nonpolynomial activation function can approximate any function," *Neural Networks*, vol. 6, no. 6, pp. 861–867, 1993.

[18] G. Litjens *et al.*, "A survey on deep learning in medical image analysis," *Medical Image Analysis*, vol. 42, pp. 60–88, 2017.

[19] E. Yurtsever, J. Lambert, A. Carballo, and K. Takeda, "A survey of autonomous driving: Common practices and emerging technologies," *IEEE Access*, vol. 8, pp. 58 443–58 469, 2020.

[20] I. J. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016, http://www.deeplearningbook.org.

[21] C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag, 2006, ISBN: 0387310738.

[22] C. Olah, "Understanding lstms," *colah's blog*, 2015.

[23] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997. eprint: https://direct.mit.edu/neco/article-pdf/9/8/1735/813796/neco.1997.9.8.1735.pdf.

[24] G. Tesauro, "Td-gammon, a self-teaching backgammon program, achieves master-level play," *Neural Computation*, vol. 6, no. 2, pp. 215–219, 1994.

[25] D. Silver *et al.*, "A general reinforcement learning algorithm that masters chess, shogi, and go through self-play," *Science*, vol. 362, no. 6419, pp. 1140–1144, 2018. eprint: https://www.science.org/doi/pdf/10.1126/science.aar6404.

[26] D. Silver *et al.*, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, pp. 484–489, Jan. 2016.

[27] T. Sogabe *et al.*, "Smart grid optimization by deep reinforcement learning over discrete and continuous action space," in *2018 IEEE 7th World Conference on Photo-*

*voltaic Energy Conversion (WCPEC) (A Joint Conference of 45th IEEE PVSC, 28th PVSEC & 34th EU PVSEC)*, 2018, pp. 3794–3796.

[28] S. S. Gu, E. Holly, T. P. Lillicrap, and S. Levine, "Deep reinforcement learning for robotic manipulation," *ArXiv*, vol. abs/1610.00633, 2016.

[29] V. Mnih *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529–33, Feb. 2015.

[30] J. O'Neill, B. Pleydell-Bouverie, D. Dupret, and J. Csicsvari, "Play it again: Reactivation of waking experience and memory," *Trends in Neurosciences*, vol. 33, no. 5, pp. 220–229, 2010.

[31] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Mach. Learn.*, vol. 8, no. 3–4, pp. 229–256, 1992.

[32] V. Konda and J. Tsitsiklis, "Actor-critic algorithms," in *Advances in Neural Information Processing Systems*, S. Solla, T. Leen, and K. Müller, Eds., vol. 12, MIT Press, 1999.

[33] R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Advances in Neural Information Processing Systems*, S. Solla, T. Leen, and K. Müller, Eds., vol. 12, MIT Press, 1999.

[34] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel, *Trust region policy optimization*, 2015.

[35] S. M. Kakade and J. Langford, "Approximately optimal approximate reinforcement learning," in *ICML*, 2002.

[36] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, *Proximal policy optimization algorithms*, 2017.

[37] A. O'Dwyer, "Pi and pid controller tuning rules: An overview and personal perspective," in *2006 IET Irish Signals and Systems Conference*, 2006, pp. 161–166.

[38] U. Zangina, S. Buyamin, M. S. Z. Abidin, M. S. Azimi, and H. Hasan, "Non-linear pid controller for trajectory tracking of a differential drive mobile robot," *Journal of Mechanical Engineering Research and Developments*, vol. 43, no. 1, pp. 255–270, 2020.

[39] J. Heikkinen, T. Minav, and A. D. Stotckaia, "Self-tuning parameter fuzzy pid controller for autonomous differential drive mobile robot," in *2017 XX IEEE Interna-*

*tional Conference on Soft Computing and Measurements (SCM)*, 2017, pp. 382–385.

[40] C. Qiu, C. Liu, F. Shen, and J. Chen, "Design of automobile cruise control system based on matlab and fuzzy pid," *Transactions of the Chinese Society of Agricultural Engineering*, vol. 28, no. 6, pp. 197–202, 2012.

[41] R. Pradhan, S. K. Majhi, J. K. Pradhan, and B. B. Pati, "Antlion optimizer tuned pid controller based on bode ideal transfer function for automobile cruise control system," *Journal of Industrial Information Integration*, vol. 9, pp. 45–52, 2018.

[42] T. Wang, R. Dong, R. Zhang, and D. Qin, "Research on stability design of differential drive fork-type agv based on pid control," *Electronics*, vol. 9, no. 7, p. 1072, 2020.

[43] J. van den Berg, P. Abbeel, and K. Goldberg, "Lqg-mp: Optimized path planning for robots with motion uncertainty and imperfect state information," *I. J. Robotic Res.*, vol. 30, pp. 895–913, Jun. 2011.

[44] P. Petrov and V. Georgieva, "Adaptive velocity control for a differential drive mobile robot," in *2018 20th International Symposium on Electrical Apparatus and Technologies (SIELA)*, 2018, pp. 1–4.

[45] H. Zhang, J. Gong, Y. Jiang, G. Xiong, and H. Chen, "An iterative linear quadratic regulator based trajectory tracking controller for wheeled mobile robot," *J. Zhejiang Univ. Sci. C*, vol. 13, no. 8, pp. 593–600, 2012.

[46] M. Ran, J. Li, and L. Xie, "Reinforcement learning-based disturbance rejection control for uncertain nonlinear systems," 2020.

[47] Z. Li *et al.*, *Reinforcement learning for robust parameterized locomotion control of bipedal robots*, 2021.

[48] N. Rudin, D. Hoeller, P. Reist, and M. Hutter, *Learning to walk in minutes using massively parallel deep reinforcement learning*, 2021.

[49] E. Coumans and Y. Bai, *Pybullet, a python module for physics simulation for games, robotics and machine learning*, http://pybullet.org, 2016.

[50] E. Todorov, T. Erez, and Y. Tassa, "Mujoco: A physics engine for model-based control," in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2012, pp. 5026–5033.

[51] J. Hwangbo, J. Lee, and M. Hutter, "Per-contact iteration method for solving contact dynamics," *IEEE Robotics and Automation Letters*, vol. 3, no. 2, pp. 895–902, 2018.

[52] R. Tedrake and the Drake Development Team, *Drake: Model-based design and verification for robotics*, 2019.

[53] V. Makoviychuk *et al.*, *Isaac gym: High performance gpu-based physics simulation for robot learning*, 2021.

[54] A. Paszke *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., Curran Associates, Inc., 2019, pp. 8024–8035.

[55] M. Macklin *et al.*, "Small steps in physics simulation," in *Proceedings of the 18th Annual ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, ser. SCA '19, Los Angeles, California: Association for Computing Machinery, 2019, ISBN: 9781450366779.

[56] G. Brockman *et al.*, *Openai gym*, 2016. eprint: arXiv:1606.01540.

[57] V. Mnih *et al.*, "Asynchronous methods for deep reinforcement learning," 2016.

[58] D. Makoviichuk and V. Makoviychuk, *Rl-games: A high-performance framework for reinforcement learning*, https://github.com/Denys88/rl_games, May 2022.

[59] A. Allshire *et al.*, *Transferring dexterous manipulation from gpu simulation to a remote real-world trifinger*, 2021.

[60] C. S. de Witt *et al.*, *Is independent learning all you need in the starcraft multi-agent challenge?* 2020.

[61] J. Wong, V. Makoviychuk, A. Anandkumar, and Y. Zhu, *Oscar: Data-driven operational space control for adaptive and robust robot manipulation*, 2021.

[62] O. Yadan, *Hydra - a framework for elegantly configuring complex applications*, Github, 2019.

[63] W. Zhao, J. P. Queralta, and T. Westerlund, "Sim-to-real transfer in deep reinforcement learning for robotics: A survey," *CoRR*, vol. abs/2009.13303, 2020. arXiv: 2009.13303.

[64] J. Bai, F. Lu, K. Zhang, *et al.*, *Onnx: Open neural network exchange*, https://github.com/onnx/onnx, 2019.

[65] L. Biewald, *Experiment tracking with weights and biases*, Software available from wandb.com, 2020.

[66] L. Espeholt, R. Marinier, P. Stanczyk, K. Wang, and M. Michalski, *Seed rl: Scalable and efficient deep-rl with accelerated central inference*, 2019.

[67] Martín Abadi *et al.*, *TensorFlow: Large-scale machine learning on heterogeneous systems*, Software available from tensorflow.org, 2015.

[68] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, Mar. 2014.

[69] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, *Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor*, 2018.

[70] Y. Wu, E. Mansimov, S. Liao, R. Grosse, and J. Ba, *Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation*, 2017.

[71] T. P. Lillicrap *et al.*, *Continuous control with deep reinforcement learning*, 2015.