

Investigating legged robots' mobility control on simple surfaces using a light-weight machine learning architecture

by

Cory Morton

B.S., University of Oklahoma, 2011
M.S., University of Oklahoma, 2013

A THESIS

submitted in partial fulfillment of the requirements for the degree

MASTER OF SCIENCE

Department of Integrated Studies
College of Technology & Aviation

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2025

Approved by:

Major Professor
Dr. Michael J. Pritchard

Copyright

© Cory Morton 2025.

Abstract

Establishing a long-term human presence on the lunar surface necessitates robust autonomous systems capable of navigating unpredictable terrain with limited external control. Surface mobility is challenged by diverse slopes, abrasive regolith, and low-gravity conditions, all while operating under strict power and processing constraints. This thesis investigates the application of Deep Reinforcement Learning (DRL) to train a neural network-based motion controller for a legged robotic platform, with the goal of deploying the trained controller to a low-resource microcontroller. Using the MuJoCo-based Ant environment from the Gymnasium API, a DRL agent was trained with the Proximal Policy Optimization (PPO) algorithm to develop stable and adaptive locomotion policies. The trained model was then quantized and converted using the ExecuTorch framework for deployment on an Arduino Nano BLE 33 Sense microcontroller, representative of the limited processing resources expected in real-world space missions. Performance was evaluated by comparing the behavior of the controller on the development machine to its performance on the microcontroller. Results showed that the deployed model could maintain similar performance to the trained model. While performance degradation due to conversion and hardware limitations was observed, the study offers a practical roadmap for bridging high-performance simulation with real-time embedded control. These findings contribute to advancing autonomous mobility systems for extraterrestrial exploration, emphasizing the potential of DRL and TinyML to operate effectively in low-power, communication-limited environments.

Table of Contents

List of Figures	v
List of Tables	vi
Acknowledgements	vii
Dedication	viii
Chapter 1 - Introduction.....	1
Background / Motivation	1
Mobility Challenges on Lunar Surface	1
Autonomy in Mobile Robots	3
Deep Reinforcement Learning (DRL)	4
Research Objectives and Scope	8
Chapter 2 - Literature Review.....	10
Machine Learning for Motion Control	10
Machine Learning on Edge Devices	15
Chapter 3 - Methodology	21
Platform and Environment Definition	21
Research Process.....	26
Chapter 4 - Results and Discussion	32
Average Episode Length.....	36
Forward Distance Traveled.....	37
Path Distance	39
Mean Joint Torques	44
Chapter 5 - Conclusions.....	46
Limitations and Future Research	46
Summary	49
Bibliography	51
Appendix A - Research Algorithms.....	57
Appendix B - System-Level Integration	60

List of Figures

Figure 1.1. Reinforcement Learning Concept (generated by AI)	5
Figure 1.2. Deep Neural Network (DNN) Concept	6
Figure 1.3. Proximal Policy Optimization (PPO) Concept.....	7
Figure 3.1. Platform Architecture	22
Figure 3.2. Physical Architecture.....	23
Figure 3.3. Research Process	28
Figure 3.4. ExecuTorch Concept (from ExecuTorch)	29
Figure 4.1. Episode Data from Flipped Platforms	35
Figure 4.2. Maximum x-Position vs. Episode Length	39
Figure 4.3. Average Total Distance per Timestep	41
Figure 4.4. Ratio of x-Position to Total Distance	42
Figure 4.5. Average x-Position per Timestep	42
Figure 4.6. Average y-Position per Timestep	43
Figure 4.7. Average Path Traveled from Origin	43
Figure 4.8. Average Mean Joint Torque per Timestep	45

List of Tables

Table 3.1. Training Parameters	28
Table 3.2. Software Configuration.....	30
Table 4.1: Performance Overview	34
Table 4.2: Average Episode Length.....	36
Table 4.3. Average Forward Distance Traveled	37
Table B.1. Gymnasium Ant-v4 Environment Parameters	61
Table B.2: Gymnasium Ant-v4 Platform Characteristics	61

Acknowledgements

I would like to express my deepest gratitude to Dr. Michael Pritchard for his unwavering encouragement, enthusiasm, and guidance throughout this process. His mentorship was instrumental in helping me refine a research topic that was both exciting and feasible, and his support inspired me to persevere through the challenges I encountered along the way.

I would also like to extend my sincere appreciation to my committee members, Dr. Balaji Balasubramaniam and Dr. Sridevi Pudipeddi, for their time, thoughtful feedback, and valuable insights, which greatly contributed to the success of this thesis.

Dedication

To my family—thank you for standing beside me through every challenge and cheering me on in every accomplishment. Your love and belief in me have made all the difference.

Chapter 1 - Introduction

Background / Motivation

With renewed motivation to establish a permanent presence on the Moon, the ability to explore the lunar surface is critical. Exploration of the surface is necessary in deciding where to establish initial and subsequent operating bases on the moon's surface. Location will play a critical role in developing a sustainable lunar base driven by access to key resources. To achieve this, various methods will be deployed ranging from high altitude, orbiting spacecraft providing wide surface imagery, and observation to on-surface vehicles for localized observation and sample analysis.

Mobility Challenges on Lunar Surface

There are numerous challenges to mobility over the lunar surface. These challenges include: 1) diverse, uneven terrain, 2) loose regolith, and 3) different environmental conditions compared to Earth.

The first challenge is the diversity of terrain, ranging from sandy fields to steep, rocky inclines. This diversity will mean that the terrain will exhibit various slopes. Using data collected from the Lunar Orbital Laser Altimeter (LOLA), Rosenberg et al. (2011) identified slopes up to 10 degrees could be found globally on the Moon's surface with potential to experience slopes between 15-20 degrees. Furthermore, our knowledge of soil composition limits our ability to model, simulate, and test representative designs on Earth. A common approach to overcome this limitation is to use terramechanics to model the interaction of the vehicle's locomotive system with that of the surface (Rodríguez-Martínez et al., 2019). However, these models are influenced by assumptions and constraints derived from the Earth's surface. This leads to the conservative

and redundant design elements in the mobility system to allow for the system to adapt and endure an unknown environment.

The second challenge is the layer of fine, powdery dust, known as regolith, that covers the lunar surface. This layer can be several meters deep, which poses the risk of entrapment of wheeled and legged vehicles. This dust is highly abrasive that can damage mechanical components, joints, and seals, reducing the overall life of the vehicle (Cannon et al., 2022; Stubbs et al., 2007). Complicating factors further, the dust can become electrostatically charged, which causes it to stick to surfaces (Horányi et al., 1998; Stubbs et al., 2007). Mobility systems designed for the lunar surface will need to utilize mitigation mechanisms to overcome this challenge.

The third challenge is the difference in environmental conditions experienced on the surface of the Moon compared to that of Earth. First, the Moon's gravitational field is 1/6-g compared to Earth. This results in decreased traction, which leads to increased slip and higher risk of entrapment within the loose soil. Niksirat et al. (2020) measured up to 20% reduction in traction in a simulated lunar-g environment to that of 1-g environment using a wheeled mechanism. Secondly, there is no atmosphere at the lunar surface. This limits mobility systems to using wheeled, tracked, or legged mechanisms. Bruzzone et al. (2022) provides an overview of different forms of locomotion for mobile ground-based robots. Alternatively, a propulsion system, using thrusters, could be employed to produce a hopping trajectory (Rimani et al., 2023). However, flight cannot be achieved using propeller-driven motion as these require an atmosphere.

Autonomy in Mobile Robots

In addition to the mobility challenges expressed in the previous section, mobile robots also face the increasing need to perform autonomously. Remote control of the vehicle is limited based on communication latency and bandwidth constraints. Remote operators commanding a lunar vehicle would experience a minimal response delay of ~ 2.6 seconds. This does not account for any process delay upon the vehicle receiving the command, executing, and returning data of its state back to the operator. Serria et al. (2023) summarizes additional communication challenges to include multipath propagation and path loss due to lunar surface roughness (i.e., craters and valleys), lunar dust and Moon's magnetic field. These environmental conditions impact the reliability in establishing communication links wirelessly. While there are projects (i.e., NASA's LunaNet) aiming to bring reliable and robust communication services Beyond Low Earth Orbit (BLEO), these services may not be fielded to support early exploration efforts (Israel et al., 2020). Autonomy in the vehicle's design would expand its capabilities in exploration through minimizing communication risks.

One approach for enabling autonomy within system is to use machine learning techniques to train an agent to perform a specific task. Machine learning is a subfield of Artificial Intelligence (AI) that uses algorithms to enable an agent to learn from data and make predictions or decisions without being explicitly programmed. There are three forms of machine learning approaches: unsupervised, supervised, and reinforcement learning. Unsupervised learning works by providing unlabeled data to the agent, where the agent attempts to identify patterns in the data. These patterns are then used to make predictions or decisions on new data provided to the agent. On the other hand, supervised learning works by providing labeled data to the agent, where the agent learns to associate the data with specific labels. This provides a more guided

approach to training than unsupervised learning. Lastly, reinforcement learning takes a different approach by allowing the agent to learn through interaction with its environment. This leans heavily on a feedback loop within the training process to allow the agent to learn the desired behavior when making predictions or decisions. Additionally, each approach may be used in conjunction with the other approaches to form hybrid scenarios. For example, an image classification agent might be trained using supervised learning initially as a generic model, then uses reinforcement learning methods after deployment to tailor the agent for certain use cases. For this study, reinforcement learning was selected based on the wide application of its use in robotic development and the nature of the problem space. The next section further describes the reinforcement learning concepts employed within the study.

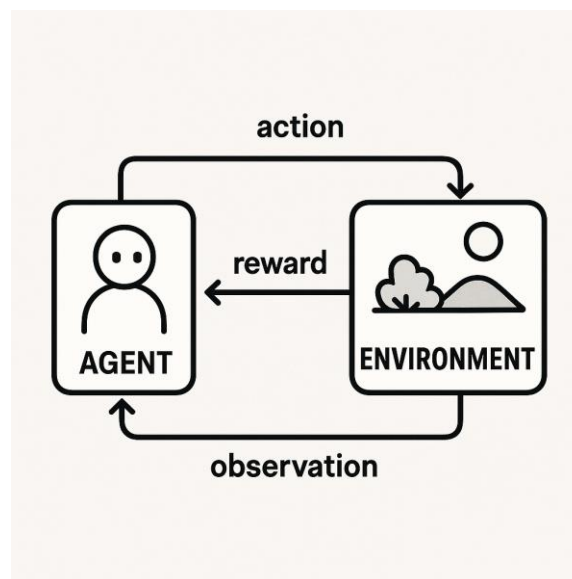
Deep Reinforcement Learning (DRL)

Machine learning is a key discipline in driving more autonomy into robotic platforms. Specifically, reinforcement learning approaches are prevalent in this area. The research presented in this thesis will be focused on using deep reinforcement learning (DRL) methods. This section introduces the key concepts behind DRL that will be referred to throughout the thesis. DRL integrates reinforcement learning with deep neural networks to allow agents to learn more complex policies. DRL is often employed where a stochastic policy is required, or the action or observation space is high-dimensional, continuous, unstructured, or a combination of those. To introduce the concepts, an introduction to reinforcement learning is provided, followed by a discussion of the chosen DRL algorithm used in this research.

Reinforcement Learning is a machine learning approach where an agent learns to make decisions by interacting with the environment. The reinforcement learning framework is composed of an agent and an environment. The agent uses a policy to select an action based on

the given state or observation of the environment. Note that the observation represents what the agent observes of the environment and may not include the entire state space of the environment. This means that the agent will choose an action based on its observation of the environment's state. Once the action is performed, the environment updates its state, and a reward is generated. The reward is used as feedback to the agent, which allows the agent to evaluate the effectiveness of the action at that given state and update its policy accordingly. This forms the concept of this machine learning approach, where the agent attempts to learn the optimal policy for a given objective or task through trial and error.

Figure 1.1. Reinforcement Learning Concept (generated by AI)



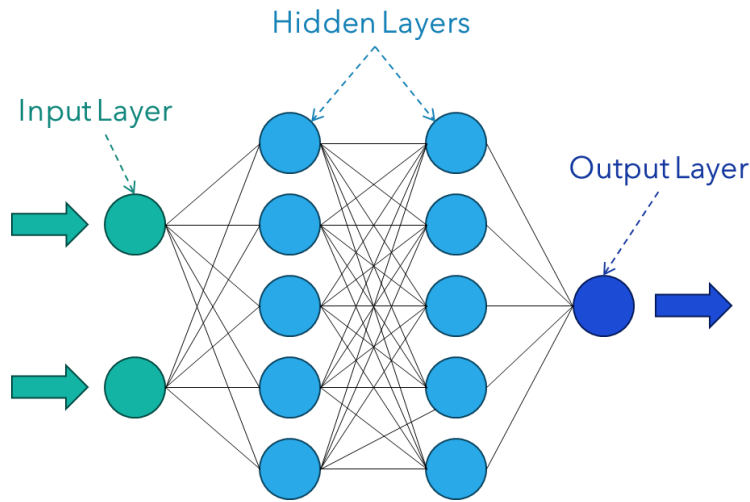
DRL uses deep neural networks (DNN) for the agent's policies within the learning framework. DNNs are a type of artificial neural network that includes multiple layers between the input and output, known as hidden layers. Neural networks mimic the structure and behavior of neurons, hence the name neural network. Each hidden layer in the network receives an input,

applies a weighted sum, adds bias, and passes it to the activation function. The activation is defined as

$$output = f\left(\sum w_i x_i + b\right) \quad (1)$$

where w_i is the weight, x_i is the input, and b is the bias. The activation functions perform non-linear transformations (i.e., Rectified Linear Unit (ReLU), Sigmoid, Tanh, etc.). The output of the activation function is fed downstream to the next layer. The final layer in the neural network represents is the output layer, which is the output response of the neural network. Training a DNN consists of adjusting the weights and biases of the hidden layers to find the optimal policy.

Figure 1.2. Deep Neural Network (DNN) Concept



The algorithms classified under DRL can be categorized under the following categories: Value-Based Methods, Policy-Based Methods, Actor-Critic Methods, and Model-Based Reinforcement Learning (MBRL). This research applied the Proximal Policy Optimization (PPO) algorithm (Schulman et al., 2017), which is categorized as an Actor-Critic method. Actor-Critic methods combined value-based and policy-based learning together. PPO extends from the

Advantage Actor-Critic (A2C) algorithm. A2C algorithm uses an “actor” (policy function) and a “critic” (value function) in the form of a neural network. The actor observes the state and chooses an action based on its policy. The policy can be classified as either deterministic or stochastic. Deterministic policies choose a single action for the given state, where the stochastic policy chooses from a probability distribution over a set of actions for the given state. A stochastic policy is define as

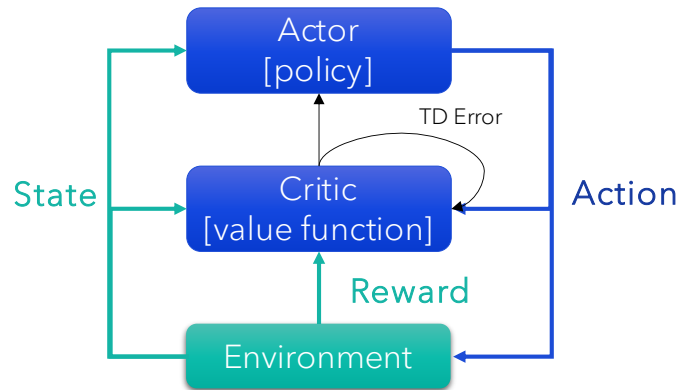
$$\pi_{\theta}(s) = P[a|s, \theta] \quad (2)$$

where π is the policy, a is the action, s is the state, and θ is the probability distribution. The critic observes the state and action chosen then predicts the Q-value of that pair. To stabilize learning, an Advantage function is used by the critic. The Advantage function takes the Q-value of the state-action pair and subtracts the mean value for state. The Advantage function is defined as

$$A(s, a) = Q(s, a) - V(s) \quad (3)$$

where $A(s, a)$ is the advantage of taking action a at state s , $Q(s, a)$ is the Q-value of the action-state pair, and $V(s)$ is the mean value at state s . This allows the critic to evaluate the value of taking an action at the given state compared to the average value of that state.

Figure 1.3. Proximal Policy Optimization (PPO) Concept



In addition to A2C, PPO implementation features parallelization, clipped surrogate objective function, and entropy term. Parallelization refers to the collection of experience through multiple, parallel environments. This improves sample throughput and training speed by allowing the agent to learn through interacting with multiple environments simultaneously. Clipped surrogate objective function is the key characteristic of PPO. This feature restricts how much the policy changes during an update through a clipping mechanism, increasing the learning stability by preventing large policy updates (Schulman et al., 2017). Lastly, the entropy term is used to encourage exploration during training. This is to prevent the stochastic policy from becoming too deterministic.

This introduces the basic concepts employed by the learning framework for this research methodology. The following sections will describe the research objectives, related work to the research topic, the design methodology, and results. The concepts described here will be referenced throughout these sections.

Research Objectives and Scope

This research aims to address the following research question: *how effective are DRL methods at training a neural network-based motion controller for a legged autonomous system to traverse simple terrain when deployed on a resource-constrained microcontroller?* The research explores the effectiveness of training a neural network to control the motion of a legged autonomous system through DRL methods. The focus aims to address the challenge for autonomy. A legged platform was selected as it offers the capability to traverse flat or rugged terrain; however, the research limited the environment to a flat plane. DRL method was selected to evaluate the capability to train a motion controller model for the legged platform that features a multi-valued observation and action space. Effectiveness of the motion controller model is

evaluated as the ability to control the movement of the autonomous system to traverse the terrain to the end goal. The motion controller model is deployed to a microcontroller as a representative processing platform. This creates additional constraints on the operating performance of the motion controller model. In addition, the research offers a description of a process from training to deployment to operation used by the experiment. The insight gained from the experiment should highlight the current state of training a motion controller for execution on a microcontroller and highlight limitations and challenges.

Chapter 2 - Literature Review

Machine Learning for Motion Control

Research in using machine learning approaches for motion control of autonomous systems has focused on global and local path planning; however, some research has looked at using machine learning models to replace or supplement the motion controller. This section summarizes the literature on using machine learning in context of the motion controller.

DRL approach is common in applications of autonomous system control and motion, ranging from unmanned aerial vehicles (UAVs) to self-driving vehicles. For example, Wan et al. (2020) trained a controller for a fixed wing, simulated UAV that controls its speed and turn rate to navigate a populated airspace. A deep deterministic policy gradient (DDPG) algorithm was used to train the controller within a simulated environment. They showed performance improvement in the learned controller through modification of the learning process, referred to as Robust-DDPG. Furthermore, Jiang & Lynch (2021) trained a full dynamic controller for a simulated quadcopter. A deep neural network (DNN) controller was trained using proximal policy optimization (PPO) algorithm. They were able to conclude similar performance in hover stabilization compared to that of a manually tuned PD controller. Next, Aradi (2022) performed a survey of applications of DRL within self-driving vehicles. They categorized the applications into four layers: 1) route planning, 2) behavioral layer, 3) motion planning, and 4) local feedback control. They also concluded that, while DRL provides advantages of adaptability and efficiency, there is still a need to combine them with classical control methods. Furthermore, Dawane et al. (2023) performed a theoretical comparison of three control methodologies: 1) Proportional-Integral-Derivative (PID) control, 2) Sliding Motion control, and 3) machine learning. Additionally, they surveyed case studies applying these methodologies. It was found that cases

often attempted some combination of the three methods to take advantage of their strengths and mitigate their weaknesses. Machine learning was found to be used primarily for path planning and object detection. These studies illustrate research into using machine learning for motion control and planning; however, it does not represent motion in legged robotics.

For this study, the objective is to study motion control within legged robots as the targeted platform. Therefore, the literature survey was narrowed to research related to machine learning application for motion control within legged robots. First, Tsounis et al. (2020) proposed a training technique to train a quadruped robot to walk across uneven terrain. Their proposed architecture is to train two neural network policies: one for motion planning and the other for motion control. The motion planner was trained using a model-free DRL approach, which aimed to optimize a set of footholds and center of mass (CoM) motion, referred to as support phases. The motion controller was trained using a model-based linear program (LP) approach to identify optimal CoM trajectories between the support phases from the motion planner. The trained models were applied to a simulated quadruped within different simulated environments, which includes narrow bridges, steppingstones, and gaps. Simulation demonstrated successful locomotion across these environments.

Similarly, Lee et al. (2020) proposed a training method and control architecture for a quadruped robot using proprioceptive sensing. The training method consisted of two phases: teacher policy training and student policy training. The teacher policy was trained with additional privileged information using reinforcement learning approach within a simulated environment. The student policy was trained using imitation learning and supervision of the teacher policy. After training, the student policy was deployed on a robotic platform, ANYmal. Unfortunately, this platform features an Intel processor and not a microcontroller (ANYbotics, n.d.).

Experiments were performed in both virtual and physical (indoor and outdoor) environments. These environments presented uneven terrain, various surface conditions and obstacles. The study showed the trained controller was able to provide adaptive control in these environments; however, did note that the motion was limited to a single gait and did not represent gait transitions.

Furthermore, Lele et al. (2020) explores the combination Spiking Neural Networks (SNNs) and Central Pattern Generators (CPGs) to allow for on-device reinforcement learning using a lightweight raspberry pi platform with integrated sensors. The objective of the training phase was to learn a closed-loop locomotion control policy for a hexapod robot. This varied from the previous two studies in that both used virtual environment and offline training to develop the trained motion controllers. The physical environment was representative of a flat surface with no obstacles. The platform observed the environment through a visual camera and its state through gyrometer. The study demonstrated that the controller was able to learn the tripod gait pattern in 70% of cases.

In addition, Ouyang et al. (2021) studied a two layered CPG approach for motion control of a hexapod robot inspired by biological neuro control systems. The first layer, called the Conscious Layer, directed the mode of locomotion or gait pattern to employ. The second layer, called the Behavior Layer, controlled the behavior of the limbs. An additional controller was trained using a DDPG to control several parameters to allow for adaptability in the control system. This architecture aimed to reduce the search space by limiting the locomotion to that modeled within the CPG, which increased learning speed. Both simulations and experiments were executed to evaluate performance. The simulations evaluated the convergence of the learning process and to estimate the characteristic velocity of the hexapod platform on four

different terrains: 1) flat, 2) sandpaper, 3) soft sand, and 4) up-slope of 10 degrees. It was observed that the average reward stabilized by the maximum time step of 1,400, and that the average training time took about 6 hours. The experiments were performed in a physical environment with the controller deployed on the targeted hexapod platform. On-device training was limited to 5 seconds. On-device training was used to tune the policy learned during simulations after being deployed. It was observed significant velocity performance differences between the results of the simulation and experiments.

Similarly, Qiu et al. (2023) used a two layered, hierarchical architecture for motion control in a hexapod robot. Their architecture used a high-level planner that included the policy network. The policy network generated gait parameters that fed into the Gait Planner. The Gait Planner would generate commands that drove the low-level controller. The low-level controller was responsible for controlling the movement of the limbs. The study investigated training the policy using four methods: 1) SAC, 2) PPO, 3) DDPG, and 4) TD3. The learning objective was to learn different gaits according to various velocity commands and motion velocity. The simulation demonstrated that SAC had the highest training efficiency compared to the other methods; however, each method was able to learn gaits that exhibited stable behavior. In addition to the simulation, an experiment was conducted that deployed the controller to the targeted physical platform. The controller's performance was observed to have decreased compared to the simulations.

Additionally, Seo et al. (2023) proposed a learning method and control architecture, featuring a high-level navigator and low-level gait controller. The high-level navigator generates velocity commands to the gait controller based on visual information of the environment. The low-level gait controller generated actuator commands for the joints based on the velocity

commands and robot state data, such as orientation. The authors proposed a training framework called PRELUDE. This framework used imitation learning to train the high-level navigator, where a steerable cart is pushed by a human for training. The low-level gait controller was trained using a PPO algorithm using a simulated environment. Experiments were performed within a simulated environment using five different configurations, including the proposed control architecture. The environment was configured based on three difficulty levels. It was found that the proposed control architecture achieved greater average distance traveled compared to the other four configurations.

Conversely, Schilling et al. (2020) proposed a different approach compared to the hierarchical control architecture presented in the previous two studies. Their study proposed a decentralized control architecture, where each limb was controlled by a dedicated controller that was trained using a PPO method. A centralized control architecture was also trained to represent the baseline for comparison. Both architectures were trained and evaluated using two virtual environments: flat surface and uneven surface. The virtual platform was modeled after PhantomX MK-II by Trossen Robotics. It was observed that the decentralized control architecture exhibits higher learning efficiency compared to the centralized control architecture. This was expected since the centralized control architecture would require a higher-dimensional action space compared to the decentralized control architecture. Additionally, it was observed that models trained in one environment exhibited degraded performance when generalizing to another environment.

Finally, Vinella (2024) explored the use of a two-agent control architecture, featuring a CPG Tuner and a Reflex Controller, for a bipedal humanoid model. The CPG Tuner network controlled the parameters for the oscillating signal produced by the CPG. This signal fed into the

Reflex controller, which controls muscle activations based on a separate learned policy. Both agents were trained using a SAC algorithm within a virtual environment. While this study focused on a model of a humanoid vs a bipedal robot, the approach to training a locomotion control architecture applies to both in this case. It was observed that the trained architecture was limited to generalizing to other situations and tended to be biased toward the Reflex Controller. This was accredited to possible dependencies within the training reward / cost structure favoring reflex control over CPG tuning. The authors did question the effectiveness of CPGs in locomotion based on their results; however, they did note that CPG did enable small modifications to the reference gait, which was key in training.

As summarized in the reviewed research, machine learning offers advantages to traditional control methodologies. Often the control architecture features hierarchical architecture featuring at least two levels of networks: 1) one for high-level motion planning, and 2) one for low-level gait control. Motion planning using machine learning allows for the autonomous system to adapt to unfamiliar terrain and obstacles, while gait control allows the system to maintain stability when faced with disturbances. This is a critical capability for systems to be deployed within environments like the lunar surface.

Machine Learning on Edge Devices

Machine learning with its advantages does come with a cost. One of those is the computational resources required to train and run inference. This can make machine learning prohibitive for applications on severely resource constrained devices, such as microcontrollers. For autonomous systems navigating the lunar surface, they face constraints of limited power, low computation resources and minimal weight. This leads to the framework of Tiny Machine Learning, or TinyML, which is focused on providing low powered artificial intelligence (AI).

TinyML is focused on developing machine learning techniques that can be deployed to microcontrollers, or similar devices. This results in a couple key benefits: 1) reduced energy consumption and 2) reduced external communication through on-board data processing (Sanchez-Iborra & Skarmeta, 2020). Both of which are desirable for an autonomous system operating on the lunar surface. Sanchez-Iborra & Skarmeta (2020) surveyed TinyML frameworks and summarized that TinyML frameworks fell into three categories: 1) conversion of already trained models to microcontrollers, 2) integrating ML libraries within the microcontroller, and 3) inclusion of a dedicated co-processor that supports the main processing unit on ML tasks. For example, De Prado et al. (2021) trains an autonomous wheeled vehicle using TinyML framework using an offline training approach. They train a tiny Convolution Neural Network (tinyCNN) through imitation learning where data is fed from the platform to a development machine to perform learning. The tinyCNN models are then deployed to the vehicle on a microcontroller. Furthermore, Ren et al. (2021) proposed an architecture to enable on-line learning, demonstrating the second category of TinyML approaches. The proposed system, referred to as Tiny Online Learning (TinyOL), provides a C++ library that can be deployed on microcontrollers. This system allows training an additional layer on the microcontroller to the static model deployed from offline training. The TinyOL layer receives input from the static model and updates the predictions based on its own parameters. This aims to address one of the main challenges of TinyML, which is adaptability of the model after deployment.

Microcontrollers are limited in storage space, memory and energy availability. Training models to fit within these constraints can be challenging. Various research has attempted to understand these limitations and explore methods for addressing them. For example, Banbury et al. (2020) proposed a differentiable neural architecture search (DNAS) to explore model space

for deployment onto microcontrollers. The study evaluated a family of models deployed on a series of microcontrollers. Model performance was characterized into latency, memory footprint (SRAM and flash), number of operations, and model accuracy. Three different tasks from the TinyMLPerf benchmark were evaluated: 1) Visual Wake Words (VWW), 2) Keyword Spotting (KWS), and 3) Anomaly Detection (AD). The study found that the number of operations correlated linearly with inference latency, and hence energy per inference. This suggests that optimizing the operation count is key when deploying to microcontrollers. Additionally, Deutel et al. (2023) proposed a neural architecture search (NAS) strategy based on Multi-Objective Bayesian optimization (MOBOpt) and a set of policies trained using Augmented Random Search (ARS) agents. The aim was to identify a NAS technique that could handle more than one objective, which is key in deploying to microcontrollers. The objectives of the search were accuracy, memory consumption and floating point operations per second (FLOPS). This aligned with the typically operating constraints of microcontrollers. The experiment performed the method using two cases: 1) image classification using ResNet18 and 2) time-series classification of human activities using MobileNetv3. These were compared with several Bayesian optimization strategies, where the proposed technique demonstrated better performance compared to other techniques. Next, Bale-Parks (2022) focused on a method for measuring power consumption and inference time when deploying TinyML models to microcontrollers. This aimed to provide more accurate measurements of power consumption compared to other measurement methods. The experiment monitored the power consumption of TinyML models deployed to an Arduino Nano BLE Sense. The TinyML model controlled the motion of a belt pulley system using on-board accelerometer and gyroscope sensor data. The power monitoring function was implemented using a Simple Linear Regression model from the Scikit-Learn

library. The power monitoring function was compared to Edge Impulse's capability for power monitoring. Unfortunately, the author was unable to develop an accurate linear regression model using this method and recommended further refining or exploring different neural network frameworks. These studies present the challenges in deploying machine learning models on microcontrollers, along with some approaches to tackling those challenges.

In addition, several studies have focused on using TinyML for more focused tasking. This primary falls into on-device inference for distributed sensing and control at edge devices. The aim is to allow for edge devices to use inference to reduce external communication and allow for smarter perception and reaction with the environment. First, Basaklar et al. (2022) investigates a solution for management energy levels of a wearable device. The proposed solution features an agent, called tinyMAN, that monitors and controls the energy allocation of the device to maintain energy neutral operation (ENO). The device architecture includes components for energy harvesting and storage, along with functional utility (energy consumption). The experiment trained the agent using a PPO algorithm in a simulated environment based on user data. TinyMAN was evaluated using three different model sizes and compared to three other approaches. It was found that TinyMAN achieved better performance compared to the other methods, and that model performance increased with model size. However, memory, energy consumption and execution time also increased with model size, which identified a trade-off between performance and low-cost model execution. Second, Bhattacharya & Pandey (2024) proposed a deployment solution for real-time monitoring and management of soil quality using TinyML framework. The aim was to identify a solution that reduced energy consumption while being responsive to changes in soil quality, which would allow for more remote monitoring of soil conditions. The proposed method combined several

methods for managing energy levels on device. The first method was applying Dynamic Voltage and Frequency Scaling (DVFS) for power management. This was configured to control the voltage and clock frequency of the device based on predicted workload of given task. The second method was Genetic Algorithm-based Sleep/Wake Strategy to control the power state of the device. The genetic algorithm was used to control the sleep/wake scheduling based on experience and environment conditions. The third method was energy harvesting to reduce the dependency on batteries as a source of energy. This method still utilizes an internal battery for local storage; however, reduces the need for external batteries to power the device, which enables more remote operations. The final method was Task Partitioning that further assisted energy management by partition tasks using a genetic algorithm. The experiment evaluated the proposed method under two scenarios: ideal, no threat scenario and threatened scenario. This aimed to understand the security of the proposed method to common network threats. In general, the experiments showed that the proposed architecture performed comparable or better than other classification methods under both scenarios. Third, Archana (2020) applied TinyML in three applications within solar energy systems. The first application was estimation of power loss on solar panels in remote locations. This aims to identify the affected region and predicted power loss due to soiling. The second application was to classify the soiling. This aided in the first application by identifying what soiling conditions required action to correct. The third application is forecasting solar irradiance to improve solar panel performance. For all three applications, a transfer learning approach was taken to train a MobileNet model on the respective dataset. An experiment for each application was conducted by deploying the model onto three different types of devices: Central Processing Unit (CPU), Vision Processing Unit (VPU) and Field-Programmable Gate Array (FPGA). For the first application, the CPU and FGPA

performed similarly, where the VPU performance degraded significantly at longer run times. For the second, the FGPA provided the best performance; however, the VPU would be more practical for remote edge computing. For the third application, the VPU demonstrated the best performance. This study demonstrated the application of deploying TinyML models to edge computing to allow for remote sensing of solar power systems. As illustrated in each of these studies, TinyML can be used to handle discrete tasking on an edge device. However, these tasks have been mainly focused on image processing for classification, energy prediction or state control, such as trigger commands

TinyML offers a promising framework for lunar exploration vehicles in that it aims to maximize energy efficiency and reduce external communication needs. Furthermore, lunar exploration is faced with an unknown environment that would require the autonomous system to adapt during operation. Using TinyML for locomotion control may offer the flexibility of machine learning, while still staying within operational constraints of the platform.

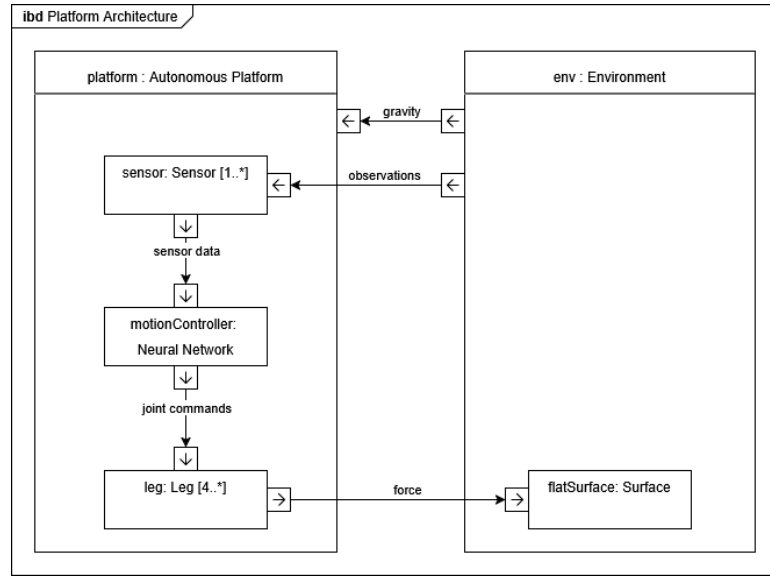
Chapter 3 - Methodology

This section outlines the research methodology employed to investigate the application of deep reinforcement learning (DRL) for controlling the motion of a legged robot, with a focus on deploying the learned control policy onto a resource-constrained microcontroller. This study employs a quantitative, experimental methodology, using simulation-based deep reinforcement learning for training and hardware-in-the-loop evaluation to assess the feasibility of deploying a motion controller for a legged robot on a resource-constrained microcontroller. The methodology encompasses three key components: the design and development of a representative robotic platform and environment simulation used for training and evaluation; the implementation of a DRL framework tailored to learn stable and efficient locomotion policies; and the strategy for deploying and demonstrating the learned model on physical hardware with limited computational resources. By integrating these elements, the research aims to bridge the gap between high-performance learning in simulation and practical deployment in real-world robotic systems.

Platform and Environment Definition

As summarized previously, autonomous systems on the lunar surface will need to navigate an unfamiliar environment with varied terrain using limited system resources. This variability drives the need for adaptability in the system's behavior and design. To address this need, the research proposes the use of a trained neural network, or agent, to control the motion of a legged robotic platform, referred to as the *motion controller model*. The design architecture is divided into three major parts: 1) the environment, 2) the platform, and 3) the motion controller model as illustrated in Figure 3.1.

Figure 3.1. Platform Architecture



Virtual representations of the platform and environment were used in the experiment's design. Using a virtual environment to host the motion controller model offers both advantages and disadvantages. One advantage is increased training speed. Reinforcement learning approaches are based around trial and error. This requires the agent to learn across multiple trials before acquiring adequate performance in a specific task. The number of required trials can range in hundreds of thousands to millions of trials. Virtualizing this allows us to leverage the computation power of the development machine to speed up the training process. Another advantage is cost and practicality. For example, this experiment aims to train an autonomous system that can move on the lunar surface. It is not practical to send an actual platform to the lunar surface to train, requiring the learning environment to be simulated. However, the main disadvantage is the "reality-gap" between the virtual environment and the real-world. The virtual environment is made up of modeled versions of reality. Models are abstract representations of reality that introduce inaccuracies. This can lead to biases in the motion controller model when deployed into a real environment on a physical platform. Additional adjustments may be required

to fine-tune the motion controller model after deployment, such as supplemental on-device training. The focus of the research is the ability to deploy the motion controller model to a microcontroller with the primary purpose of controlling the motion of a legged robot. Using a virtual environment and platform was sufficient to demonstrate this capability. However, it should be noted that the agent may not perform as well on a physical platform due to this “reality-gap”.

Figure 3.2. Physical Architecture

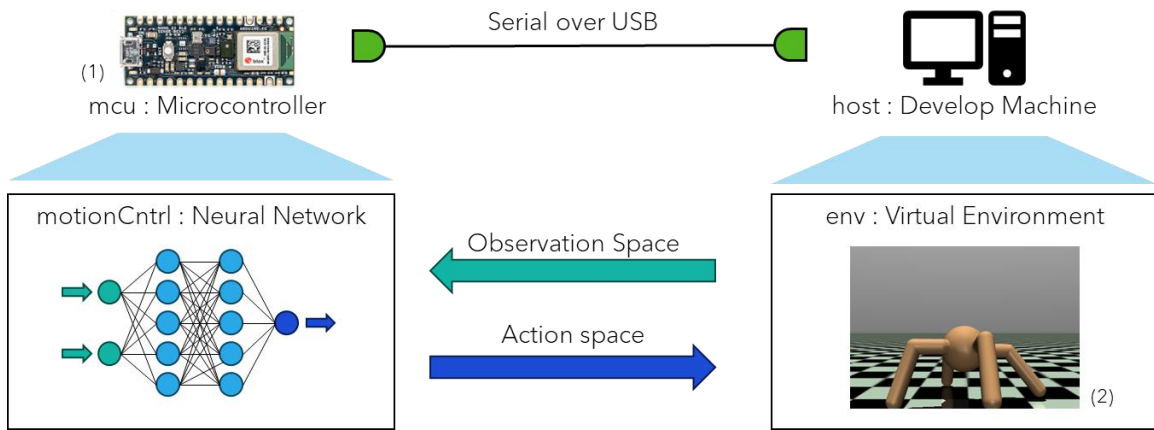


Figure 3.2 illustrates the physical configuration of the experiment. A development machine was used to host the virtual environment and platform during training and evaluation. Once the motion controller model was trained, it was deployed to the microcontroller. During evaluation, the microcontroller would run inference using the motion controller model and communicated back to virtual environment hosted on the development machine. This architecture allowed for the function of the microcontroller to remain limited to just serial communication and running of the motion controller model.

The Ant environment from the Gymnasium project (Kwiatkowski et al., 2024) was used as a virtual environment and platform. This environment uses the Multi-Joint Dynamics with Contact (MuJoCo) physics engine (Google DeepMind, n.d.), which allows the agent to learn in a physics-informed environment. The Ant environment features a flat plane for the platform to interact with, which limits the ability of the motion controller model to adapt to more varied terrain. However, the focus was to evaluate the performance of the motion controller model running inference on a microcontroller with the objective to control the platform’s movement. The environment offers up the complexity of the motion controller to send commands to each individual joint on the platform, while monitoring contact forces with the surface. This creates an action and observation space that is continuous in nature, which was key aspect of the task levied on the motion controller model (Schulman et al., 2018). Furthermore, the Gymnasium API and MuJoCo offers the ability to define custom environments that allow for different arenas, tasks, and platform design. Follow-on research can build these environments to provide a more robust training curriculum for the platform (Azayev & Zimmerman, 2020). This research is focused on using the Ant environment as a baseline to compare the performance after deployment onto the microcontroller.

The platform within the Ant environment is a simplified legged vehicle. The platform is composed of a single, spherical body and four legs. Each leg contains two hinge joints that can be articulated independently. This architecture introduces a complexity in that the simultaneous operation of each leg creates an action space comprised of the articulation commands to each joint at a given state. The action space is defined as

$$A(t) = \{ \tau_{hrb}, \tau_{krb}, \tau_{hlf}, \tau_{klf}, \tau_{hrf}, \tau_{krf}, \tau_{hlb}, \tau_{klb} \} \quad (4)$$

where τ is the applied torque to the joint. The joint nomenclature is identified as h / k for *hinge / knee*, r / l for *right / left* and b / f for *back / front*, respectively. This represents a continuous action space with actions bounded between $[-1, 1]$. The action space is generated from running inference through the motion controller model based upon the provided observation space.

The observation space is limited by the ability of the platform to sense its environment. This means external observations are not provided to the platform, and the platform relies on its on-board sensor suite. In this design, the observation space is assumed to be able to measure the position and velocity of each body part.

$$O(t) = \{z_T, \theta_{wT}, \theta_{xT}, \theta_{yT}, \theta_{zT}, v_{xT}, v_{yT}, v_{zT}, \omega_{xT}, \omega_{yT}, \omega_{zT}, \theta_i, \omega_i\}, \text{ where } i = [2, 13] \quad (5)$$

For the torso, the observation space includes the vertical position z , orientation θ , linear velocity v , and angular velocity ω . For each joint i , the angular position θ and angular velocity ω is provided. This creates a continuous observation space with a size of 27 elements over the interval $(-\infty, \infty)$.

The last piece of the virtual environment is the objective or task to be completed. The reward function is developed to train the agent in completing this objective. The Ant environment objective is for the platform to move in a forward direction by coordinating the motion of the four legs, where the following reward function was used.

$$R_{total} = R_{healthy} + R_{forward} - C_{ctrl} - C_{contact} \quad (6)$$

The total reward, R_{total} , is based on four terms. The healthy reward, $R_{healthy}$, was awarded each timestep that the platform remained healthy. The platform had to meet the following two conditions to be considered healthy: 1) all of the state space values were finite, and 2) z -coordinate of the torso was within the defined closed interval. This reward promoted the platform to maintain a favorable orientation and torso height above the surface. The forward

reward, $R_{forward}$, was awarded each timestep the platform moved in a forward direction. This reward could become negative and penalize the platform if the motion resulted in a backward motion. The control cost, C_{ctrl} , was a negative reward to penalize the platform for taking large actions. The contact cost, $C_{contact}$, was a negative reward to penalize the platform for exerting large contact forces. Both costs were to limit the platform from applying large torque to the joints, which could result in damaging the platform in a real environment.

To summarize, a virtual environment was used to train and evaluate the motion controller model to leverage the computation power of the development machine due to limited resources on the microcontroller. The functionality of the microcontroller focuses on running model inference and communication with the development machine. The Ant environment was chosen as it featured a legged vehicle with individual joint control using the physic-engine, MuJoCo. This closely aligns with the research objective of training a motion controller model to control the motion of a legged robot. Next, we will discuss how the model was trained, deployed and evaluated.

Research Process

This section describes the process for training, deploying, and evaluating the motion controller model. The process begins with training the motion controller model, followed by the conversion and deployment of the motion controller model to the microcontroller, and ending with the evaluation of the motion controller model running on the microcontroller, as shown in Figure 3.3.

The training framework is composed of two parts: 1) the machine learning algorithm, and 2) the learning environment. As described in **Deep Reinforcement Learning (DRL)** section, the PPO algorithm was chosen as the machine learning algorithm used for training the motion

controller model. The implementation provided by Stable Baselines3 (Raffin et al., 2021) was used to train the agent with the training parameters defined in Table 3.1. The learning environment is composed of the task or objective and the virtual environment and platform. This was implemented through the Gymnasium API (Kwiatkowski et al., 2024) that provided a connection point between the machine learning algorithm and the virtual environment and platform. To leverage existing baseline data, the Ant environment was selected, which uses the MuJoCo simulation engine. For details about the virtual environment, see **Platform and Environment Definition** section.

A custom Python script was developed to run the training process, which resulted in a motion controller model based on the PyTorch framework. Overall, the implementation focused on the integration of open-source resources to train the motion controller model. This involved initializing the Gym environment, calling the training algorithm, logging the training data, and saving the motion controller model as a PyTorch model. The Stable Baselines3 PPO algorithm handled the training loop, which allowed the research to focus on developing the deployment implementation over the learning algorithm implementation. In addition, Stable Baselines3 provided some benchmarks for the PPO algorithm using the PyBullet environment, which could be used in comparison to the results of this experiment. See **Research Algorithms** section in the appendices for details about the training script.

Figure 3.3. Research Process

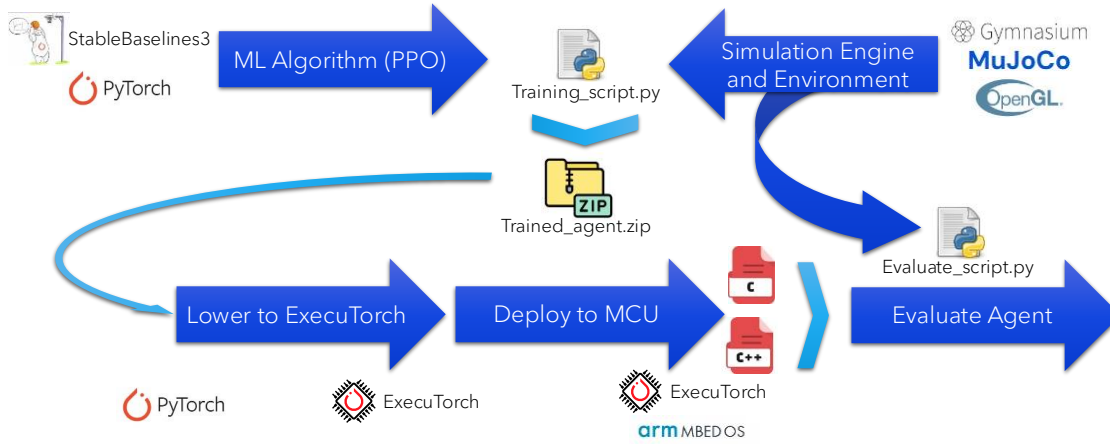
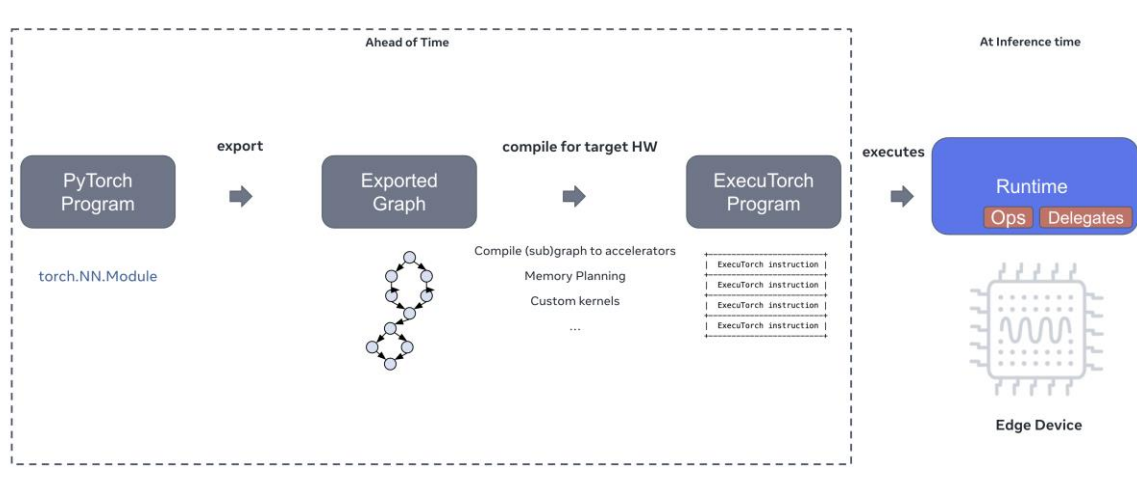


Table 3.1. Training Parameters

Parameter	Value
Number of environments	4
Learning rate	0.0003
Total timesteps	2,000,000
Number of steps	2048
Batch size	64
Number of epochs	10
Gamma	0.99
GAE lambda	0.95
Clip range	0.2

The deployment process is composed of three parts: 1) lowering the PyTorch model to an ExecuTorch Program, 2) building the runtime application and libraries, and 3) flashing application and libraries to the microcontroller. The first step in this process is to lower the motion controller model from a PyTorch model to an ExecuTorch Program. This involves a series of steps aimed at converting the motion controller model to be compatible with the ExecuTorch runtime and optimizing the model to reduce the size of the model. During the process the PyTorch model is captured as a graph, which represents the motion controller model as a series of operators.

Figure 3.4. ExecuTorch Concept ([from ExecuTorch](#))



Once the model has been lowered, it is integrated into an application that will invoke the ExecuTorch runtime and run inference using the motion controller model. In this experiment, the microcontroller remains tethered to the development machine during operation over a USB connection. A serial interface emulated across the USB connection connects the microcontroller application with the virtual environment running on the development machine. This creates the process where the observations generated by the virtual environment are sent to the microcontroller and passed to the model. The model runs inference with the given set of observations and generates a set of actions. The actions are then passed back to the virtual environment from the microcontroller, and the virtual environment updates its state. A new set of observations are generated, which marks the start of the next iteration of the process.

To implement this concept, the ExecuTorch runtime and MBed OS (Arm Inc, n.d.) were integrated together to run the microcontroller application. ExecuTorch runtime provides the software framework to run the model inference using capabilities found within microcontrollers and other edge devices. The microcontroller used in this experiment was an Arduino Nano BLE 33 Sense Rev2 (Arduino, n.d.). While Arduino provides its own version of the MBed OS for the microcontroller, integrating ExecuTorch with Arduino proved to be difficult, and the decision

was made to integrate ExecuTorch and MBed OS directly (*Mbed-Torch-Fusion-Os*, n.d.). Furthermore, integrating it in this way should make this approach adaptable to other boards that use the MBed OS beyond the Arduino framework. The microcontroller application was a simple program focused on maintaining serial communication and running model inference. The program followed the process of initializing and configuring both the serial interface and the motion controller model, monitoring the serial interface, running model inference upon receiving the next observation set, sending the action set produced by the motion controller model inference. For details, see **Research Algorithms** in the appendices for the application pseudocode.

Table 3.2. Software Configuration

Software	Version
ExecuTorch	0.6.0
Gymnasium	1.0.0
MBed OS	6.17.0
StableBaselines3	2.5.0

The last step in the research process is to evaluate the motion controller model’s performance. The step consists of two phases. The first phase evaluates the motion controller model as configured immediately after training prior to applying any optimization/quantization techniques. The aim is to provide a baseline to evaluate the motion controller model after deployment to the microcontroller. The motion controller model is executed on the development machine with the virtual environment. The second phase evaluates the motion controller model deployed to the microcontroller. This would represent the targeted configuration of motion

controller model after quantization/optimization on the microcontroller. The results from this phase were compared to the baseline captured in the first phase.

The evaluation script for both phases used the same objective and flow, only accounting for differences in required communication between the virtual environment and motion controller model based on physical deployment. The objective of the task was for the motion controller model to guide the platform down a straight path while maintaining a healthy state. The evaluation process began by establishing a virtual environment and platform to a defined set of initial conditions. Once initialized, the observations were sampled from the environment. These observations were logged and passed to the motion controller model. Logging the observations served as a measurement point for determining performance parameters. The motion controller ran inference to determine the set of joint torques to apply (the action set). The environment stepped forward using the new action set and generated a new set of observations. The performance parameters were updated using the new and previous observations to determine the performance of that timestep and logged. The process was repeated until an episode end condition was reached. At which point, the performance parameters for the episode were recorded for post evaluation analysis. The evaluation process executed iteratively based on the specified number of evaluations steps provided to the evaluation script. See **Research Algorithms** section within the appendices for script pseudocode.

Chapter 4 - Results and Discussion

The data collected from the demonstrations was used to evaluate the effectiveness of training a neural network to control the motion of a legged platform across a flat surface. The expectation was to train a motion controller model embedded in a virtual platform to successfully reach the end goal. Model performance was evaluated to compare the motion controller model's performance before and after deployment to the microcontroller. This aims to identify performance impacts by optimizing the motion controller model to run on a microcontroller. The focus was not to develop a highly optimized motion controller model. The performance of the motion controller model may be inadequate for applications and require further training. If proven successful, this insight will be valuable in developing autonomous system for deployment in unknown and resource constraint environments, such as the lunar surface.

This section documents the results of the experiment. The experiment focused on the performance of the motion controller model in accomplishing the task. The performance was evaluated using the PyTorch configuration of the motion controller model on the development machine against the converted configuration deployed on the microcontroller. The following performance parameters were derived from the objective, which provided a quantitative measure of the motion controller's performance.

- Average episode length (seconds)
- Forward distance traveled (meters)
- Path distance (meters)
- Average joint torques (Newton-meters)

The experiment evaluated the motion controller model using a simulated environment and platform. Two configurations of the motion controller model were evaluated: 1) the

configuration prior to conversion and run on the development machine, labeled “PyTorch”, and 2) the configuration after conversion and run on the microcontroller, labeled “MCU”. Data was collected over thirty episodes for each configuration. Each episode featured the same termination conditions as those defined in training since the same environment configuration was used for both training and evaluation. The task of the episode was to move the platform in a forward motion (x – coordinate direction) without triggering a termination condition. The episode was truncated at 1000 timesteps to prevent the episode from running indefinitely. The evaluation aimed to see how far in the forward direction the motion controller model could control the movement of the virtual platform within the specified duration.

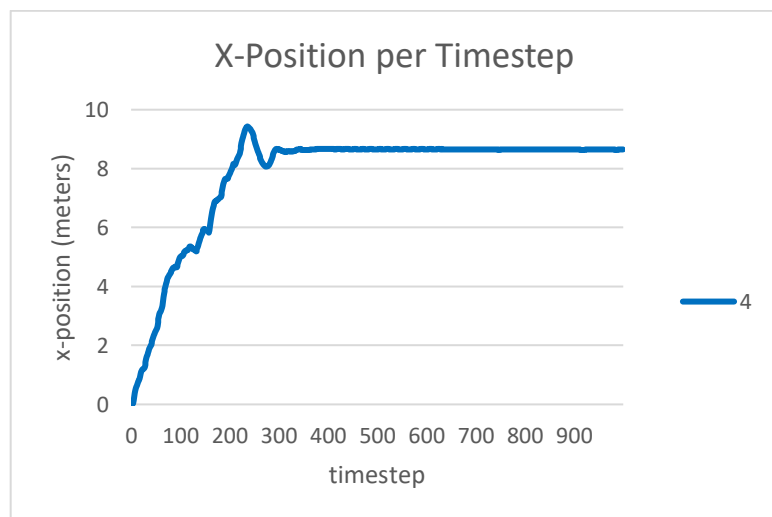
Note, it was visually observed in several episodes where the episode would not terminate after the platform flipped over and continued to run until reaching the maximum timesteps. To account for these scenarios, each episode’s data was reviewed for indications of a flipped platform situation. Data collected after the flipping event was removed from the data set. The following indications were used to determine when the platform flipped over during an episode. First, the x-position was assessed for the behavior where the position remains relatively the same on subsequent timesteps. From here it was observed that when the platform flipped, the x-position would feature a sinusoidal curve. This is a result of the platform flipping over, achieving a maximum x-position before rocking backward and forward along its spherical body due to momentum prior to coming at rest. Once an episode was a suspect for flipping over, the x-coordinate velocity and x- and y-orientation were observed to confirm the event. As shown in Figure 4.1 (C) and (D), the flipping event exhibited a reduction in x-coordinate velocity as the platform assumed a rest state, and the orientation would rapidly offset and remain offset due to not coming to rest in a forward and upright facing position. After concluding that the episode

features a flipping event, the x-position data was assessed to identify the timestep of the local peak during the flipping event. This was assumed to mark the position prior to the initial roll backwards. While this did include additional data where the platform was not exactly in an upright position, it was assumed negligible to allow for simplicity in parsing the data. After this timestep, the data was removed from the data set used in calculations.

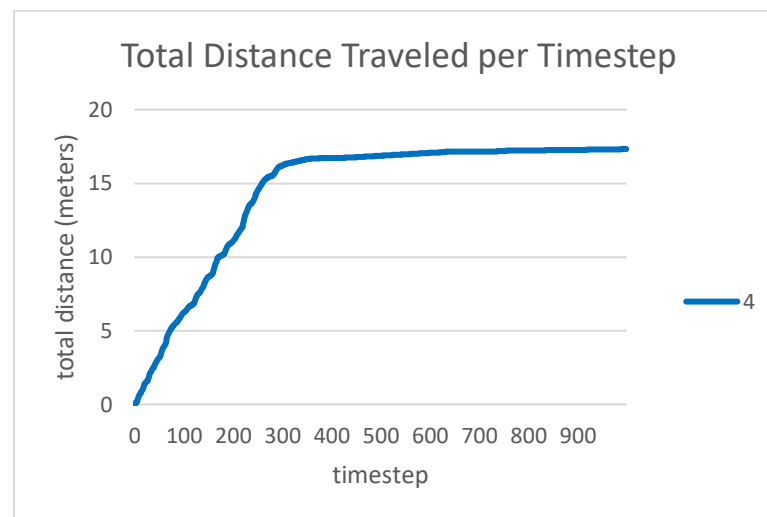
Table 4.1: Performance Overview

	PyTorch Configuration (PyTorch)	Microcontroller Configuration (MCU)
Average episode length (seconds)	5.59	6.23
Forward distance traveled (meters)	26.98	27.07
Path efficiency (Δx / distance)	0.75	0.71
Average joint torques (Newton-meters)	0.02	0.02

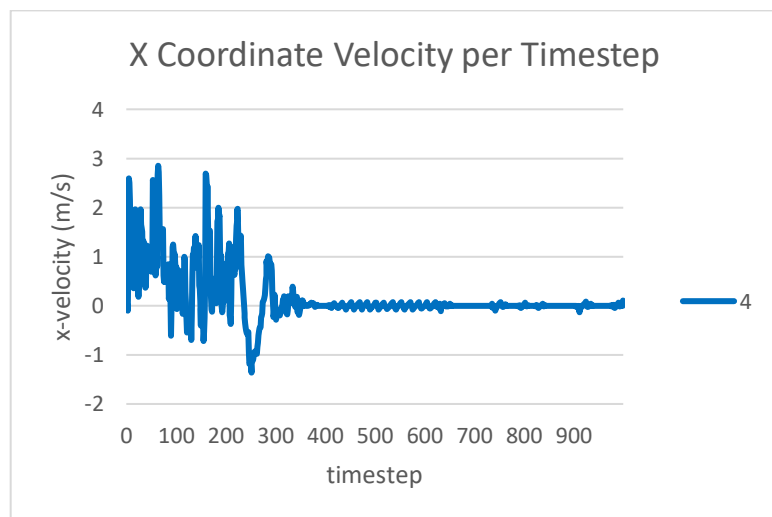
Figure 4.1. Episode Data from Flipped Platforms



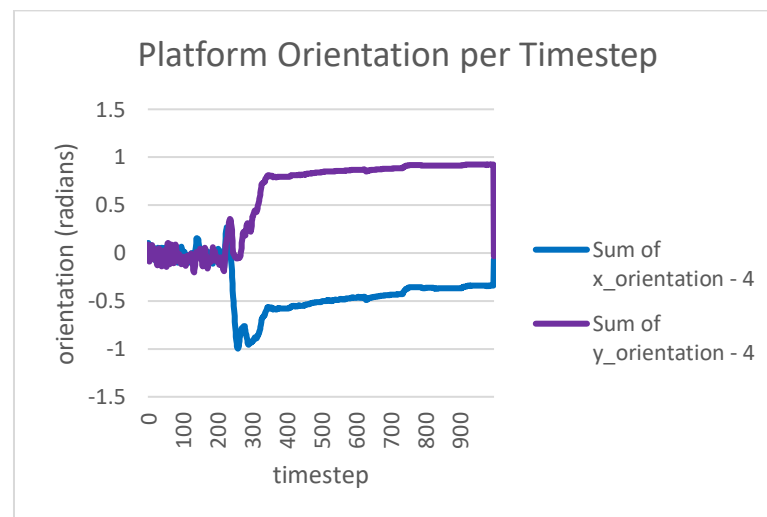
(A)



(B)



(C)



(D)

Average Episode Length

The *average episode length* represented the total timesteps of the episode. This provided a measure for how long the motion controller model could maintain the platform in a healthy state. The conditions for a healthy state were the same as those used in training since the same environment was used in evaluation. The only difference is that the evaluation was performed using a single environment instead of multiple parallel environments. The environment truncated the episode after 1000 timesteps, which limited the maximum timesteps. Each timestep was equivalent to 10 milliseconds, which left the maximum episode length at 10 seconds.

Table 4.2: Average Episode Length

Configuration	Average Episode Length	Standard Deviation	Percent Complete
MCU	623 (6.23 seconds)	363	33%
PyTorch	559 (5.59 seconds)	345	27%

The MCU configuration outperformed the PyTorch configuration in episode duration, maintaining a healthy state of the platform up to 623 timesteps on average, which is a 10.85% increase compared to the PyTorch configuration. However, there was significant variability in the episode length data. This is most likely due to the model performance overall as both configurations exhibited similar variability. Additional evaluation of the training performance and tuning of the training parameters is recommended to achieve lower variability in the performance. In addition, the overall episode length was truncated at 1000 timesteps due to the configuration of the environment. For episodes that reached the maximum timesteps, these were identified as completing the tasks successfully. The MCU configuration achieved a 6% higher completion percentage compared to the PyTorch configuration. Overall, both achieved a lower

than 50% success rate, which is not ideal for actual deployment. The reliability of the motion controller to prevent failures, such as flipping over, would need to be improved. Once again, this is attributed to the training performance since both configurations exhibited similar behavior.

Forward Distance Traveled

The *forward distance traveled* represented the maximum distance achieved in the forward position (x-coordinate) based on the starting position. This provided a measure of performance on how far the platform traveled while maintaining a healthy state. While the episode duration was truncated to 1000 timesteps, the x-position was not limited, and the platform could travel as far as possible within the time limit.

Table 4.3. Average Forward Distance Traveled

Configuration	Average Forward Distance (meters)	Standard Deviation	Max Distance (meters)	Average (x) Velocity (meters/second)
MCU	27.07	17.66	57.01	0.89
PyTorch	26.98	16.88	51.67	0.94

Both configurations achieved an average distance of about 27 meters and were able to reach distances between 40-50 meters at maximum episode length. The MCU configuration reached a peak of about 57 meters. There was significant variability in the distances reached. A key factor is the coupling of distance with episode duration. For episodes that terminated early, the distance was shorter than those that ran longer. This resulted in a large range of distances being reached. On the other hand, the average instantaneous x-velocity per episode provided a measure independent of the total episode length. From this perspective, the configuration that exhibited higher speeds would theoretically achieve greater distances within the maximum episode length. This provided a more equivalent comparison between episodes that terminated early with those that completed successfully. To acquire an overall x-velocity of the

configuration as listed in Table 4.3, the instantaneous x-velocity at each timestep was recorded for each episode during simulation. Then, the instantaneous x-velocity at each timestep was average across all timesteps for an episode to provide an average instantaneous x-velocity for that episode.

$$\bar{u}_e = \frac{1}{T} \sum_{t=1}^T u_t \quad (7)$$

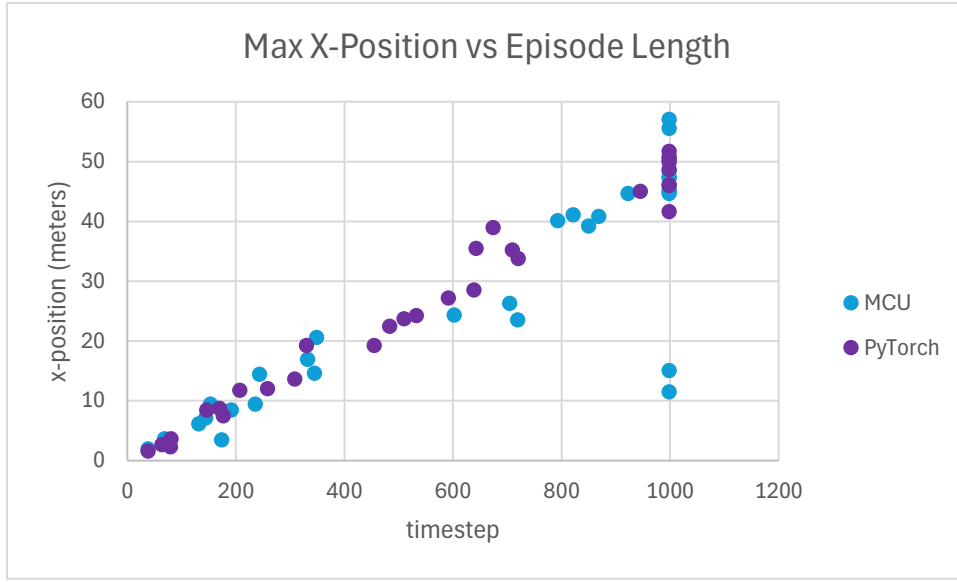
where, u is the instantaneous x-velocity, T is the total number of timesteps, and t is the timestep. Next, the average instantaneous x-velocity for each episode was averaged across all episodes to provide an average x-velocity for the configuration as listed in Table 4.3.

$$\bar{u}_{configuration} = \frac{1}{n} \sum_{e=1}^n \bar{u}_e \quad (8)$$

where, n is the total number of episodes. On average, the PyTorch configuration exhibited greater performance in forward speed compared to the MCU configuration, which contrasts with the episode length and maximum forward distance traveled data.

To explore this behavior, the maximum forward position was plotted against the episode length, as illustrated in Figure 4.2. The PyTorch configuration data provided a more linear fit in maximum distance reached per episode length, which would contribute to the better forward speed performance as the slope was held more consistent between episodes. The MCU configuration exhibited more variability in maximum distance reach, especially at longer episode lengths, where a couple of episodes did not exceed 20 meters after 1000 timesteps. This resulted in greater variation in forward speed of each episode, with a tendency to achieve slower forward speeds.

Figure 4.2. Maximum x-Position vs. Episode Length



Path Distance

The *path distance* represented the total amount of distance traveled. The *path distance* varied from the *forward distance traveled* parameter in that the platform may not follow a straight path or backtrack. This provided a sense of how much work the platform exercises in achieving the *forward distance traveled*. The total distance was calculated and recorded at each timestep during runtime.

$$total\ distance = \sqrt{(x_t - x_{t-1})^2 + (y_t - y_{t-1})^2} \quad (9)$$

Figure 4.3 shows the average total distance per timestep across the episodes. Both configurations traveled similar distances up to around 450 timesteps, at which time they began to deviate. The PyTorch configuration, on average, traveled greater distances than the MCU configuration. This is partly due to the MCU configuration achieving lower x-position per timestep compared to the PyTorch configuration, as shown in Figure 4.5. As discussed in the previous section, the MCU

configuration exhibited greater variation in maximum forward distance traveled, featuring several episodes that traveled far less than the PyTorch configuration. That variation is resulting in the MCU configuration achieving smaller x-position reached at later timesteps, and hence lower total distance traveled.

Furthermore, both x- and y-position affected the total distance performance. So, achieving higher total distance did not equate to more efficient performance. Ideally, the total distance would be equal to the x-position at each timestep. This would mean that platform moved forward (positive x-coordinate) and did not deviate to the side (y-coordinate) or backtrack (negative x-coordinate). To evaluate the efficiency of the configuration in taking a straight, forward path, the x-position was divided by the total distance at each timestep to get an efficiency ratio. Value of 1 would indicate the platform moved purely in the positive x-direction, where value of 0 would indicate no movement in the x-direction. Negative values were experienced at the beginning when the platform moved backward beyond the original starting x-position. As shown in Figure 4.4, the platform exhibited low efficiency at the beginning of the episode; however, it quickly increased to around 0.7-0.75. This is attributed to the controller gaining a sense of the platform and environment prior to establishing a rhythm of forward motion. Once again, the performance is close between the two configurations with the PyTorch configuration exhibit slightly better path efficiency.

Evaluating the average y-position per timestep provided insight into the difference in path efficiency between the two configurations. As shown in Figure 4.6, both configurations experienced motion along the y-coordinate, indicating the platform did not move solely in a forward motion. Both configurations exhibited consistent drifting to one side as it moved forward. The PyTorch configuration exhibited less drifting than the MCU configuration. This is

mainly observed after timestep 300, where the MCU configuration began to deviate from the PyTorch configuration. To further visualize the path,

Figure 4.7 illustrates the average path taking by both configurations. It validates the consistent drift to one direction for both configurations. The greater path efficiency observed in the PyTorch configuration correlates with the lower path deviation.

Overall, the path performance of the MCU configuration was comparable to that of the PyTorch configuration. The path efficiency of the MCU configuration was only about 0.04 lower than that of the PyTorch configuration. This is mainly observed to be a factor of the degree of sideways (y-coordinate) drifting in one direction found in the MCU configuration. In practice the drifting would need to be corrected to allow the platform to reach a specific destination. Since both configurations exhibited drifting behavior in the same direction, it was assumed that this learned behavior from the training.

Figure 4.3. Average Total Distance per Timestep

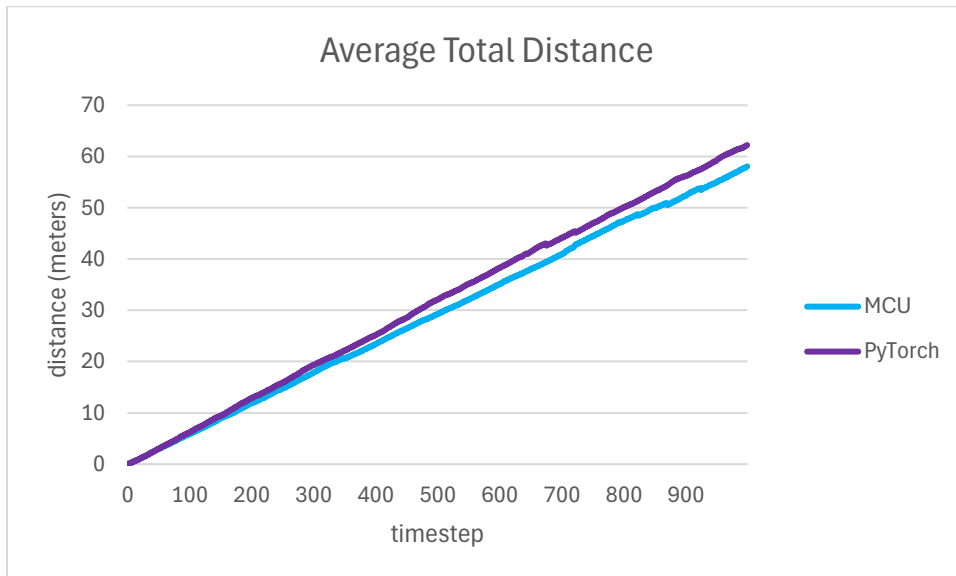


Figure 4.4. Ratio of x-Position to Total Distance

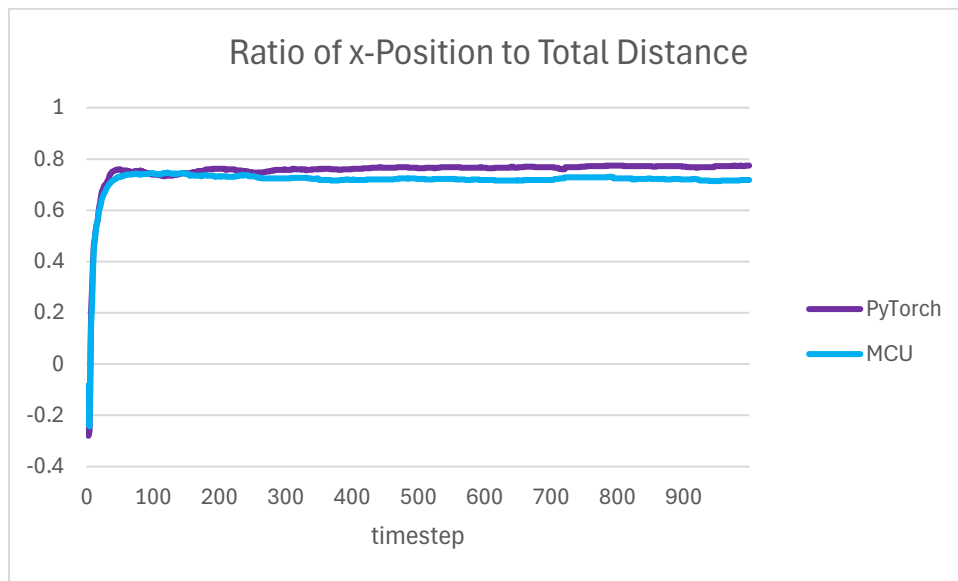


Figure 4.5. Average x-Position per Timestep

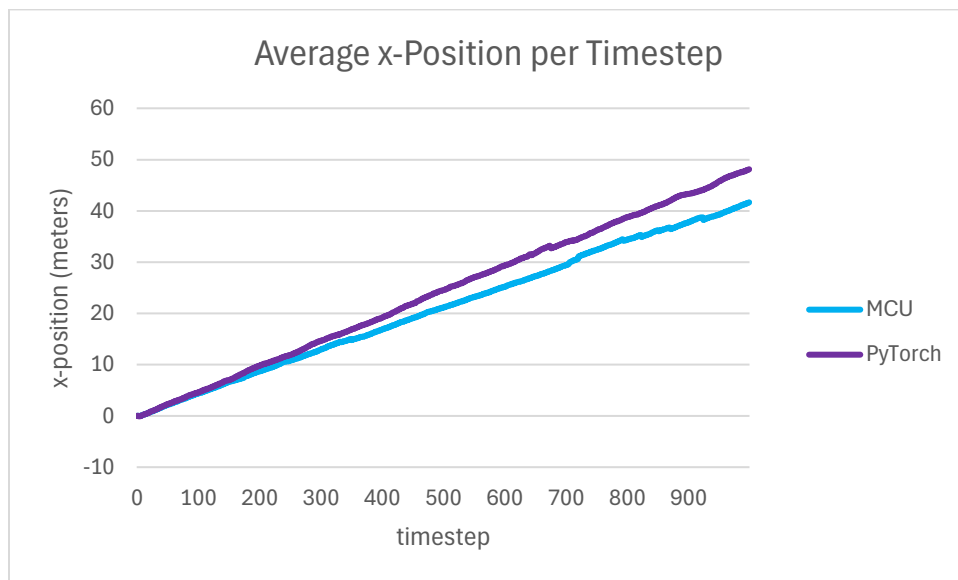


Figure 4.6. Average y-Position per Timestep

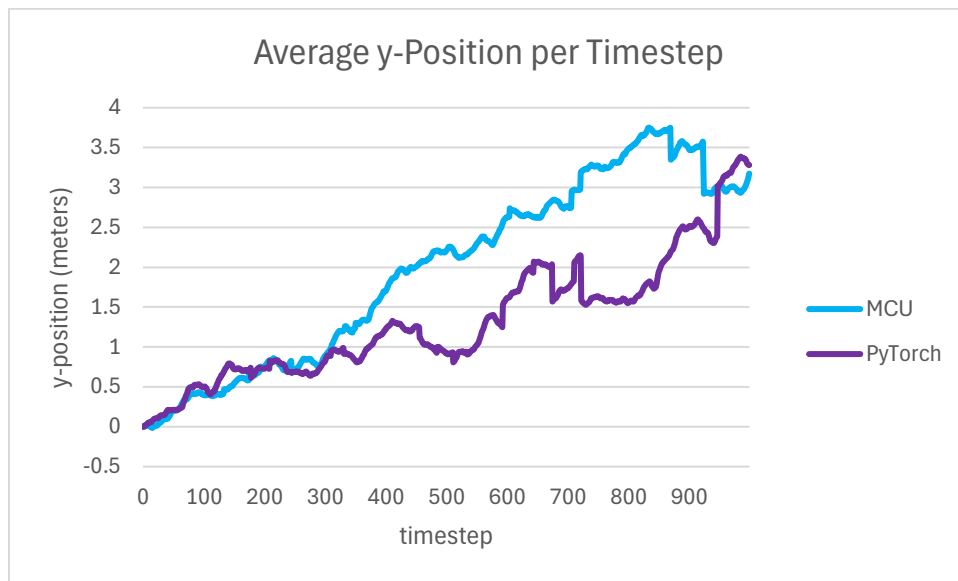
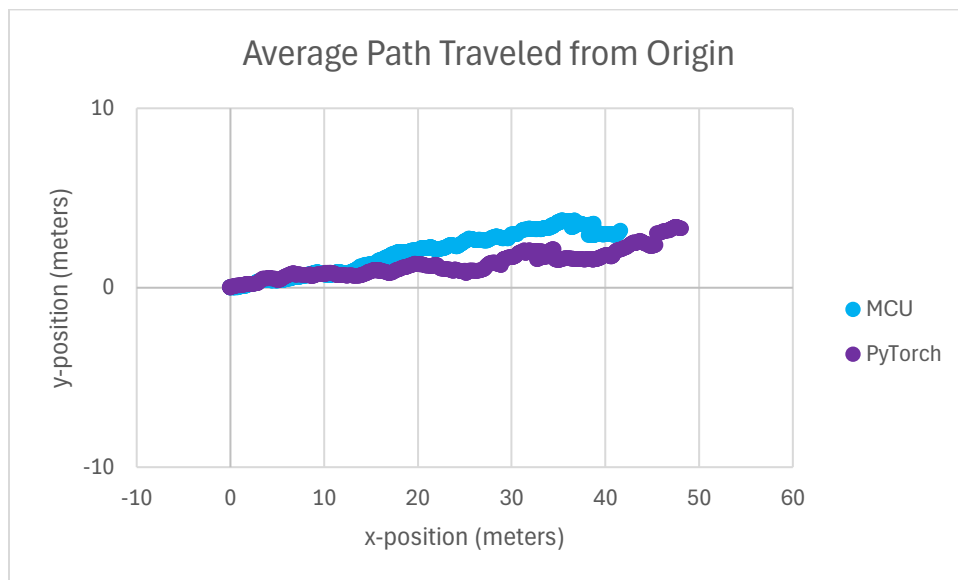


Figure 4.7. Average Path Traveled from Origin



Mean Joint Torques

Another way of assessing the amount of work the platform exerted to achieve a forward motion was to evaluate the torque applied to the joints during the episode. This was accomplished by calculating the mean of the applied torque across all eight joints at a given timestep, referred to as the *mean joint torque*, providing an average magnitude of torque applied by each joint at that timestep.

$$\bar{\tau}_t = \frac{1}{J} \sum_{j=1}^J \tau_j \quad (10)$$

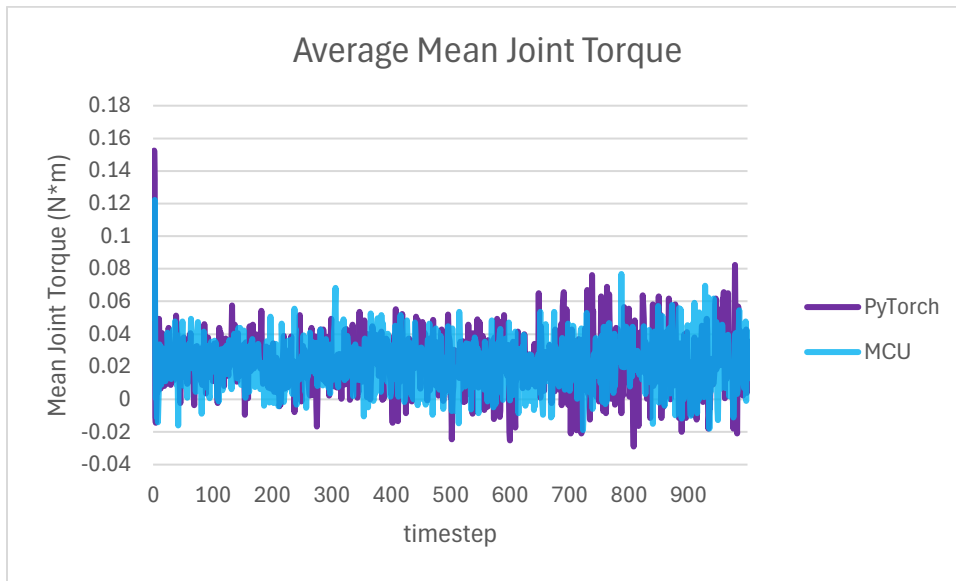
where τ is the joint torque, t is the timestep, and J is the total number of joints. The *average mean joint torque* represented the average *mean joint torque* across the sample of episodes.

$$\bar{\tau}_{t,avg} = \frac{1}{n} \sum_{e=1}^n \bar{\tau}_{t,e} \quad (11)$$

where n is the total number of episodes, t is the timestep, and e is the episode. This parameter was evaluated to see any impacts to the action space resulting from the model conversion process. While the applied torque at a given timestep would greatly depend on the environmental conditions, the aim was to observe overall trends in torque behavior that would indicate a change in model performance.

Overall, the model conversion process and execution on the ExecuTorch runtime had minimal to no impact on the action space performance. As shown in Figure 4.8, the MCU configuration exerted similar joint torque compared to the PyTorch configuration. On average, both configurations exerted between 0-0.05 N*m per joint at each timestep.

Figure 4.8. Average Mean Joint Torque per Timestep



Chapter 5 - Conclusions

This section presents a summary and conclusions drawn from the findings of the study. Additionally, it describes the limitations within the study and offers recommendations for future research related to those limitations.

Limitations and Future Research

While this study was able to successfully demonstrate the capability of training a motion controller for a legged platform and deployment of the motion controller model onto a microcontroller, there are some notable limitations of that would prevent the application onto actual platforms and deployment in targeted environments. Summarized below are the limitations and recommendations for future research based on this study.

1. **Limitation:** The study used a simplified model of the platform that did not exhibit features of a physical platform, such sensors and actuators. This leads to significant performance gaps due to the “reality-gap” translation going from simulation to reality.

Recommendation: Development of a more accurate model of platform to support training and deployment on a physical platform. This will require changes to the training environment to provide the appropriate sensor data and actuator commands. Additionally, it offers the opportunity for on-device learning to fine-tune the motion controller after deployment.

2. **Limitation:** The study used a single and simplified learning environment, the flat plane. This limits the trained model from application to other environments that feature more challenging terrain features.

Recommendation: A training curriculum that uses a variety of environments that allow for learning on different terrain types. This would include varying environmental factors, such as gravity and atmospheric conditions.

3. **Limitation:** The study used a task limited to forward movement. Research has shown that gait patterns can be effective in legged locomotion, especially in adapting to different terrain or scenarios.

Recommendation: Using a learning curriculum that allows the agent to learn different gait patterns. This may require a hierarchical or distributed architecture approach to motion control architecture; however, different gait patterns can allow for optimized performance for a specific task.

4. **Limitation:** The study did not explore fine-tuning the training performance or evaluate other training algorithms (i.e, SAC, DDPG, etc). This influenced the overall model performance and reduced the capability to deploy onto a physical platform.

Recommendation: Exploration of different training algorithms and parameter configurations to provide more reliable performance.

5. **Limitation:** Due to time constraints, the study did not explore quantization of the model during the conversion process. This could lead to differences in the converted model performance, along with size and resource utilization.

Recommendation: Incorporate quantization techniques into the model conversion process and evaluate the effects on model performance.

Future research should focus on expanding the simulation environment to include variable terrain, slope gradients, and lunar regolith dynamics to close the “reality gap” between virtual and physical performance. Additional studies could investigate adaptive gait learning

through hierarchical or distributed control architectures, enabling more natural locomotion transitions and improved stability. Incorporating on-device fine-tuning and quantization-aware training would allow the model to optimize performance post-deployment while maintaining energy efficiency. Furthermore, integrating physical hardware, such as sensors and actuators, would provide critical insights into hardware-software interaction under constrained computing and power conditions.

The outcome of this research position it well for dissemination within robotics, space systems, and embedded AI communities. Appropriate venues for publication and presentation include *Advanced Intelligent Systems*, *IEEE Robotics and Automation Letters (RA-L)*, *Frontiers in Robotics and AI*, *Journal of Intelligent & Robotic Systems*, and *Journal of Field Robotics*. Additionally, conferences such as the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), *IEEE International Conference on Robotics and Automation (ICRA)*, *International Conference on Space Robotics (iSpaRo)*, *Space Robotics Workshop (IEEE SMC-IT/SCC)* and the *AIAA SciTech Forum* provide ideal opportunities to present extensions of this work to researchers focused on autonomous mobility and space robotics.

Summary

This thesis investigated the feasibility of applying deep reinforcement learning (DRL) to train a neural network motion controller for a legged robotic platform and deploying that controller on a resource-constrained microcontroller. The research was motivated by the mobility challenges posed by the lunar surface—uneven terrain, abrasive regolith, and strict power and communication constraints—that demand autonomous, adaptive, and lightweight robotic solutions.

Using the MuJoCo-based Ant environment and the Proximal Policy Optimization (PPO) algorithm, a motion controller model was successfully trained to generate locomotion policies. The trained model was then converted and deployed to an Arduino Nano 33 BLE Sense microcontroller via the ExecuTorch framework. Comparative evaluation demonstrated that the deployed model was able to maintain performance similar to the PyTorch baseline, with only modest degradation due to conversion and hardware limitations. Key performance measures—including episode duration, forward distance traveled, and path efficiency—showed that while variability and instability were present, the microcontroller-based system retained the essential locomotion capabilities of the trained model. Importantly, joint torque analysis confirmed that the conversion process did not introduce significant distortions in the action space.

The contributions of this study are threefold. First, it provides a practical demonstration of bridging high-performance DRL training environments with real-time embedded execution, delivering insight on the critical “reality gap” between simulation and deployment. Second, it outlines a replicable workflow for training, converting, and deploying neural controllers on low-power hardware, offering a roadmap for TinyML applications in extraterrestrial robotics. Third,

it highlights both the promise and current limitations of DRL-based mobility control when constrained by microcontroller resources.

The study’s limitations, such as reliance on a simplified simulation environment, a restricted set of tasks (forward locomotion), and the absence of on-device adaptation, underscore the challenges of moving toward robust, real-world lunar applications. However, these constraints also open productive directions for future research, including the development of richer training environments with varied terrains, the exploration of gait-based motion strategies, fine-tuning of DRL hyperparameters, and integration of quantization techniques to further optimize deployment.

In conclusion, this thesis demonstrates that deep reinforcement learning, when paired with TinyML deployment strategies, holds significant potential for advancing autonomous robotic mobility in extreme environments. While additional research is required to bridge the remaining gaps between simulation and lunar reality, the findings here establish a foundation for scalable, adaptive, and efficient robotic systems capable of supporting humanity’s long-term exploration goals beyond Earth.

Bibliography

- ANYbotics. (n.d.). *ANYmal Overview*. Retrieved August 13, 2025, from <https://www.anybotics.com/robotics/anymal/>
- Aradi, S. (2022). Survey of Deep Reinforcement Learning for Motion Planning of Autonomous Vehicles. *IEEE Transactions on Intelligent Transportation Systems*, 23(2), 740–759. <https://doi.org/10.1109/TITS.2020.3024655>
- Archana, V. (2020). *Tinyml for Solar Panels: Bringing Edge Computing Applications to Solar Energy Systems* [National University of Singapore]. <https://www.proquest.com/pqdtglobal/docview/2723855919/D9677B52EA384A1BPQ/13?accountid=11789&sourcetype=Dissertations%20&%20Theses>
- Arduino. (n.d.). *Nano 33 BLE Sense Rev 2*. Arduino Documentation. Retrieved May 9, 2025, from <https://docs.arduino.cc/hardware/nano-33-ble-sense-rev2>
- Arm Inc. (n.d.). *Mbed OS | Mbed*. Retrieved May 9, 2025, from <https://os.mbed.com/mbed-os/>
- Azayev, T., & Zimmerman, K. (2020). Blind Hexapod Locomotion in Complex Terrain with Gait Adaptation Using Deep Reinforcement Learning and Classification. *Journal of Intelligent & Robotic Systems*, 99(3–4), 659–671. <https://doi.org/10.1007/s10846-020-01162-8>
- Bale-Parks, P. (2022). *Examining Energy Use in Tinyml Systems* [University of Colorado]. <https://www.proquest.com/pqdtglobal/docview/2673379921/D9677B52EA384A1BPQ/5?accountid=11789&sourcetype=Dissertations%20&%20Theses>
- Banbury, C., Zhou, C., Fedorov, I., Navarro, R. M., Thakker, U., Gope, D., Reddi, V. J., Mattina, M., & Whatmough, P. N. (2020). *MicroNets: Neural Network Architectures for*

Deploying TinyML Applications on Commodity Microcontrollers.

<https://doi.org/10.48550/ARXIV.2010.11267>

Basaklar, T., Tuncel, Y., & Ogras, U. Y. (2022). *tinyMAN: Lightweight Energy Manager using Reinforcement Learning for Energy Harvesting Wearable IoT Devices* (No. arXiv:2202.09297). <http://arxiv.org/abs/2202.09297>

Bhattacharya, S., & Pandey, M. (2024). Deploying an energy efficient, secure & high-speed sidechain-based TinyML model for soil quality monitoring and management in agriculture. *Expert Systems with Applications*, 242, 122735.

<https://doi.org/10.1016/j.eswa.2023.122735>

Bruzzzone, L., Nodehi, S. E., & Fanghella, P. (2022). Tracked Locomotion Systems for Ground Mobile Robots: A Review. *Machines*, 10(8), 648.

<https://doi.org/10.3390/machines10080648>

Cannon, K. M., Dreyer, C. B., Sowers, G. F., Schmit, J., Nguyen, T., Sanny, K., & Schertz, J. (2022). Working with lunar surface materials: Review and analysis of dust mitigation and regolith conveyance technologies. *Acta Astronautica*, 196, 259–274.

<https://doi.org/10.1016/j.actaastro.2022.04.037>

Dawane, M. K., Malwatkar, G. M., Deshmukh, S. P., & Chouhan, D. (2023). Advancements in Control Systems: Sliding Motion Control, Integrating PID Control, and Machine Learning for Precision and Robustness. *2023 IEEE International Conference on ICT in Business Industry & Government (ICTBIG)*, 1–7.

<https://doi.org/10.1109/ICTBIG59752.2023.10455979>

- De Prado, M., Rusci, M., Capotondi, A., Donze, R., Benini, L., & Pazos, N. (2021). Robustifying the Deployment of tinyML Models for Autonomous Mini-Vehicles. *Sensors*, 21(4), 1339. <https://doi.org/10.3390/s21041339>
- Deutel, M., Kontes, G., Mutschler, C., & Teich, J. (2023, May 23). *Combining Multi-Objective Bayesian Optimization with Reinforcement Learning for TinyML*. arXiv.Org. <https://arxiv.org/abs/2305.14109v2>
- Google DeepMind. (n.d.). *MuJoCo—Advanced Physics Simulation*. Retrieved May 9, 2025, from <https://mujoco.org/>
- Horányi, M., Walch, B., Robertson, S., & Alexander, D. (1998). Electrostatic charging properties of Apollo 17 lunar dust. *Journal of Geophysical Research: Planets*, 103(E4), 8575–8580. <https://doi.org/10.1029/98JE00486>
- Israel, D. J., Mauldin, K. D., Roberts, C. J., Mitchell, J. W., Pulkkinen, A. A., Cooper, L. V. D., Johnson, M. A., Christe, S. D., & Gramling, C. J. (2020). LunaNet: A Flexible and Extensible Lunar Exploration Communications and Navigation Infrastructure. *2020 IEEE Aerospace Conference*, 1–14. <https://doi.org/10.1109/AERO47225.2020.9172509>
- Jiang, Z., & Lynch, A. F. (2021). Quadrotor motion control using deep reinforcement learning. *Journal of Unmanned Vehicle Systems*, 9(4), 234–251. <https://doi.org/10.1139/juvs-2021-0010>
- Kwiatkowski, A., Towers, M., Terry, J., Balis, J. U., De Cola, G., Deleu, T., Goulão, M., Kallinteris, A., Krimmel, M., KG, A., Perez-Vicente, R., Pierré, A., Schulhoff, S., Tai, J. J., Tan, H., & Younis, O. G. (2024). *Gymnasium: A Standard Interface for Reinforcement Learning Environments* (No. arXiv:2407.17032). <http://arxiv.org/abs/2407.17032>

- Lee, J., Hwangbo, J., Wellhausen, L., Koltun, V., & Hutter, M. (2020). Learning quadrupedal locomotion over challenging terrain. *Science Robotics*, 5(47), eabc5986.
<https://doi.org/10.1126/scirobotics.abc5986>
- Lele, A. S., Fang, Y., Ting, J., & Raychowdhury, A. (2020). *Learning to Walk: Spike Based Reinforcement Learning for Hexapod Robot Central Pattern Generation*. 208–212.
<https://doi.org/10.1109/AICAS48895.2020.9073987>
- Mbed-torch-fusion-os*. (n.d.). GitHub. Retrieved May 9, 2025, from
<https://github.com/ChristophKarlHeck/mbed-torch-fusion-os/blob/main/CITATION.cff>
- Niksirat, P., Daca, A., & Skonieczny, K. (2020). The effects of reduced-gravity on planetary rover mobility. *The International Journal of Robotics Research*, 39(7), 797–811.
<https://doi.org/10.1177/0278364920913945>
- Ouyang, W., Chi, H., Pang, J., Liang, W., & Ren, Q. (2021). Adaptive Locomotion Control of a Hexapod Robot via Bio-Inspired Learning. *Frontiers in Neurorobotics*, 15, 627157.
<https://doi.org/10.3389/fnbot.2021.627157>
- PyTorch Foundation. (n.d.). *Welcome to the ExecuTorch Documentation—ExecuTorch 0.6 documentation*. ExecuTorch Documentation. Retrieved May 9, 2025, from
<https://docs.pytorch.org/executorch/stable/index.html>
- Qiu, Z., Wei, W., & Liu, X. (2023). Adaptive Gait Generation for Hexapod Robots Based on Reinforcement Learning and Hierarchical Framework. *Actuators*, 12(2), 75.
<https://doi.org/10.3390/act12020075>
- Raffin, A., Hill, A., Gleave, A., Kanervisto, A., Ernestus, M., & Dormann, N. (2021). Stable-Baselines3: Reliable Reinforcement Learning Implementations. *Journal of Machine Learning Research*, 22(268), 1–8.

- Ren, H., Anicic, D., & Runkler, T. A. (2021). *TinyOL: TinyML with Online-Learning on Microcontrollers*. 1–8. <https://doi.org/10.1109/IJCNN52387.2021.9533927>
- Rimani, J., Bucchioni, G., Ryals, A. D., Viola, N., & Lizy-Destrez, S. (2023). Integrated Conceptual Design and Parametric Control Assessment for a Hybrid Mobility Lunar Hopper. *Aerospace*, 10(8), 669. <https://doi.org/10.3390/aerospace10080669>
- Rodríguez-Martínez, D., Van Winnendael, M., & Yoshida, K. (2019). High-speed mobility on planetary surfaces: A technical review. *Journal of Field Robotics*, 36(8), 1436–1455. <https://doi.org/10.1002/rob.21912>
- Rosenburg, M. A., Aharonson, O., Head, J. W., Kreslavsky, M. A., Mazarico, E., Neumann, G. A., Smith, D. E., Torrence, M. H., & Zuber, M. T. (2011). Global surface slopes and roughness of the Moon from the Lunar Orbiter Laser Altimeter. *Journal of Geophysical Research*, 116(E2), E02001. <https://doi.org/10.1029/2010JE003716>
- Sanchez-Iborra, R., & Skarmeta, A. F. (2020). TinyML-Enabled Frugal Smart Objects: Challenges and Opportunities. *IEEE Circuits and Systems Magazine*, 20(3), 4–18. <https://doi.org/10.1109/MCAS.2020.3005467>
- Schilling, M., Konen, K., Ohl, F. W., & Korthals, T. (2020). *Decentralized Deep Reinforcement Learning for a Distributed and Adaptive Locomotion Controller of a Hexapod Robot*. 5335–5342. <https://doi.org/10.1109/IROS45743.2020.9341754>
- Schulman, J., Moritz, P., Levine, S., Jordan, M., & Abbeel, P. (2018). *High-Dimensional Continuous Control Using Generalized Advantage Estimation* (No. arXiv:1506.02438). <http://arxiv.org/abs/1506.02438>
- Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). *Proximal Policy Optimization Algorithms* (No. arXiv:1707.06347). <http://arxiv.org/abs/1707.06347>

- Seo, M., Gupta, R., Zhu, Y., Skoutnev, A., Sentis, L., & Zhu, Y. (2023). Learning to Walk by Steering: Perceptive Quadrupedal Locomotion in Dynamic Environments. *2023 IEEE International Conference on Robotics and Automation (ICRA)*, 5099–5105.
<https://doi.org/10.1109/ICRA48891.2023.10161302>
- Serria, E., Gadhafi, R., AlMaeni, S., Mukhtar, H., Copiaco, A., Abd-Alhameed, R., Lemieux, F., & Mansoor, W. (2023). A Review of Lunar Communications and Antennas: Assessing Performance in the Context of Propagation and Radiation. *Sensors*, 23(24), 9832. <https://doi.org/10.3390/s23249832>
- Stubbs, T. J., Vondrak, R. R., & Farrell, W. M. (2007). *Impact of Dust on Lunar Exploration*. 643, 239–243.
- Sutoh, M. (2024). Development and Evaluation of Mobility and Excavation Rover Toward Lunar Base Construction. *Journal of Robotics and Mechatronics*, 36(2), 334–342.
<https://doi.org/10.20965/jrm.2024.p0334>
- Tsounis, V., Alge, M., Lee, J., Farshidian, F., & Hutter, M. (2020). DeepGait: Planning and Control of Quadrupedal Gaits Using Deep Reinforcement Learning. *IEEE Robotics and Automation Letters*, 5(2), 3699–3706. <https://doi.org/10.1109/LRA.2020.2979660>
- Vinella, A. E. (2024). *Learning Bipedal Locomotion Using Central Pattern Generators and Deep Reinforcement Learning* [M.S., University of California, Los Angeles].
<https://www.proquest.com/pqdtglobal/docview/3067125924/abstract/128D98BDFEB43F8PQ/4>
- Wan, K., Gao, X., Hu, Z., & Wu, G. (2020). Robust Motion Control for UAV in Dynamic Uncertain Environments Using Deep Reinforcement Learning. *Remote Sensing*, 12(4), 640. <https://doi.org/10.3390/rs12040640>

Appendix A - Research Algorithms

This appendix describes the different algorithms used in the implementation of the experiment. These are provided to allow others to replicate to the experiment.

Algorithm 1: SCRIPT FOR TRAINING AGENT

Input: Training parameters (including Gym environment)

Output: Motion controller model (PyTorch model)

- 1 *Initialize environment using Gymnasium API*
 - 2 *Create PPO model using Stable Baselines3*
 - 3 *Train model using PPO algorithm from Stable Baselines3*
 - 4 *Save model as PyTorch model*
-

Algorithm 2: RUN INFERENCE ON MICROCONTROLLER

Input: Environment observations sent via serial

Output: Joint actions in the form of applied torque sent via serial

- 1 *Initialize model*
 - 2 ***LOOP WHILE true DO*** //Run continuously while MCU is powered
 - 3 ***IF*** serial data available ***THEN***
 - 4 *Process serial input*
 - 5 *Pass observation data into model*
 - 6 *Run model inference*
 - 7 *Send action data (joint torques) via serial*
 - 8 ***END IF***
 - 9 ***END LOOP***
-

Algorithm 3: SCRIPT FOR EVALUATING AGENT – MCU CONFIGURATION

Input: Evaluation parameters (including Gym environment)

Output: Performance data logs

```
1  start_idx ← starting index of episodes
2  stop_idx ← last index of episodes
3  Set current_episode to start_idx
4  Initialize environment
5  LOOP WHILE (current_episode <= stop_idx ) DO
6      Reset end_episode flag to False
7      Reset environment
8      Log initial observations from environment
9      Send initial observations to microcontroller via serial
10     LOOP WHILE (NOT end_episode) DO //run until episode termination condition
11         IF serial data available THEN //Wait for response from microcontroller
12             Process serial input
13             Step environment using received action data
14             Log new observations from environment
15             IF unhealthy THEN //stop episode
16                 Increment current_episode by 1
17                 BREAK
18             END IF
19             Send observation data to microcontroller via serial
20         END IF
21     END LOOP
22 END LOOP
```

Algorithm 4: SCRIPT FOR EVALUATING AGENT – PYTORCH CONFIGURATION

Input: Evaluation parameters (including Gym environment)

Output: Performance data logs

```
1  start_idx ← starting index of episodes
2  stop_idx ← last index of episodes
3  Set current_episode to start_idx
4  Load PyTorch model
5  Initialize environment
6  LOOP WHILE (current_episode ≤ stop_idx ) DO
7      Reset end_episode flag to False
8      Reset environment
9      Log initial observations from environment
10     Send initial observations to microcontroller via serial
11     LOOP WHILE (NOT end_episode) DO //run until episode termination condition
12         Run model inference with current observation
13         Step environment using received action data
14         Log new observations from environment
15         IF unhealthy THEN //stop episode
16             Increment current_episode by 1
17             BREAK
18         END IF
19     END LOOP
20 END LOOP
```

Appendix B - System-Level Integration

As defined in the problem statement, the lunar surface is the targeted environment for the autonomous platform to operate within. Since the training of the motion controller is performed in a virtual representation of this environment, understanding how the virtual simulation parameters are modeled within this environment is key to translating to a real environment. The Gymnasium Ant environment uses the MuJoCo physics engine and simulator. Different parameters of the simulation can be configured to represent the appropriate environmental parameters, see Table B.1.

For the Ant-v4 configuration, the values for the parameters are shown in Table B.1. These values represent the state variables of the environment that the platform is interacting with. They were not reconfigured for the experiment to allow data to be collected using the environment as defined in the software library. This is noted as one of the research limitations as they represent Earth's values instead of the lunar environment, and it is recommended to reconfigure these to represent the lunar environment. The two primary parameters to reconfigure would be gravity and global magnetic flux, as these are based on Earth's values. The parameters wind, density and viscosity represent the medium in which the platform is moving within. Since the lunar surface does not contain an atmosphere (no medium), the zero values here are pertinent to the lunar surface. Finally, the timestep is set to 10 milliseconds. This parameter may need to be reconfigured when evaluating performance for real-time operation. Note, this will be limited by the host machine's central processing unit (CPU) performance. Refer to <https://mujoco.readthedocs.io/en/stable/XMLreference.html#option-timestep> for more details on timestep configuration.

In addition, platform design is important to development of the motion controller as well. The characteristics of the platform per the Ant-v4 environment is shown in Table B.2. While the height may not be as critical in the experiment, environments that feature the platform to navigate enclosed spaces or areas with obstacles that may overhang would need to consider the height. For this study, this provides a sense of how large the platform is, which is useful in volume estimates for payload requirements aboard launch and transport vehicles. Next, weight is provided as well. Once again, this plays a factor in deployment and operation cost. Both

characteristics are defined by the virtual model of the parameter and assumed to be constant during the simulation.

Table B.1. Gymnasium Ant-v4 Environment Parameters

Parameter	Functional Type	Data Type	Value
timestep	Time	Real	0.01 seconds
gravity	Acceleration vector	Real(3)	[0 0 -9.81] m/s ²
wind	Velocity vector	Real(3)	[0 0 0] m/s
magnetic	Global magnetic flux	Real(3)	[0 -0.5 0]
density	Medium density	real	0 kg/m ³
viscosity	Medium viscosity	real	0 m/s ²

Table B.2: Gymnasium Ant-v4 Platform Characteristics

Characteristic	Value
Height	75 cm
Weight	910.88 g