Spring 5-18-2020

VIRTUAL ROBOT LOCOMOTION ON VARIABLE TERRAIN WITH ADVERSARIAL REINFORCEMENT LEARNING

Phong Nguyen

VIRTUAL ROBOT LOCOMOTION ON VARIABLE TERRAIN WITH ADVERSARIAL REINFORCEMENT LEARNING

A Thesis

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Phong Nguyen

May 2020

The Designated Committee Approves the Master's Project Titled

VIRTUAL ROBOT LOCOMOTION ON VARIABLE TERRAIN WITH ADVERSARIAL REINFORCEMENT LEARNING

by

Phong Nguyen

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY

May 2020

ABSTRACT

VIRTUAL ROBOT LOCOMOTION ON VARIABLE TERRAIN WITH ADVERSARIAL REINFORCEMENT LEARNING

by Phong Nguyen

Reinforcement Learning (RL) is a machine learning technique where an agent learns to perform a complex action by going through a repeated process of trial and error to maximize a well-defined reward function. This form of learning has found applications in robot locomotion where it has been used to teach robots to traverse complex terrain. While RL algorithms may work well in training robot locomotion, they tend to not generalize well when the agent is brought into an environment that it has never encountered before. Possible solutions from the literature include training a destabilizing adversary alongside the locomotive learning agent. The destabilizing adversary aims to destabilize the agent by applying external forces to it, which may help the locomotive agent learn to deal with unexpected scenarios. For this project, we will train a robust, simulated quadruped robot to traverse a variable terrain. We compare and analyze Proximal Policy Optimization (PPO) with and without the use of an adversarial agent, and determine which use of PPO produces the best results.

ACKNOWLEDGEMENTS

I would like to thank my advisor Prof. Robert Chun for his patience and guidance throughout the duration of the project.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 INTRODUCTION

With the growing popularity of machine learning (ML) there has been increasingly more interest in applying ML techniques to fields where an agent may encounter situations that do not exactly match up with what it has learned, such as with creating robots with improved robustness to various environments and real-world scenarios [1], [2], [3]. Robots are relevant today and in the future because of their applications in environments where using humans may not be safe, hospitable or efficient, such as with bomb detection and deactivation, navigation through radiation-littered environments, and repetitive manufacturing tasks [4], [5], [6], [7], [8]. Creating an efficient robot requires the robot to be sufficiently robust, so that it can effectively perform its desired actions in any situation. Reinforcement learning (RL) with the use of an adversarial agent is one such proposed solution to providing robustness to robots without having to train the robot in every perceivable environment [9].

RL has been applied to robot motion. In a closed testing environment, a physical robot may only be exposed to a specific set of training environments, where it may eventually learn to perform well in such environments. However, when exposed to the real world, such robots may fail to perform when they encounter unexpected situations that they had never learned to deal with during the training phase [9]. In this paper, we explore the usage of RL techniques to train a virtual locomotive agent with the robustness to traverse variable terrain and situations that it has never encountered before.

2 BACKGROUND

2.1 Robot Locomotion on Variable Terrain

The focus of this project is on legged robot locomotion on variable terrain. Legged locomotion is generally considered more versatile than wheeled locomotion because of its resistance to getting stuck on malleable terrain. Whereas a wheeled robot or vehicle would be prone to getting stuck in mud, snow, and sand, a legged robot would be able to walk through such environments [10]. Likewise, legs also allow for traversal through uneven and debris-filled terrain, which would be a challenge for a wheeled vehicle or robot [11]. These advantages make legged robots suited for applications such as search and rescue missions [11] and carrying cargo through battlefields and rough terrain [12].

2.2 Reinforcement Learning

Reinforcement Learning (RL) is a type of ML where the agent learns to perform a complex action, such as walking, by learning to perform specific smaller actions that maximize an accumulated reward value and eventually lead up to the complex action being performed. Much like training a dog, RL is a method of trial and error where the agent is given more rewards for performing actions that are explicitly defined by the programmer to be correlated with leading to the target action. For example, if our target action is to walk from Point A to Point B, then a possible condition that would yield high reward would be for the agent to perform some action that puts it closer to Point B. Based on its current state and past experiences, the agent would perform a specific action, such as rotating a limb a specific amount, that it thinks will maximize its reward. The result of this reward maximization would potentially result in the agent performing the desired action when played out in successive steps. Each reward signal is defined by the programmer, and how the rewards are assigned may greatly influence how well the agent learns. However, designing a reward that corresponds with a desired action can be tricky. Heess et al. [13] suggested that it is not always obvious what a “right” reward function should be, and that picking a “wrong” reward function could lead to unexpected and undesired results [13]. As an answer to this, Heess et al. [13] proposed that rather than train the agent with a complex reward function, the agent would learn complex actions if it were exposed to complex training conditions with a simple reward function.

2.3 Reinforcement Learning in Rich Environments

In their paper, Heess et al. [13] found that they could create a robust agent by training it in rich and varied environments while keeping its reward function simple. Their motive for a simple reward function comes from the tendency of well-defined rewards to overfit when trained in environments with minor variances, which results in the trained agent being over-specialized to its training environment and ill-suited for traversing varied environments that it has never encountered before. Additionally, the difficulty of crafting a reward function to encompass every use case that the agent may encounter during the testing phase makes it more desirable for the programmer to keep the reward function simple. In this case, Heess et al.'s [13] agent received rewards entirely based on its progress in moving towards the forward direction, and nothing else. To create a robust locomotive agent, Heess et al. [13] traded off a complex reward function for a complex training environment. Their training environment was a procedural obstacle course with varying degrees of difficulty, making use of transfer learning to allow the agent to gradually train in more difficult environments as it became proficient with its current training grounds. Transfer learning is a ML technique that allows knowledge gained from one training environment to be “transferred” and applied to another similar training environment to improve training efficiency [14]. The procedural training environment included various adjustable parameters such as floor steepness, unevenness, and gap distance. Heess et al. [13] concluded their research with the following results: "simple rewards lead to effective motion; transfer training leads to faster results; diverse training environments leads to robustness" [13].

2.4 Reinforcement Learning with an Adversary

Training an agent as a physical robot adds extra challenges that must be addressed during the training process because the environmental awareness of the learning agent is restricted by the limitations of technology. In other words, while we may be able to freely feed specific environmental parameters into a virtual agent as part of its observations, doing the same with a physical robot is constrained by the available physical components that the researcher has at hand. Sometimes those components may not be sufficient for accomplishing what the researcher had desired. Pinto et al. [15] encountered this limitation while training a physical robot to grasp objects, where they found that the force sensor in their robot's hand was unable to accurately determine what a good grip "felt" like since it was unable to take the object's orientation and other parameters into consideration. They hypothesized that the robot would learn to grip objects more effectively if it were to be discouraged from having a bad grip. Their proposed solution was to train an adversarial robot alongside the grasping robot that would learn to snatch the object from the grasper's hand. The idea was that the presence of an adversary would lead to an arms race where the grasping robot would learn firmer grasping technique as the adversarial snatching robot learned to exploit weaknesses in the grasper's grip on the object. The results supported their hypothesis and showed that an adversarial approach required less data and was more robust than the control case trained without an adversary [15].

2.5 Robust Adversarial Reinforcement Learning

While Pinto et al.'s [15] research was focused on compensating for the restrictions of physical hardware, Pinto et al. [9] found that normal RL did not generalize well with varying environments even in a virtual setting. Like with Pinto et al.'s [15] research, their proposed solution was Robust Adversarial Reinforcement Learning (RARL), where an adversary is trained alongside the learning agent and attempts to disrupt its learning progress. With their RARL experiment, Pinto et al. [9] set out to train a robust, virtual locomotive agent that is capable of traversing varied terrain that it has never encountered before. Unlike with Pinto et al.'s [15] research, the RARL experiment was conducted on virtual agents, so there were opportunities to give the adversary capabilities that a physical entity would not be capable of. Rather than simply make the adversary physically interrupt the learning agent, Pinto et al. [9] gave the adversary "super powers" and made it capable of adjusting physics values and applying external forces directly to the learning agent, to create an adversary that is capable of exploiting weaknesses based on domain knowledge. Making use of the adversary and various environment settings such as differing friction and mass values, they found that the agent trained with RARL was more effective at generalizing its locomotive capabilities to more varied terrain, compared to the baseline trained with RL without an adversary. Additionally, the RARL adversary that makes use of domain knowledge was more effective at putting the learning agent in situations that resulted in lesser rewards, which resulted in the agent discovering more robust locomotive actions [9].

2.6 Reinforcement Learning for Slope Climbing

Garg [16] made use of RL to train a virtual agent to climb various slopes. Garg [16] performed experiments using Trust Region Policy Optimization (TRPO) [17] and Deep Deterministic Policy Gradients (DDPG) [18] and compared which of the two would be more effective for training a slope-climbing agent. Garg [16] put a crawler agent through numerous tests to determine which algorithm was more effective, starting off with a flat terrain, attempting flat terrain with steps, and then gradually moving on to sloped surfaces. In their initial tests with flat planes and steps, Garg [16] found that TRPO was roughly three times faster at yielding viable locomotive agents than DDPG, while also producing agents that were found to be more stable, flipping themselves over at a less frequent rate than those trained with DDPG. Because of the combination of speed and stability provided by TRPO, Garg [16] decided to perform the final sloped tests entirely with TRPO. Using TRPO as the RL algorithm, Garg's [16] agent learned to climb slopes of 9 degrees, but struggled to climb steeper slopes due to the friction of its feet being insufficient to stick to the slope surface. Garg's [16] solution was to train the agent on grooved terrain, which allowed the agent to hook its feet onto the grooves in the terrain and climb slopes of up to 36 degrees. As an additional test, each slope was tested with a trial where only horizontal velocity was considered for its reward function, and a second trial where both horizontal and vertical velocity were considered. Generally, the tests with both horizontal and vertical velocity considered for the reward function were more successful and allowed the agent to climb steeper slopes. This is an example of how designing a well-defined reward function is important for the success of the agent's learning. After successfully training the agent to climb 36 degree slopes, Garg [16] decided to utilize transfer learning to train their agent on more complex tasks, such as climbing a slope with potholes. Unfortunately, even with the use of transfer learning, Garg's [16] agent was unable to climb these slopes any steeper than 18 degrees. The agent was trained with the same reward function as the previous slope-climbing agent, and was unable to adapt to the new terrain.

3 IMPLEMENTATION

While bipedal humanoid locomotion is one of the ultimate goals of locomotive ML, we chose to reduce the scope of our locomotive ML learning model so that it was possible to complete with our limited hardware. A quadruped crawler was chosen over a humanoid for this experiment because it is a simpler model, with fewer degrees of freedom, making it a more viable approach with our limited hardware [13]. Unlike a humanoid that stands upright on two legs with its center of mass placed far above its legs, a quadruped crawler stands on all four legs with its center of mass placed closer to the ground and evenly distributed between all four legs. Like a four-legged table, this standing configuration gives the crawler a centralized weight distribution and the advantage of being better balanced, so the agent does not need to solve the inverted pendulum problem with the complexity that a biped humanoid would need to for it to learn locomotion [19], [20].

The inverted pendulum problem is a balance problem where the goal is to maintain the balance of a mass atop a balancing pivot, illustrated in Fig. 1. A normal pendulum has its center of mass hanging below its pivot point, which puts its resting equilibrium position directly below the pivot point. An inverted pendulum on the other hand, has its center of mass above its pivot point, so if left unattended, the mass will swing along the balancing pivot point until it runs out of energy and rests at its equilibrium position below the pivot. A common representation of the inverted pendulum problem is balancing a glass of liquid on top of a stick. In order to maintain balance and prevent the glass from falling off, the stick is allowed to move horizontally or freely pivot along its point of contact between itself and the floor. The glass, in turn, passively reacts to the stick's movements, moving along with it and pivoting due to the effects of the stick's motions and gravity.

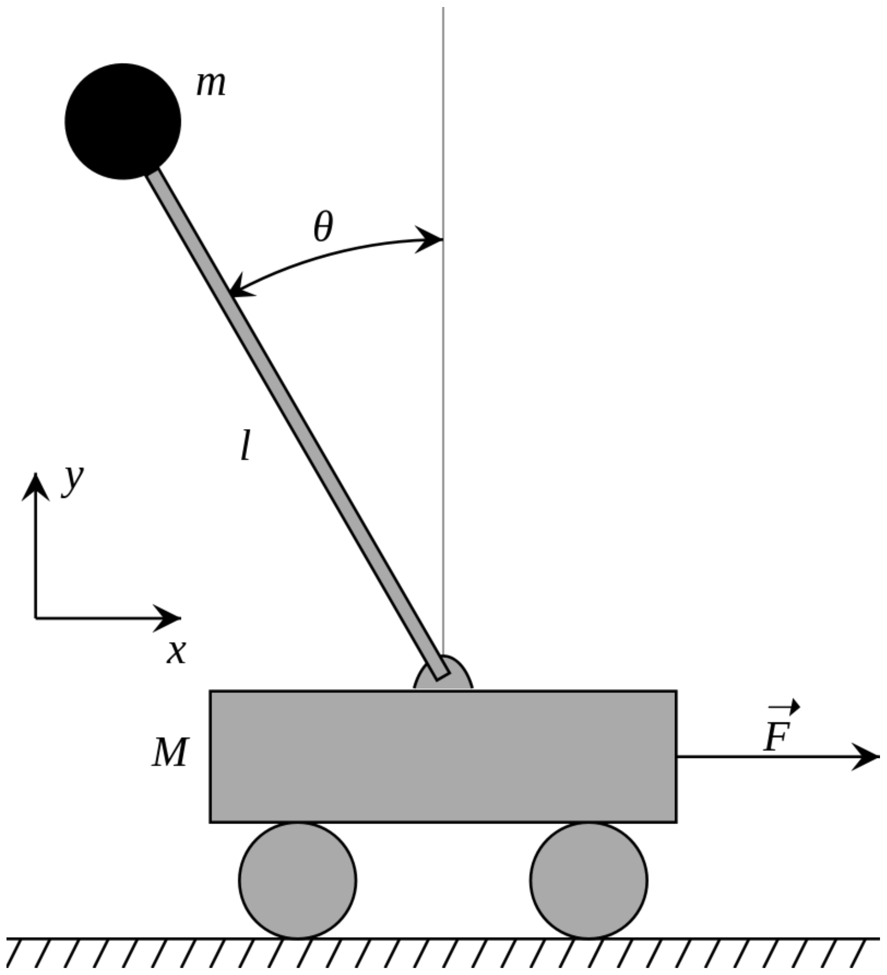


Fig. 1. Example of inverted pendulum on a moveable cart, with point mass m and pivot attached to top of cart [21].

The image of a mass balanced atop a stick can be applied to a humanoid when seen from the side, where the mass is the humanoid's torso, and the stick is the humanoid's legs. Whenever the humanoid stands with both legs side-by-side, or with one leg lifted, the humanoid must solve the inverted pendulum problem in order to maintain balance [22]. It can easily be seen that biped forward locomotion is the act of the humanoid lifting one leg up and forward, intentionally falling forward, and then placing the lifted leg back down to regain balance, thus taking a step forward in the resulting process.

4 EXPERIMENT AND RESULTS

For our experiment, we utilized the Unity game engine along with the Unity Machine Learning Agents Toolkit (ML-Agents) to train a virtual agent to traverse variable terrain. We used Proximal Policy Optimization (PPO) [23] as the ML algorithm for both the agent and the adversary because it was included as part of the ML-Agents package and was ready to use. The Unity game engine provided us with basic 3D game development tools such as a built-in 3D physics simulation engine and the ease of implementing additional features with C# code. Unity allows the user to create any kind of controllable agent that is capable of interacting with its environment through physics contact and collisions, which made it ideal for creating and training a locomotive agent controlled by our machine learning model. The ML-Agents package is a machine learning framework added on top of Unity that enables the user to train and test agents using the included PPO algorithm or with their own algorithm. The Unity ML-agents package made it simple to get started with a physics-based ML training environment, and even included some basic sample projects to expand upon.

4.1 Project Settings

We made use of the ML-Agents Toolkit's Crawler project as the basis for our experiment. The Crawler project included everything required to train a quadruped crawler agent, such as the premade crawler agent object, as well as a scene for training the agent in. By default, the Crawler project was a barebones project that only included a setup for training the crawler on a flat plane, but we had made enhancements to the project to allow for training and testing on more complex terrain. In the following subsections, we will go over each of the components of the project, as well as the settings used in each test.

4.2 Terminology and Components

4.2.1 Learning Environment

At a high level, the ML-agents package is made up of three main components: the Learning Environment, the Python API, and the External Communicator, as seen in Fig. 2. The Learning Environment includes a Unity scene and all agents and interactive objects that exist within that scene. In Unity, a scene is a file that contains the environment and all relevant objects and scripts that are needed to view or play the scene. In the case of ML-agents, the Learning Environment is a scene that includes the terrain, agents, and relevant scripts that are required for making the agents run and for training to take place. The Python API is an external toolset that includes all the machine learning algorithm functionality used for training agents. The Python API interacts with the Learning Environment through the use of the External Communicator, which exists in and is a component of the Learning Environment.

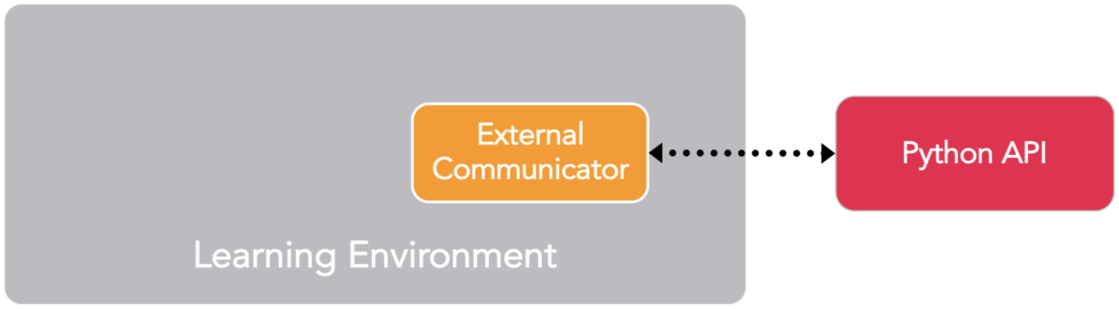


Fig. 2. The high-level components of ML-agents [24].

4.2.2 Academy

The entity that we train is called the agent. The agent is capable of interacting with its environment or other agents, and learns by accumulating knowledge in its brain. By utilizing Unity's included game development tools, the user can build an agent from the ground up, and customize it to fit the needs of the training experiment. The agent performs actions by first taking observations of its surroundings and environment, then it makes decisions on what actions to perform. The agent then continuously performs that action each time interval until its next decision step is reached where it chooses a different action to perform. The agent’s decision making is done by its brain.

A brain in the context of Unity ML agents is a component that stores knowledge and issues actions based on current physical observations and its accumulated knowledge [24]. In ML agents, we train brains using the agent as the brain's means for interacting with its environment. For example, a crawler agent has a brain that can be swapped out depending on which environment and settings we want to train it with. After training our brains, we can swap out the brain assigned to the crawler depending on which trained brain we want to observe in a given testing environment. To improve the efficiency of the training process on a single machine, we can train multiple brains concurrently through the use of an academy, illustrated with its related components in Fig. 3.

An academy is the central processing center that orchestrates observations and actions among the brains being trained, keeping agents synchronized as it issues them commands. Agents with the same brain linked to an academy share the same knowledgebase, so separate instances of an agent will make decisions based on the same shared knowledge, and can be considered the same agent [25].

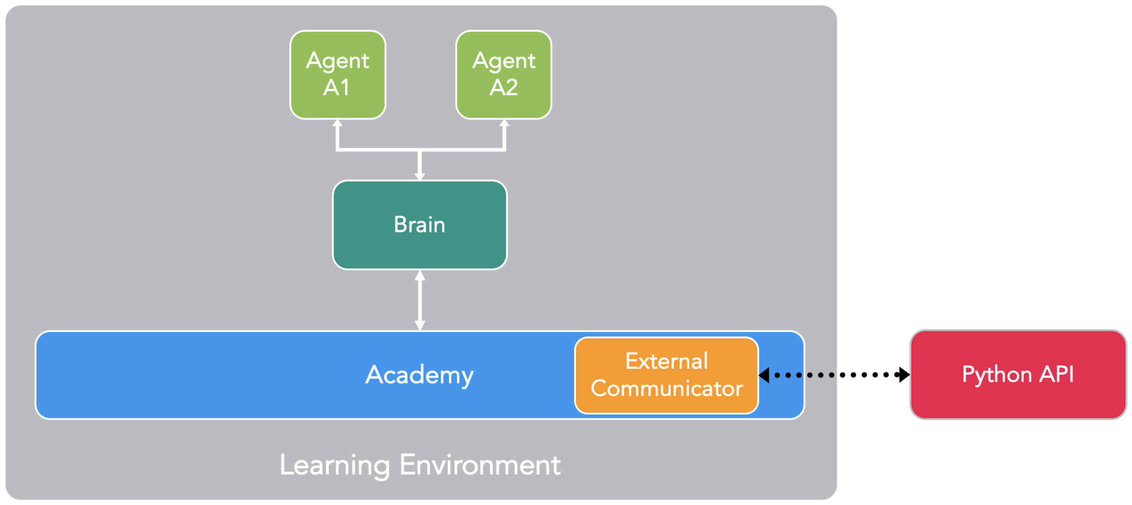


Fig. 3. The main components in the Learning Environment that a researcher will have to configure when utilizing ML-agents [24].

4.2.3 Terminology

In general, reinforcement learning is centered around tailoring a reward signal in order to steer the learning agent towards learning the intended motions or actions. The reward is an accumulated and stored value that is used to quantify if the agent is on the right track or not, and it is the researcher's job to tailor the reward according to the desired goal. In this case, our goal was to create a robust locomotive agent that is capable of traversing variable terrain. However, because such a complex goal is not specific enough, we broke down the goal into smaller components in order to determine how to assign the reward. We decided to have the agent first learn to walk by having it move towards a target point located at the end of a long running plane. Our criteria for properly walking towards the goal included facing the target position and moving towards the target, with a time penalty deducted from the reward as time progressed.

The agent attempts to increase its accumulated reward by performing actions issued by the brain. In the context of ML-Agents, an action is a specific act that the agent can physically perform, such as rotating a limb by a specific amount, along a specific axis. The agent decides what action to perform based on its perceived state of itself and the world, called observations.

The agent uses observations to determine what its current state is in the world so that it can determine how to react based on its accumulated knowledge. Observations are data and values of the agent and interactive objects within the environment that are collected by the agent during each step of the physics update.

A step is a fixed time interval that the physics engine is updated at, which can be configured by the user of the ML-Agents package. An episode is the duration of one lap through the training course, and can also be terminated after a preset number of steps has passed. An episode may also be terminated prematurely before reaching the step limit if the agent accomplishes an end condition such as successfully reaching the end target or by toppling over and failing.

4.2.4 Quadruped Crawler Agent

The crawler is a non-humanoid quadruped capable of rotating each leg at two points of articulation with two axes of rotation on its upper leg and one axis of rotation on its foreleg, as seen in the following figures.

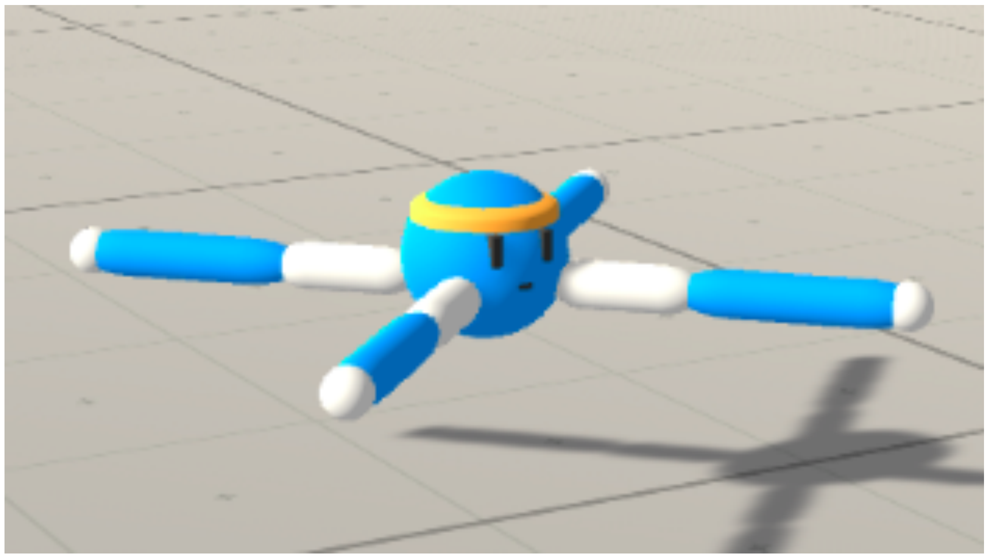


Fig. 4. Image of crawler in its default pose.

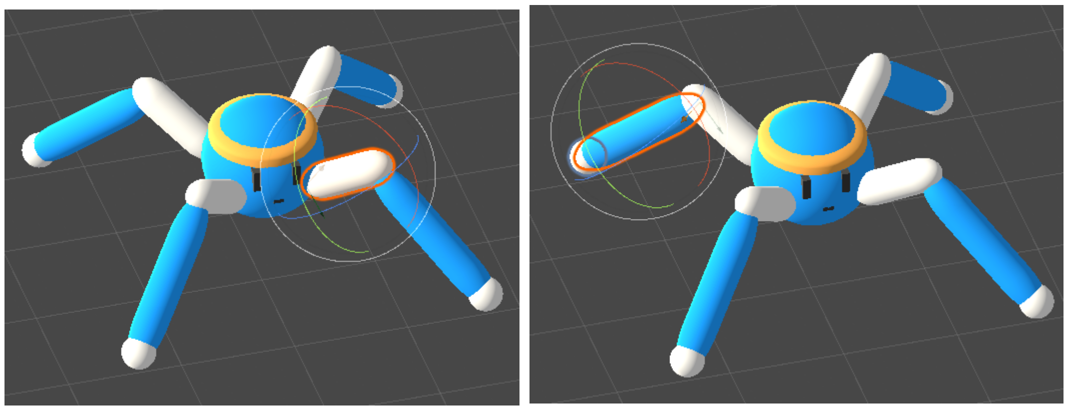
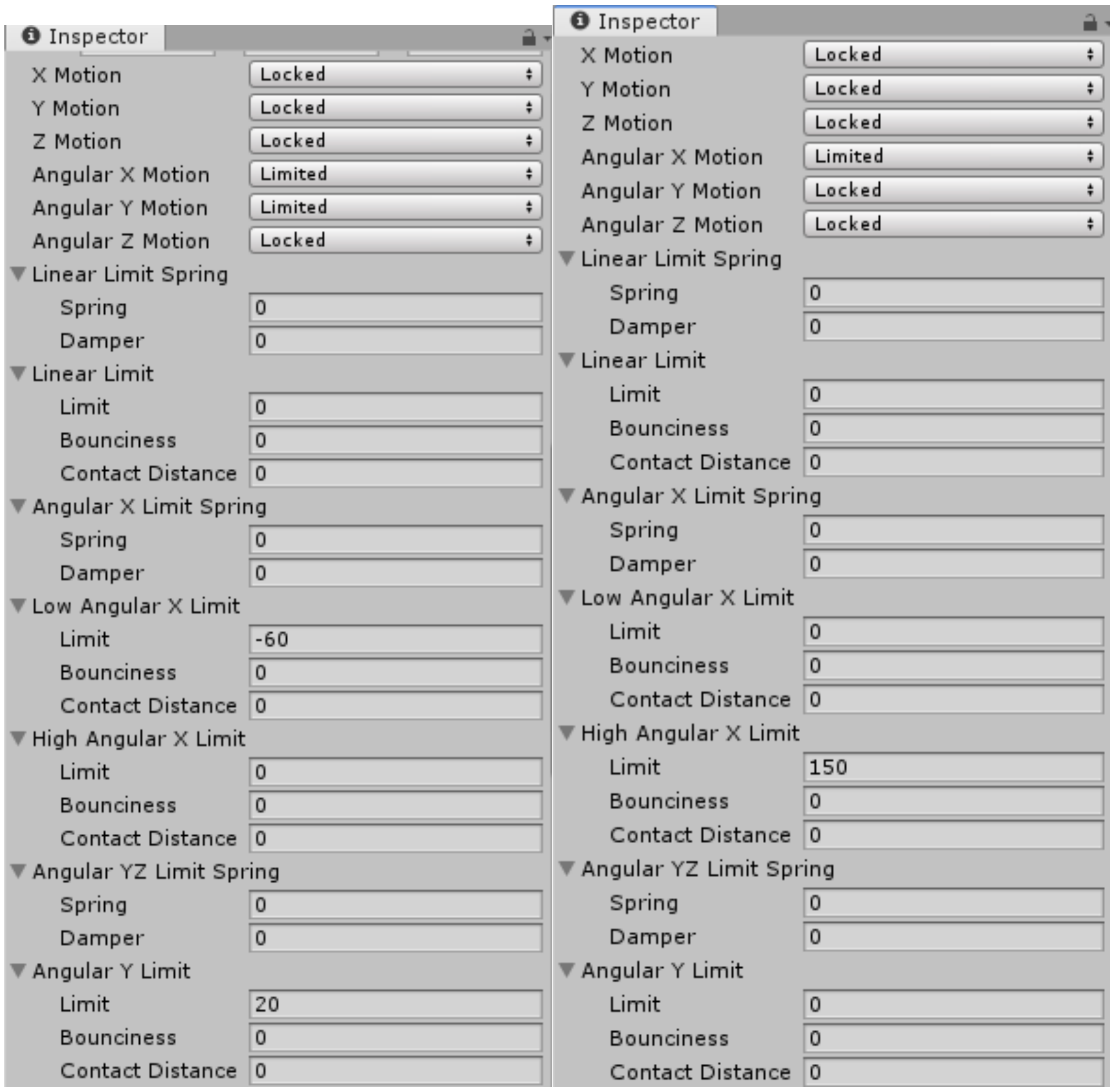


Fig. 5. Agent and its range of motion on upper leg (left image) and foreleg (right image).



Inspector

X Motion	Locked
Y Motion	Locked
Z Motion	Locked
Angular X Motion	Limited
Angular Y Motion	Limited
Angular Z Motion	Locked
▼ Linear Limit Spring	
Spring	0
Damper	0
▼ Linear Limit	
Limit	0
Bounciness	0
Contact Distance	0
▼ Angular X Limit Spring	
Spring	0
Damper	0
▼ Low Angular X Limit	
Limit	-60
Bounciness	0
Contact Distance	0
▼ High Angular X Limit	
Limit	0
Bounciness	0
Contact Distance	0
▼ Angular YZ Limit Spring	
Spring	0
Damper	0
▼ Angular Y Limit	
Limit	20
Bounciness	0
Contact Distance	0

Inspector

X Motion	Locked
Y Motion	Locked
Z Motion	Locked
Angular X Motion	Limited
Angular Y Motion	Locked
Angular Z Motion	Locked
▼ Linear Limit Spring	
Spring	0
Damper	0
▼ Linear Limit	
Limit	0
Bounciness	0
Contact Distance	0
▼ Angular X Limit Spring	
Spring	0
Damper	0
▼ Low Angular X Limit	
Limit	0
Bounciness	0
Contact Distance	0
▼ High Angular X Limit	
Limit	150
Bounciness	0
Contact Distance	0
▼ Angular YZ Limit Spring	
Spring	0
Damper	0
▼ Angular Y Limit	
Limit	0
Bounciness	0
Contact Distance	0

Fig. 6. Leg angle limits in degrees are shown for upper leg (left image) and foreleg (right image), with Low Angular Limit representing the minimum angle a limb is capable of rotation, and High Angular Limit representing the maximum angle. The angles are shown in degrees relative to the legs' default position.

Other observations given to the agent were designed with the idea of giving it some spatial awareness and knowledge of its limbs' orientations. We attempted to improve the crawler's balance capabilities by giving it sensors at the ends of its feet that return the surface normal, simulated in Fig. 7, of the point of contact that each foot makes, somewhat simulating the effect of having feet that pivot at its ankles when pressed along a surface. In addition to having a sense of touch with the use of surface normals, we gave the agent awareness of its local up vector so that it can have a sense of orientation in the world, somewhat simulating the balancing effects of an inner ear. We also calculated the agent's center of mass as an observation in an attempt to improve the agent's balance. The surface normal of each foot contact, along with the agent's current local up vector and center of mass are all passed to the brain as observations. To determine how high a foot was off the ground, we made each of the agent's feet calculate the distance from its center to the ground. The agent's altitude relative to the origin, and the agent's leg rotation and position values were also passed as observations. In order to give the agent awareness of the target point that it should be moving towards, we calculated the direction to the target and passed that vector as an observation.

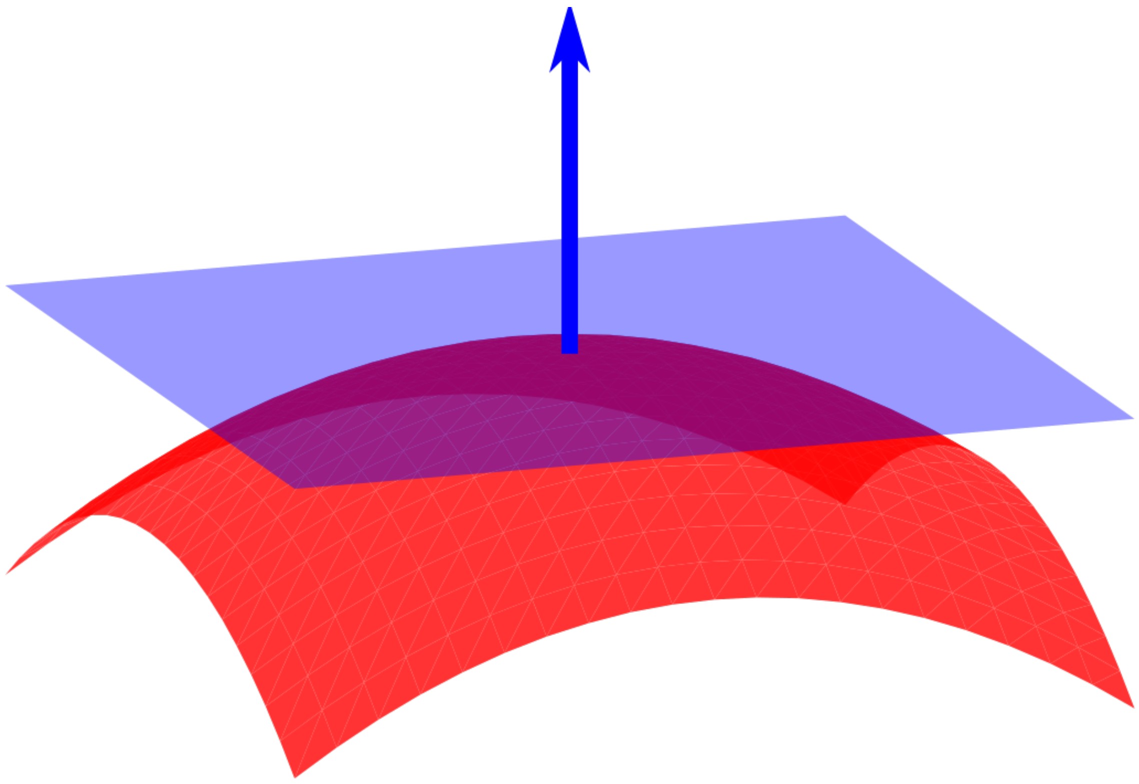


Fig. 7. Example of a surface normal (blue arrow) of a curved terrain [26].

The agent's reward function was based on rewarding the agent whenever it performed an action that would lead to progressing towards the target. After the agent performed an action, it gained reward whenever it faced the target, moved closer to the target, or touched the target, and lost a small amount of reward as time progressed. Meanwhile, whenever the agent gained reward, the adversary gained the same amount of reward as the agent, but multiplied by -1.

4.2.5 Adversary Agent

Pinto et al. [15] and Pinto et al. [9] have shown that implementing an adversary is a viable method for training a robust locomotive agent, so we decided to also use an adversary for our locomotive experiment. Pinto et al.'s [9] RARL approach consisted of training the locomotive learning agent alongside an adversary agent that learned to apply optimal forces/disturbances to destabilize the locomotive agent as it was learning.

The adversary is an agent that learns how to optimally interfere with the crawler's learning process in an attempt to improve the robustness of the crawler's learned behavior. With an adversarial approach to RL, the adversary learns to exploit the agent's weaknesses as the agent learns to overcome challenges of its task along with the additional challenges created by the adversary. It is an arms race that ultimately leads to an optimized and robust agent and adversary, where the trained agent can skillfully react to nearly any challenge it faces thanks to the hardships it faced while learning with interference from the adversary, while a well-trained adversary is cunning enough to be able to thwart the learning attempts of a less clever agent. The adversary in our setup does not physically exist, and instead interacts with the crawler agent by directly applying forces to, and by modifying the mass and friction values of one of the crawler's limbs or body. Fig. 8 illustrates a physical representation of the adversary.

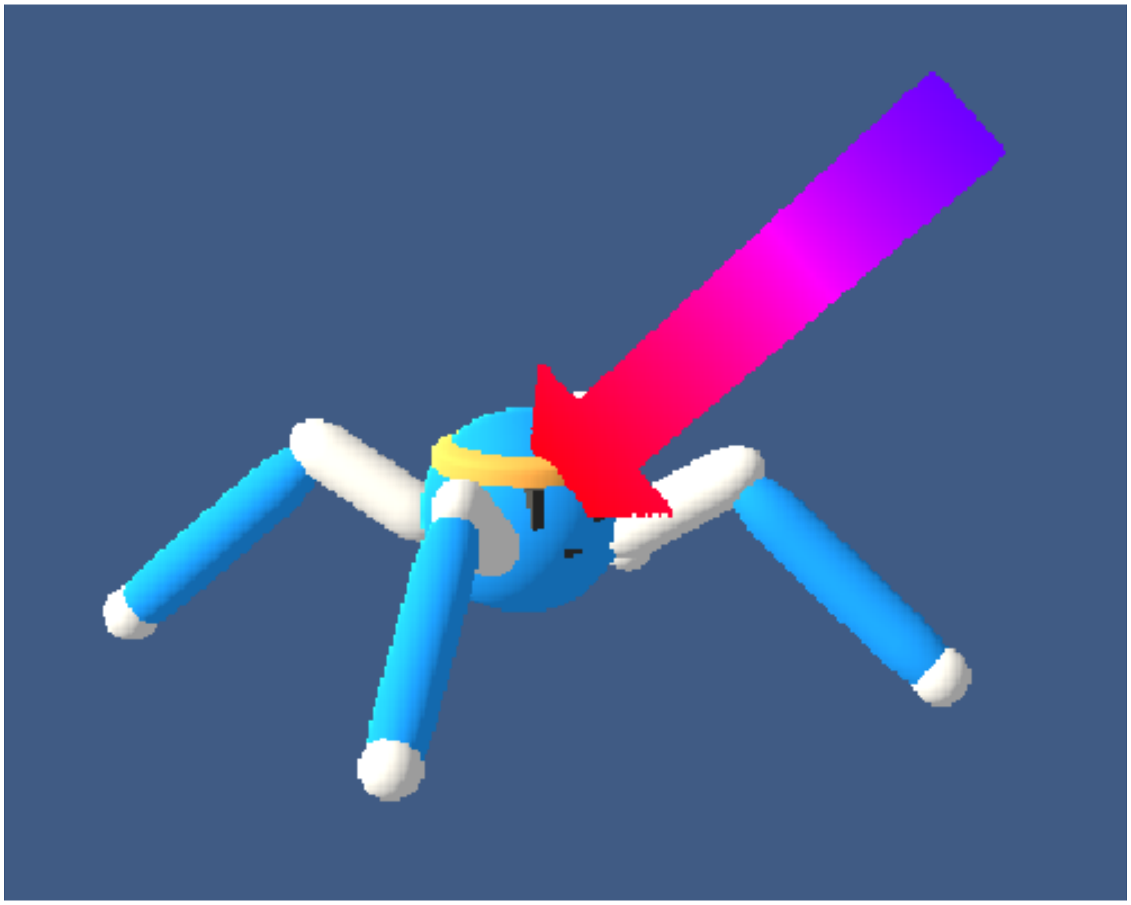


Fig. 8. Crawler agent with adversarial force applied to its body, represented by the colored arrow.

We chose to make the adversary a non-physical agent due to Pinto et al.’s [9] results using a similar approach. We gave the adversary preset values that limit the amount of force it is able to apply to the agent, based on trial and error, that prevent the adversary from instantly sending the agent flying off the terrain. Unlike with Pinto et al.'s [9] training method, we trained our agent and adversary concurrently, so they both actively learned how to outsmart each other as the training trial progressed.

Because the adversary is itself an agent, its learning procedure is similar to that of the learning agent. Each step, it takes in the same observations of the learning agent that it is attached to, and then decides on an action based on its past experience with other observations. The adversary may then choose to apply an adversarial action to one of the learning agent's limbs or body during its decision step, with actions such as modifying the selected limb's mass or friction, and applying a force to it in a direction of the adversary's choosing. As with a normal agent, the adversary performs the same chosen action every step until it selects a new action during its decision step. For our project, we set the adversary to make a new decision every 150 steps, which we found was enough steps for the adversary to have an impact on the agent without being too overwhelmingly powerful.

4.2.6 Terrain

We used transfer learning to train the agent on a simple terrain, and then gradually increased the difficulty of the training environment. In order to make use of transfer learning, we started off by training the crawler agent on a flat plane, and then gradually increased the slope of the terrain when the agent became proficient at flat plane locomotion. We procedurally generated slopes within a set range at the beginning of the training session and do not change the slopes for the entire session. Each agent being trained has its own unique set of slopes to traverse to reduce the chance of overfitting. Fig. 9 shows an example of a procedurally generated terrain seen from the side.

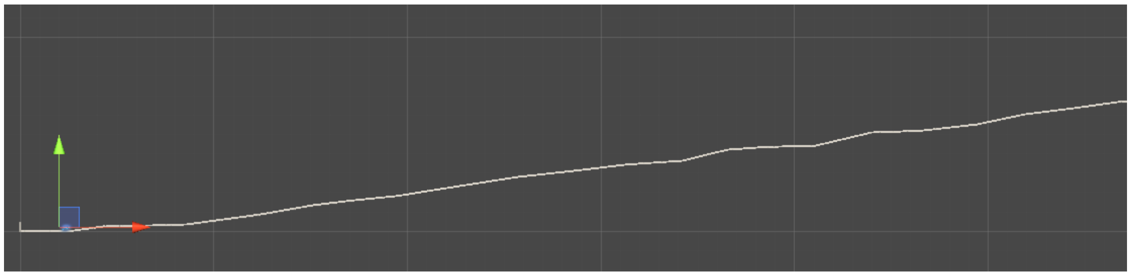


Fig. 9. Sample of a randomly-generated terrain with crawler's starting position located at the axis arrows.

4.3 Training Conditions

4.3.1 Flat Solo

We started off by training an agent alone on a simple flat terrain without an adversary, with the goal of creating an agent that can proficiently traverse its training environment. Having an agent that can proficiently traverse flat terrain gave us a starting point to make use of transfer learning to gradually teach the agent to traverse increasingly complex terrain.

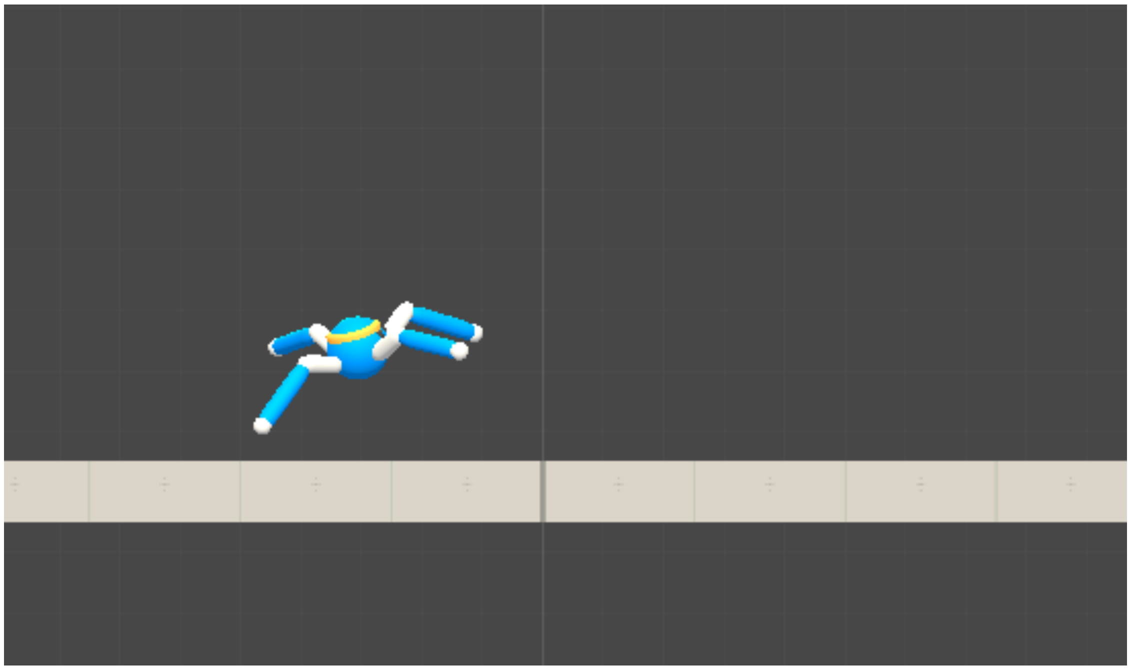


Fig. 10. Agent on flat terrain.

4.3.2 Flat Adversarial

For our adversarial approach, we initially trained our learning agent on a flat plane, similarly to the Flat Solo trial except with an adversary attached to the agent. The idea was to train the agent to be proficient at traversing the flat plane while constantly having external forces and changes in friction and mass applied to it by the adversary. Results of the research by Pinto et al. [9] suggest that a similar adversarial approach will result in a robust locomotive agent. The Flat Adversarial brain is separate from the Flat Solo brain, and is similarly used with transfer learning to teach the agent to traverse increasingly complex terrain while simultaneously being attacked by the adversary.

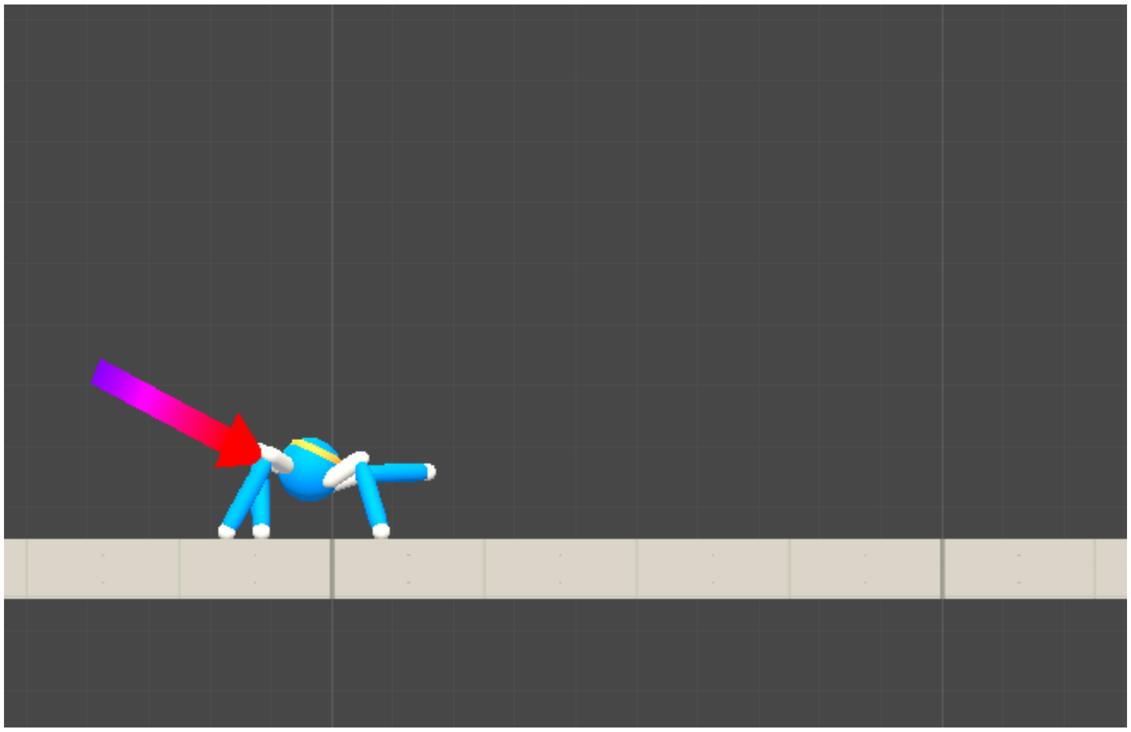


Fig. 11. Agent on flat terrain with adversarial force enabled.

4.3.3 Flat Adversary

Since the adversary is itself a learning agent, it also has a brain. As with the Flat Solo and Flat Adversarial locomotive agent brains, the Adversary brain is used with transfer learning to teach the adversary how to deal with increasingly complex terrain. Because the adversary is a parasitic agent, its brain corresponds with the brain of its host agent, so the Flat Adversary brain is paired with the Flat Adversarial brain, while the Sloped Adversary brain is paired up with the Sloped Adversarial brain.

4.3.4 Sloped Solo (Baseline)

We made use of transfer learning to gradually ease the crawler agent to become masterful at slope climbing by continuing to train the Flat Solo brain on terrain with gradually increasing slopes. The terrain was procedurally generated at the beginning of the run and was split into segments of varying slopes, where the slopes are randomized within a set range to allow the agent to experience new slope angles as well as the non-sloped terrain that it has already mastered. We increased the range of slopes with each successive run of a set of episodes, so the agent was allowed to become proficient at traversing a fixed range of slopes for a set number of episodes before the slope range was increased. The slope range was not designed to automatically increase after a set number of episodes had passed, and instead needed to be set manually. As a result of this, the training session needed to be stopped and restarted in order to increase the slope range, so the agent was set to train with the same set of slopes for each entire training or testing session. The Sloped Solo brain, which we will refer to as the baseline brain, would later be used as our control to compare against our brains trained with an adversary and in environments with varying slopes.

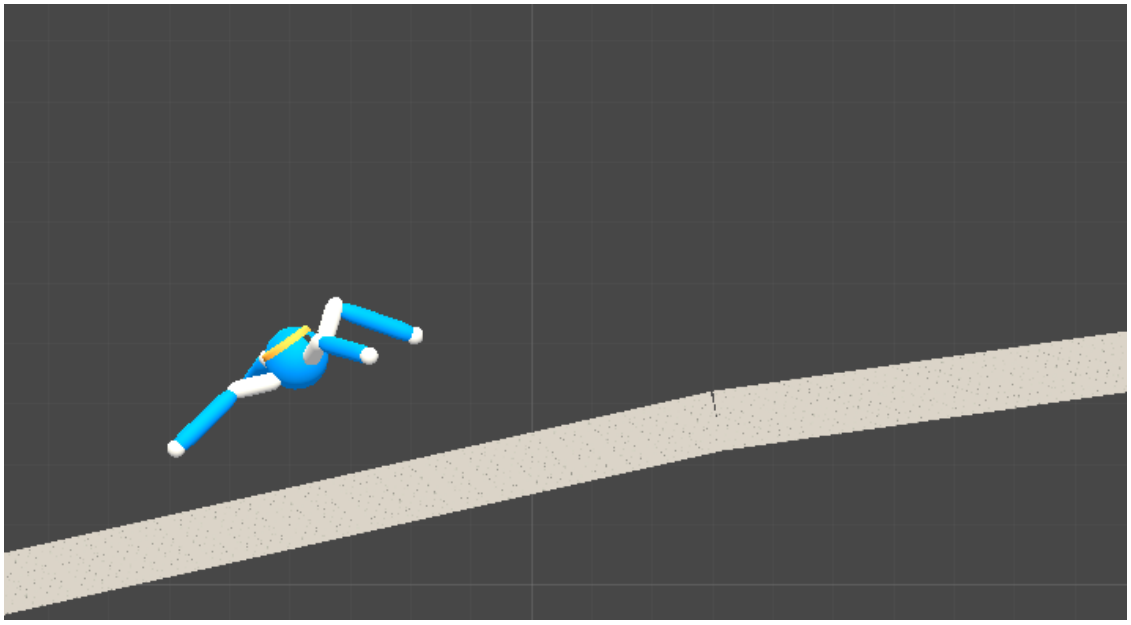


Fig. 12. Agent on sloped terrain.

4.3.5 Sloped Adversarial

As with the baseline brain, we made use of transfer learning and continued training the Flat Adversarial brain on terrain of increasing sloped ranges with the intent of increasing the robustness of our adversarial locomotive agent. The slope generation method and episode counts for the Sloped Adversarial training session was the same as the one used for the baseline trial, with the only difference being the presence of the adversary that learns alongside the locomotive agent. As with the baseline trial, the slope ranges were fixed for each set of episodes, where we only changed the slope range at the beginning of each training session.

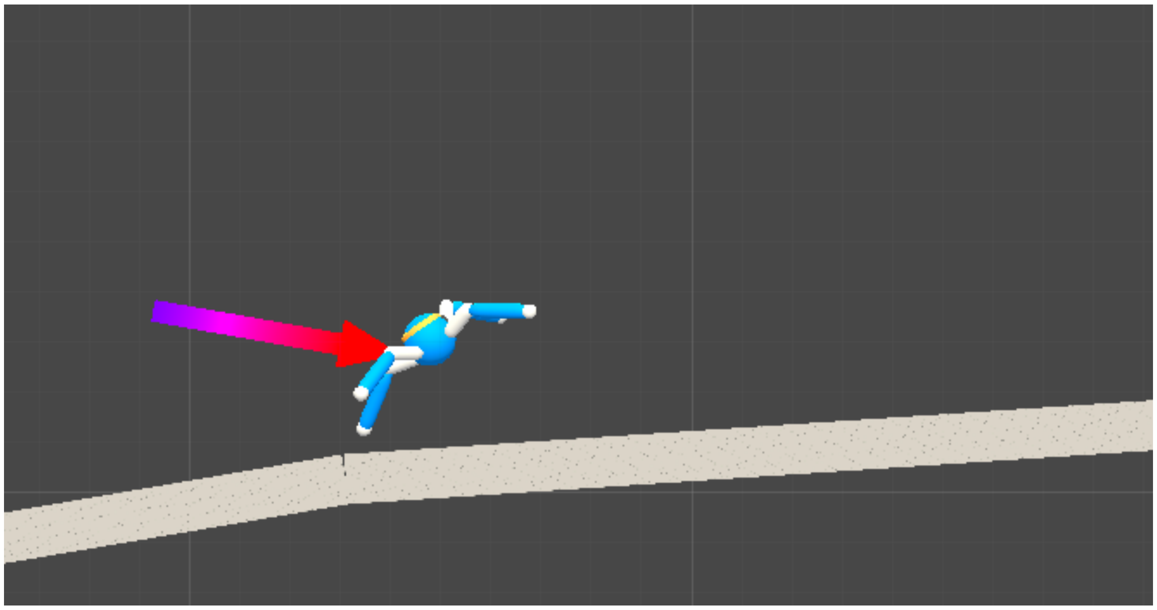


Fig. 13. Agent on sloped terrain with adversarial force enabled.

4.3.6 Sloped Adversary

Analogous to the previously-mentioned Sloped brains, the Sloped Adversary brain was an expansion of the Flat Adversary brain that learned to interrupt a Sloped Adversarial agent that was learning to traverse an increasingly complex sloped terrain.

4.4 Training Results

The results of the training for the baseline, adversarial, and adversary brains can be viewed in Fig. 14, where the average cumulative reward is plotted for each successive episode. Because we trained multiple agents concurrently, the average cumulative reward represents the average of the total reward accumulated by all trained agents for that episode. A higher cumulative reward means that the agent was more successful during its run, since it was making more "correct" actions that increased its reward.

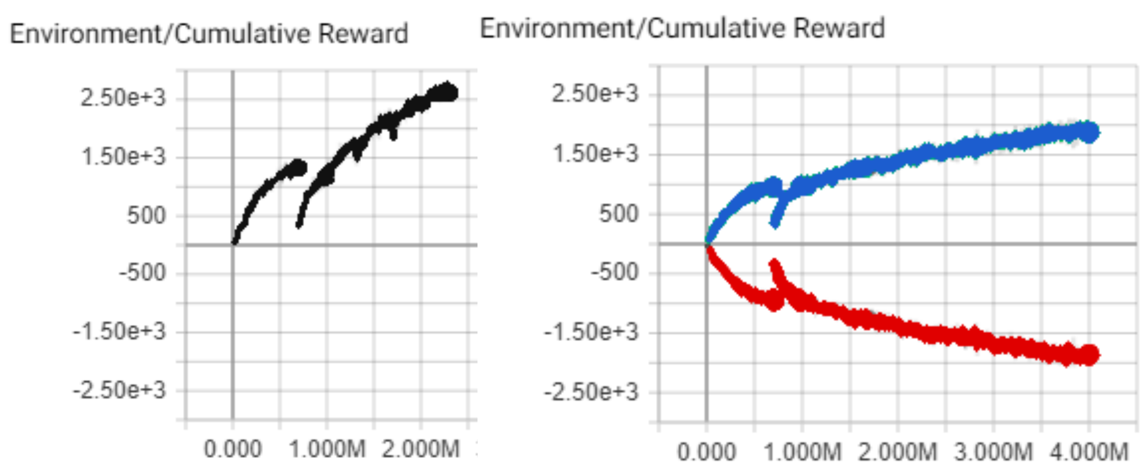


Fig. 14. Graphs show cumulative reward over multiple episodes for training baseline (left) and adversarial (right) brains. Initial arc indicates the portion of the training where the brains were trained on a flat plain. The following parts of the graphs indicate when training was moved onto gradually-increasing sloped terrain. For the adversarial brains, the blue curve shows the reward for the agent brain, while the red curve shows the reward for the adversary brain.

4.5 Test Scenarios

To test the robustness of our trained agents, we paused the learning of each agent's brain and let the agents traverse the testing environment using their knowledge gained during the training phase. The training environment consisted of gradually-increasing, procedurally generated slopes, similar to the training environment, but with much steeper slopes. In order to add more variety and unexpected situations to the training environment and simulate real-world situations where the robot implementing the agent's brain does not match the crawler's physical parameters exactly, the trials were performed for each slope setting with different parameters such as with trained adversary enabled, adjusted agent body mass, friction, and with projectiles randomly fired at the agent.

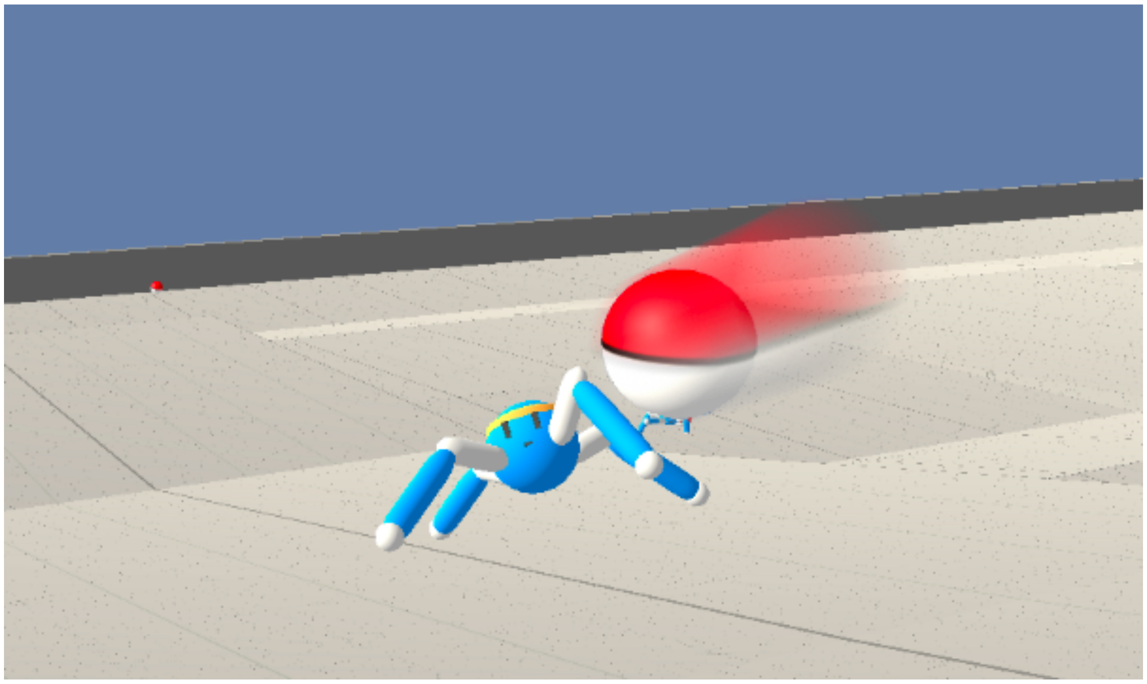


Fig. 15. Testing session with a projectile randomly fired at the agent.

Each trial consisted of a specific combination of parameters and was split into the following: Flat, Adversary Flat, Sloped, Adversary Sloped, Sloped with Projectile, Sloped with Different Mass, Sloped with Different Friction, Sloped with Different Mass and Projectiles, and Sloped with Different Friction and Projectiles. The exact parameters for the tests can be found in the following results section along with their corresponding results. The effectiveness of a brain during its test is rated by the total amount of reward that the agent is able to accumulate throughout its run with a particular parameter setting.

4.5.1 Flat

We first tested the baseline and Sloped Adversarial brains on a simple flat terrain to confirm that they are comparable with each other. For the entirety of our tests, we tested these two brains only, with the Sloped Adversarial brain as our main brain to test and the baseline brain as our baseline control. We set the crawler agent to have a default body mass of 1 unit and with a leg/foot friction value of 0.6 units. The agents were run for 2000 episodes before the testing trial was terminated. Twelve agent/adversary pairs were trained concurrently to speed up the training procedure. The resulting data shows that the two brains are comparable with each other.

Table 1
Test on Flat Terrain

Brain	**Average Cumulative Reward**
Baseline	3080.375
Sloped Adversarial	3089.161

4.5.2 Adversary Flat

Our next test was to have the baseline and Sloped Adversarial crawlers traverse the same flat terrain but with the fully-trained adversary enabled to add external disturbances to the locomotive agents. Since this test uses the trained adversary brain that is intended to actively target the crawler's weak points, we expected a drastic drop in total reward at the end of the test.

Table 2
Test on Flat Terrain with Adversary Enabled

Brain	**Average Cumulative Reward**
Baseline	1584.717
Sloped Adversarial	2191.16

As expected, there was a notable drop in total reward, especially for the baseline brain that had never learned to deal with adversarial disturbances. While the Sloped Adversarial agent performed 29% worse than with the default Flat test, it still performed 27.7% better than the baseline agent when traversing terrain with an adversarial disturbance.

4.5.3 Sloped

For our sloped terrain test, we used the same procedurally generated sloped terrain method as in our training sessions, with four separate trials for each slope range.

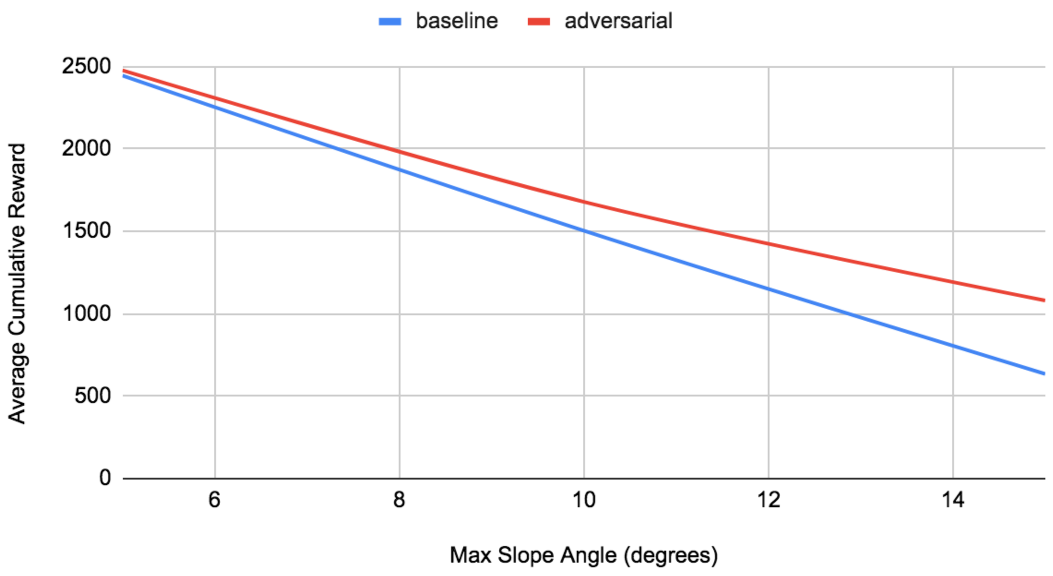


Fig. 16. The graph shows how the adversarial brain was more robust to changes in slope angle steepness.

As the slope ranges increased, the difference between the baseline agent and Sloped Adversarial agent's total reward also increased, suggesting that the Sloped Adversarial agent was more effective at dealing with newly-encountered terrain.

4.5.4 Adversary Sloped

As with the Adversary Flat test, we ran the Sloped test again but with the adversary enabled. This test case was intended to test how our agents would respond to unfamiliar terrain as well as unexpected disturbances.

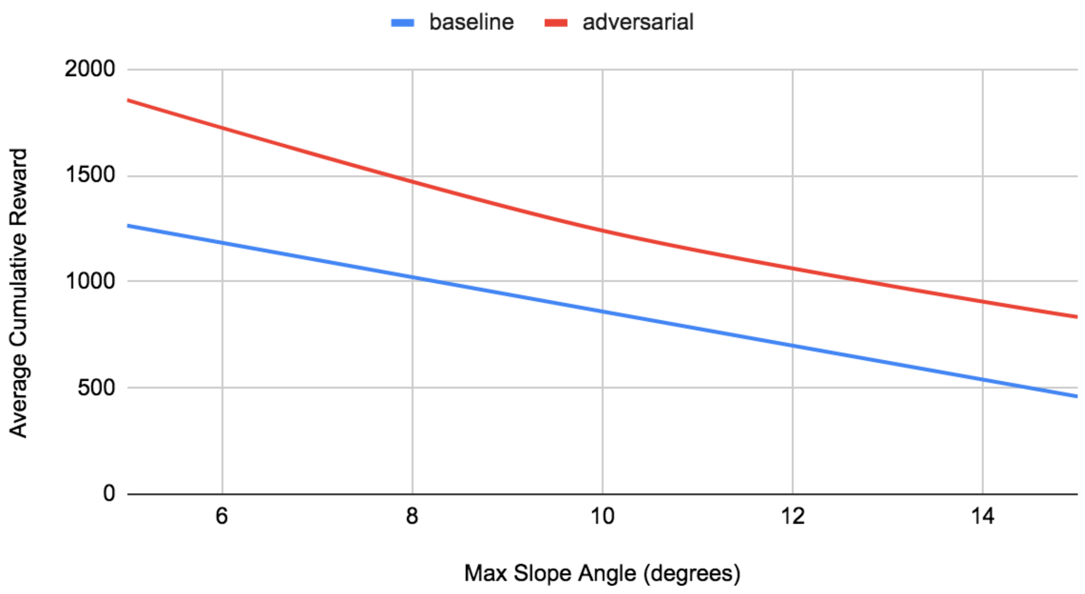


Fig. 17. The graph shows how the adversarial brain was more robust to changes in slope angle steepness while the trained adversary actively applied forces to the agent.

Upon completion of the test, both agents continued to yield lower scores as the terrain complexity increased, with the Adversarial agent obtaining twice the reward of the Solo agent when the slope range was 0-15 degrees.

4.5.5 Sloped with Projectiles

After testing the agents against terrain with an intelligent disturbance agent, we tried testing the agents against an unintelligent agent to see how well-equipped they were to deal with complete randomness. For this test, we randomly threw projectiles of random mass with random force at the agents as they traversed the procedurally-generated sloped terrain.

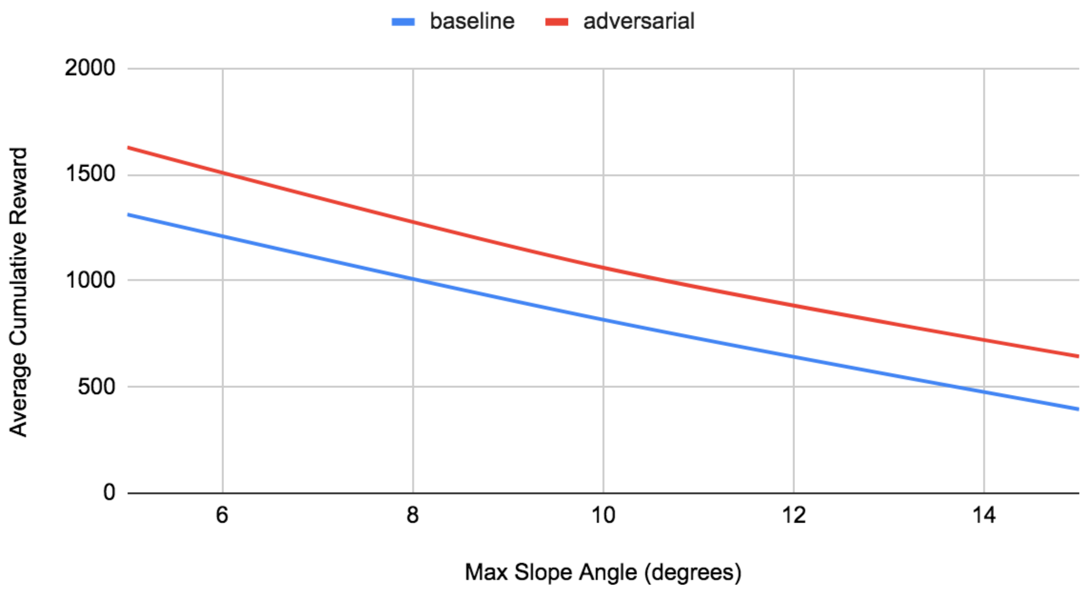


Fig. 18. The graph shows how the adversarial brain was more robust to changes in slope angle steepness while projectiles were randomly thrown at the agent.

While the Sloped Adversarial agent managed to perform better than the baseline agent, it suffered a noticeable drop in performance when compared to its reward score during the Adversary Sloped test. Meanwhile, the baseline's reward score was close to its score during the Adversary Sloped test.

4.5.6 Sloped with Varied Mass

Since the purpose of a virtual locomotive agent is to eventually be able to apply its learned behavior to a physical robot of varying parameters, we tested the brains in crawlers of varying parameters, starting with different mass values.

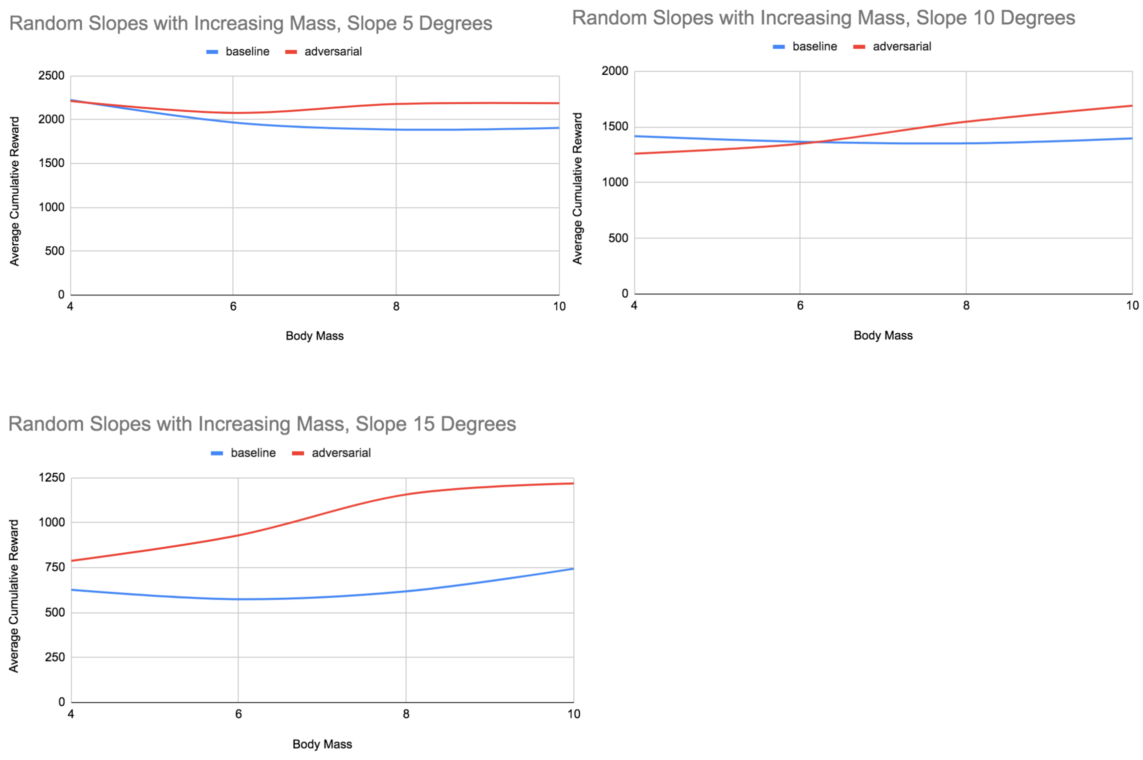


Fig. 19. The above graphs show how the adversarial brain was generally more robust to changes in mass. The slopes in the graph titles refer to the max slope in degrees, with minimum slope 0 degrees.

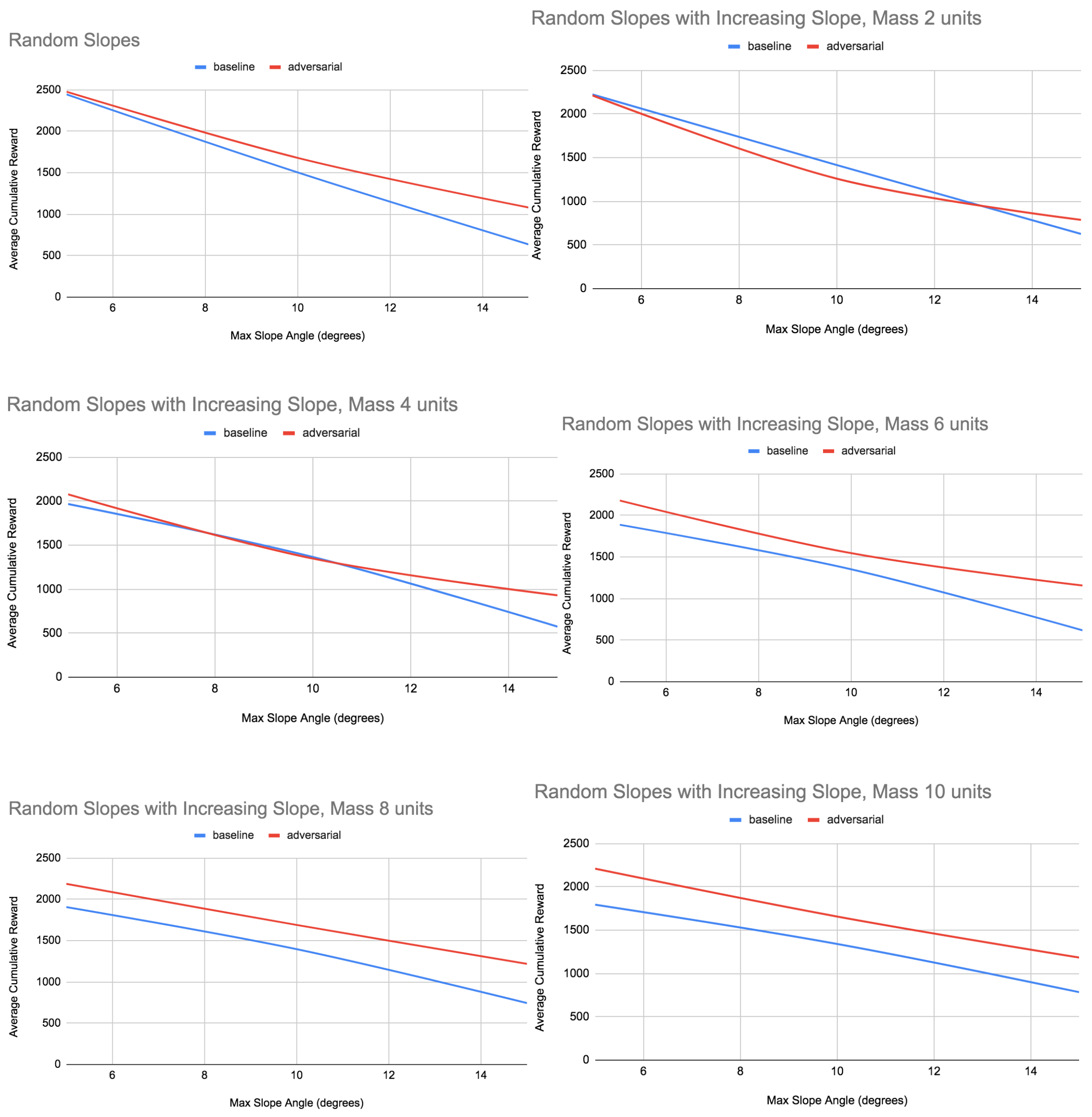


Fig. 20. The graph for Random Slopes refers to the default test case with mass set to 1 unit. The graphs depict the same data as the previous set of graphs (Fig. 19), but with mass shown as constant and with slope angle increasing. The adversarial brain generally performed better when mass was high.

The results generally showed that the agent trained alongside an adversary to be more effective at traversing terrain than the baseline, although there was a lag in performance as mass increased. The downwards slope in reward as max angle increased indicates that both the adversarial and baseline were susceptible to changes in terrain angle, although the adversarial agent continued to perform better than the baseline.

4.5.7 Sloped with Projectiles and Varied Mass

For our final two tests, we repeated the previous mass and friction tests with projectiles enabled to simulate the case where varying robots encounter unintelligent external phenomena.

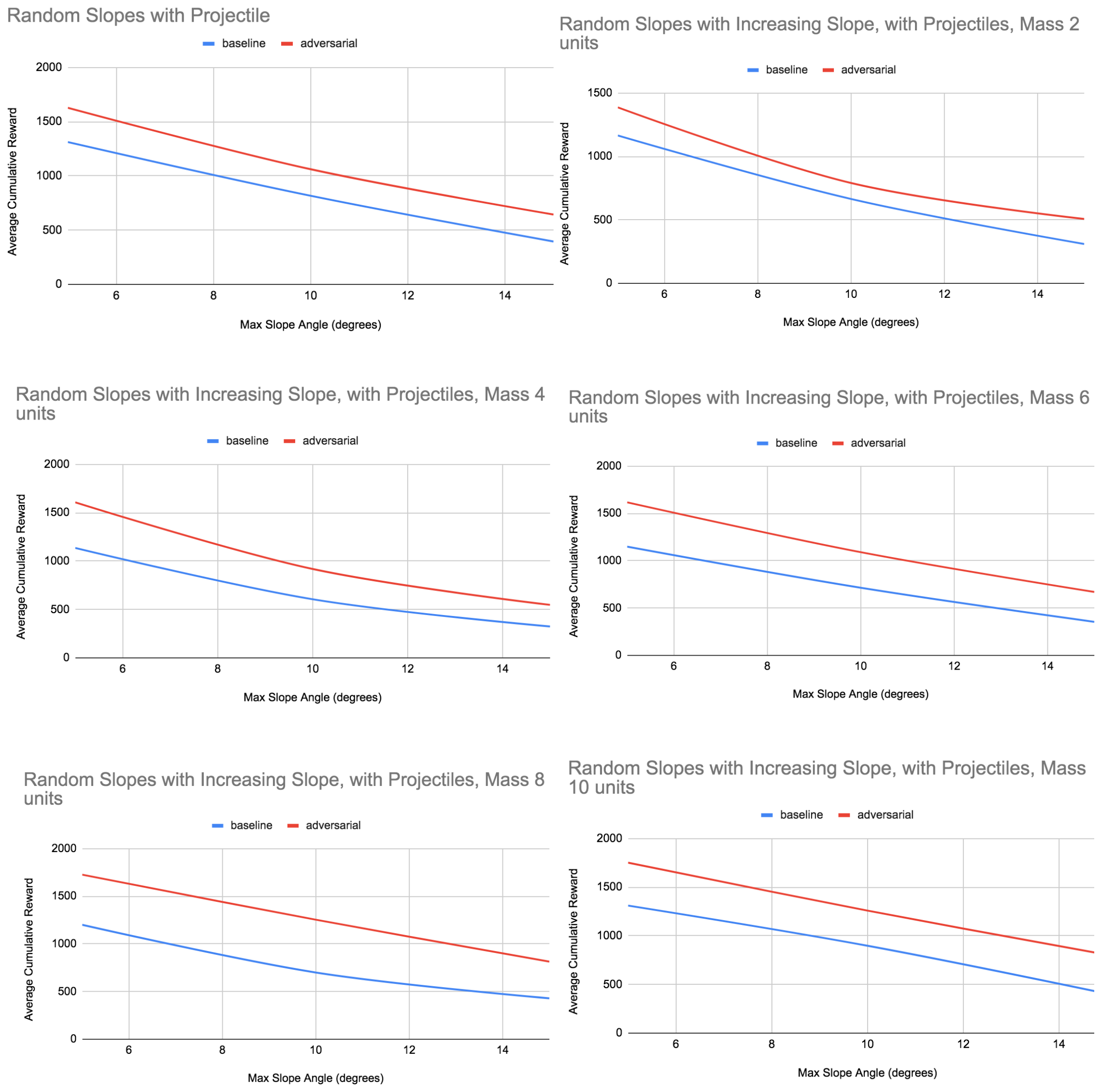


Fig. 21. The graph for Random Slopes with Projectile(top-left) refers to the test with mass set to 1 unit. The graphs show how the adversarial brain was more robust to changes in slope angle steepness while projectiles were randomly thrown at the agent, and with different body masses set for the agent.

As expected from the previous projectile test, the total rewards for this test were continued in a downwards trend as max slope angle increased, but the adversarial brain continued to show better results than the baseline. The results suggest that the adversarial brain is better at adapting to agents of different mass than what it was trained with, compared to the baseline brain.

4.5.8 Sloped with Varied Friction

Since the material of a physical robot may not always be consistent, we performed multiple tests with the agent's foot friction adjusted for each test in order to simulate real-world variances in materials.

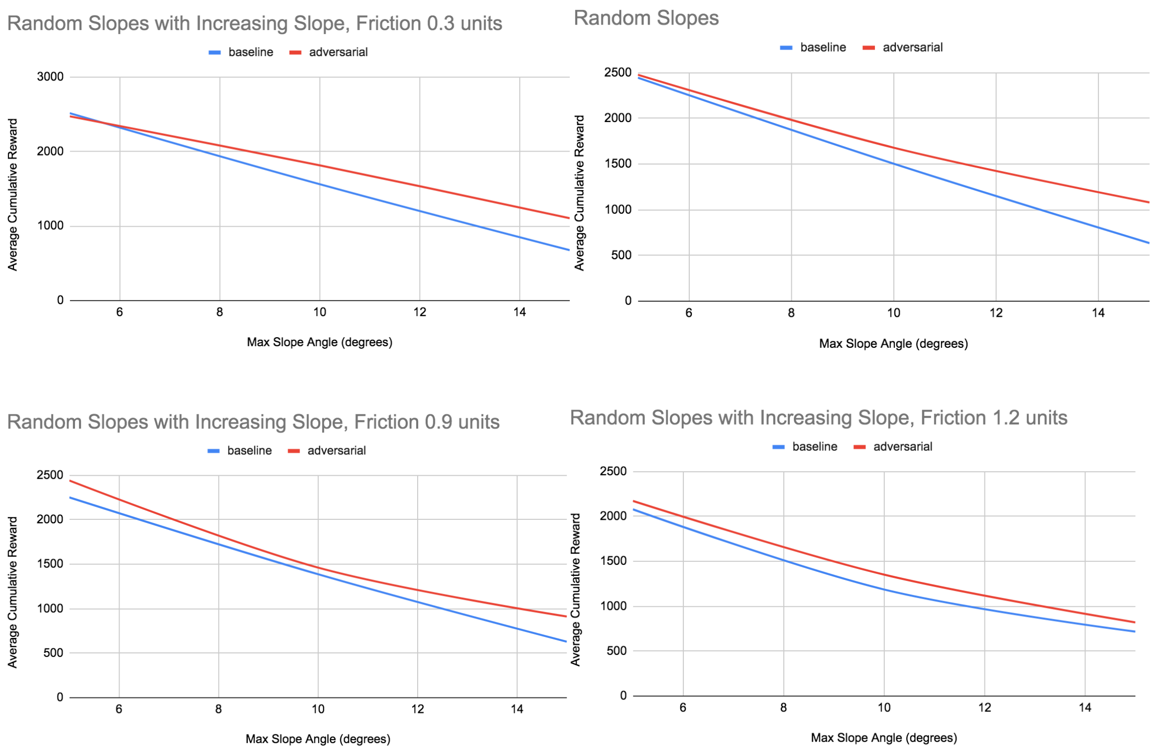


Fig. 22. The graph for Random Slopes(top-right) refers to the default test case with friction set to 0.6. The graphs show how the adversarial brain was more robust to changes in slope angle steepness when traversing terrain of varying friction values.

The adversarial brain continues to show better results, despite the slight lag in performance during the initial change in friction. The differences in results are smaller than in previous tests, but are still noticeable.

4.5.9 Sloped with Projectiles and Varied Friction

Continuing our simulated real-world test, we tested the agent with different foot friction while having projectiles thrown at it.

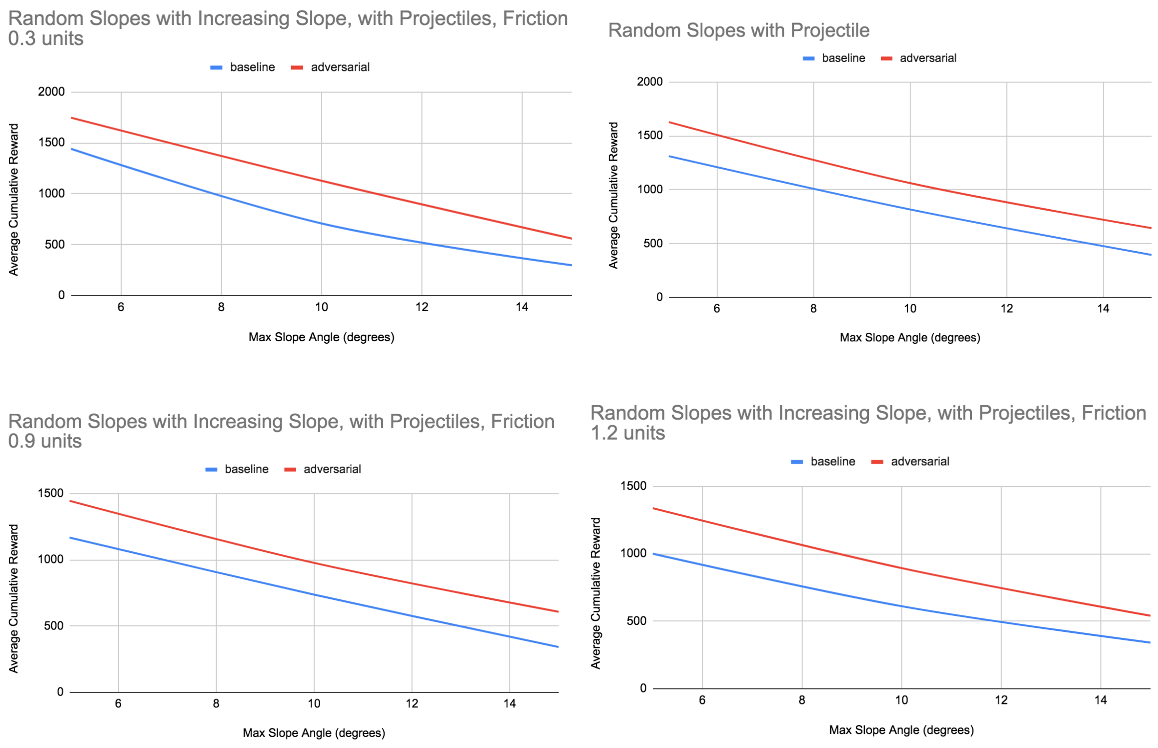


Fig. 23. The Random Slopes with Projectile graph refers to the default test case with friction set to 0.6. The graphs show how the adversarial brain was more robust to changes in slope angle steepness when traversing terrain of varying friction values and while being bombarded with random projectiles.

As with the previous test, the Adversarial Agent was more effective at maintaining balance while under attack by random projectiles and with adjusted foot friction values. The difference in performance between the adversarial brain and the baseline is similar to the previous test, but the addition of random projectiles to the test brought both cumulative rewards down even further.

5 CONCLUSION

In our tests, an adversarial approach to training a locomotive agent was found to be more effective than training an agent alone. Our adversarial agent was found to be more robust than our baseline agent, being able to better adapt to unexpected situations such as varying terrain slopes, friction, and body mass, as well as to being under attack by external forces such as a trained adversary and random projectiles.

With our current training setup, the training terrains were generated at the start of the training session and did not change at any time during training, which made it possible for agents to overfit to the currently-generated terrain slopes. An improvement would be to make the terrain regenerate when the agent concludes an episode of training so that its next lap would involve completely new terrain that the agent had never encountered yet, although with the same range of slope angles that can be randomly generated.

Our testing conditions only accounted for mass and friction, as far as the agent's physical properties were concerned, and we did not test agents with different proportions and weight distribution. Future tests could involve changing the agent's leg lengths to account for cases where a physical robot may not match the proportions of the crawler agent. Such tests would give us a better understanding of how robust the adversarial approach can accommodate for the physical properties of the agent itself rather than just variances in its environment.

References

[1] O. Tutsoy and M. Brown, "Convergence analysis of reinforcement learning approaches to humanoid locomotion," in *UKACC International Conference on Control 2010*, Coventry, 2010, pp. 1-6.

[2] Y. Mao et al., "A Reinforcement Learning Based Dynamic Walking Control," in *Proceedings 2007 IEEE International Conference on Robotics and Automation*, Roma, 2007, pp. 3609-3614.

[3] C. Fu and K. Chen, "Gait Synthesis and Sensory Control of Stair Climbing for a Humanoid Robot," *IEEE Transactions on Industrial Electronics,* vol. 55, no. 5, pp. 2111-2120, May 2008.

[4] "Robot," 12 Mar. 2020. [Online]. Available: https://en.wikipedia.org/wiki/Robot.

[5] O. SeungSub, H. Jehun, J. Hyunjung, L. Soyeon and S. Jinho, "A study on the disaster response scenarios using robot technology," in *2017 14th International Conference on Ubiquitous Robots and Ambient Intelligence (URAI)*, Jeju, 2017, pp. 520-523.

[6] A. Unluturk and O. Aydogdu, "Design and implementation of a mobile robot used in bomb research and setup disposal," in *Proceedings of the International Conference on ELECTRONICS, COMPUTERS and ARTIFICIAL INTELLIGENCE - ECAI-2013*, Pitesti, 2013, pp. 1-5.

[7] A. K. B. Motaleb, M. B. Hoque and M. A. Hoque, "Bomb disposal robot," in *2016 International Conference on Innovations in Science, Engineering and Technology (ICISET)*, Dhaka, 2016, pp. 1-5.

[8] S. Sakakibara, "The latest robot systems which reinforce manufacturing sector," in *2003 IEEE International Conference on Robotics and Automation (Cat. No.03CH37422)*, Taipei, Taiwan, 2003, pp. 2878-2883 vol.2.

[9] L. Pinto, J. Davidson, R. Sukthankar and A. Gupta, "Robust Adversarial Reinforcement Learning," arXiv:1703.02702, Mar. 2017.

[10] Yu She, C. J. Hurd and H. Su, "A transformable wheel robot with a passive leg," in *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Hamburg, 2015, pp. 4165-4170.

[11] Y. Mae, A. Yoshida, T. Arai, K. Inoue, K. Miyawaki and H. Adachi, "Application of locomotive robot to rescue tasks," in *Proceedings. 2000 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2000) (Cat. No.00CH37113)*, Takamatsu, Japan, 2000, pp. 2083-2088 vol.3.

[12] "BigDog," 24 Mar 2020. [Online]. Available: https://en.wikipedia.org/wiki/BigDog.

[13] N. Heess, D. TB, S. Sriram, J. Lemmon, J. Merel, G. Wayne, Y. Tassa, T. Erez, Z. Wang, S. M. A. Eslami, M. Riedmiller and D. Silver, "Emergence of Locomotion Behaviours in Rich Environments," arXiv:1707.02286, Jul. 2017.

[14] G. Karimpanal, T. Roland and B. Roland, "Self-Organizing Maps for Storage and Transfer of Knowledge in Reinforcement Learning," arXiv:1811.08318 [cs.AI], Nov. 2018.

[15] L. Pinto, J. Davidson and A. Gupta, "Supervision via Competition: Robot Adversaries for Learning Tasks," arXiv:1610.01685, Oct. 2016.

[16] U. Garg, "Virtual Robot Climbing using Reinforcement Learning," Dec. 2018. [Online]. Available: https://scholarworks.sjsu.edu/etd_projects/658. [Accessed Sept. 2019].

[17] J. Schulman, S. Levine, P. Abbeel, M. Jordan and P. Moritz, "Trust region policy optimization," in *International Conference on Machine Learning*, 2015, pp. 1889-1897.

[18] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver and D. Wierstra, "Continuous control with deep reinforcement learning," arXiv:1509.02971v1, Sept. 2015.

[19] "Legged Robot," 25 Feb. 2020. [Online]. Available: https://en.wikipedia.org/wiki/Legged_robot.

[20] G. A. Bekey, Autonomous robots: from biological inspiration to implementation and control, Cambridge, Massachusetts: MIT Press, 2005.

[21] "Inverted Pendulum," 3 Mar. 2020. [Online]. Available: https://en.wikipedia.org/wiki/Inverted_pendulum.

[22] Akash, S. Chandra, Abha and G. C. Nandi, "Modeling a bipedal humanoid robot using inverted pendulum towards push recovery," in *2012 International Conference*

on Communication, Information & Computing Technology (ICCICT), Mumbai, 2012, pp. 1-6.

[23] Y. Sun, X. Yuan, W. Liu and C. Sun, "Model-Based Reinforcement Learning via Proximal Policy Optimization," in *2019 Chinese Automation Congress (CAC)*, Hangzhou, China, 2019, pp. 4736-4740.

[24] "ML-Agents Toolkit Overview," 18 Mar. 2020. [Online]. Available: https://github.com/Unity-Technologies/ml-agents/blob/master/docs/ML-Agents-Overview.md.

[25] "Academy.cs," 11 Mar. 2020. [Online]. Available: https://github.com/Unity-Technologies/ml-agents/blob/master/com.unity.ml-agents/Runtime/Academy.cs.

[26] "Normal (geometry)," 1 Feb. 2020. [Online]. Available: https://en.wikipedia.org/wiki/Normal_(geometry).

Appendix A: Source Code

CrawlerAcademy2.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using MLAgents;
using UnityEditor;

/// <summary>
/// Academy for Crawler and Adversary agents.
/// Has the option to configure agent’s body mass and feet friction,
/// and run with projectiles randomly thrown at the agent.
/// author: Phong Nguyen
/// with minor code segments from Unity ML-Agents
/// </summary>
public class CrawlerAcademy2 : CrawlerAcademy, Testable
{
    // Test values
    float testReward = 0;
    float testCRreward = 0; //cumulative reward
    public int testEpisodeLimit = -1; //print results and terminate when limit reached
    int testEpisodeCount = 0;
    public bool isTesting = false; //set to true if you want to print reward data
    public bool throwProjectiles = false;
    public float bodyMass = 1f;
    public float forelegFriction = 0.6f;
    public float forelegBounciness = 0f;
    public List<Rigidbody> bodies;
    string stringBody = "Body";
    public List<Collider> forelegs;
    string stringForeleg = "foreleg";
    public int projectileThrowInterval = 15;
    int currentStepInterval = 0;
    ProjectileThrower projectileThrower;
    public GameObject baseProjectile;

    public class ProjectileThrower
    {
        public int projectileCount = 50;
        GameObject baseProjectile;
        Stack<GameObject> inactiveProjectiles;
        Dictionary<GameObject, Rigidbody> rigidbodyDictionary;
        GameObject currentProjectile;

        public void Init(GameObject proj)
        {
            inactiveProjectiles = new Stack<GameObject>();
            rigidbodyDictionary = new Dictionary<GameObject, Rigidbody>();
            baseProjectile = proj;
            //fill projectile stack
            if (baseProjectile != null)
            {
                //spawn projectiles
                for (int i = 0; i < projectileCount; i++)
                {
                    currentProjectile = Instantiate(baseProjectile);
                    currentProjectile.SetActive(false);
                    currentProjectile.GetComponent<ProjectileScript>().Init(this);
                    rigidbodyDictionary.Add(currentProjectile,
currentProjectile.GetComponent<Rigidbody>());
                    inactiveProjectiles.Push(currentProjectile);
                }
            }
            else
            {
                Debug.Log("No base projectile found Cannot throw projectiles");
                return;
            }
        }

        /// <summary>
        /// Select a projectile if any and throw at target
        /// </summary>
        public void ThrowProjectile(Rigidbody target) {
            if (inactiveProjectiles.Count > 0)
            {
                //select a random position around target
                currentProjectile = inactiveProjectiles.Pop();
                //get pos 5m away from target, rotate by random amt w/ target as pivot
                currentProjectile.transform.position = target.transform.position +
target.velocity*Time.deltaTime + new Vector3(0, Random.Range(1f, 2f),4f);
                currentProjectile.transform.RotateAround(target.transform.position + target.velocity *
Time.deltaTime + Vector3.up*.5f, Vector3.up, Random.Range(0,720f));
                currentProjectile.transform.LookAt(target.transform.position + target.velocity *
Time.deltaTime, Vector3.up);
                currentProjectile.SetActive(true);
                //apply a force to projectile towards target
                rigidbodyDictionary[currentProjectile].AddForce(currentProjectile.transform.forward *
3000f);
            }
        }

        /// <summary>
        /// Reset and disable projectile, and store it for future use
        /// </summary>
        public void CleanupProjectile(GameObject p)
        {
            rigidbodyDictionary[p].velocity = Vector3.zero;
            rigidbodyDictionary[p].angularVelocity = Vector3.zero;
            p.SetActive(false);
            inactiveProjectiles.Push(p);
        }
```

```
}

/// <summary>
/// Initialize academy and setup projectiles if testing with projectiles
/// </summary>
public override void InitializeAcademy()
{
   base.InitializeAcademy();
   if (isTesting)
   {
      if (throwProjectiles)
      {
         projectileThrower = new ProjectileThrower();
         projectileThrower.Init(baseProjectile);
      }

      bodies = new List<Rigidbody>();
      //setup body mass
      foreach (Rigidbody rb in GameObject.FindObjectsOfType<Rigidbody>())
      {
         if (rb.CompareTag(stringBody))
         {
            bodies.Add(rb);
            rb.mass = bodyMass;
         }
      }
      forelegs = new List<Collider>();
      //setup feet friction
      foreach (Collider col in GameObject.FindObjectsOfType<Collider>())
      {
         if (col.CompareTag(stringForeleg))
         {
            forelegs.Add(col);
            col.material.staticFriction = forelegFriction;
            col.material.dynamicFriction = forelegFriction;
            col.material.bounciness = forelegBounciness;
         }
      }
   }
}

public override void AcademyReset()
{
   testReward = 0;
   testCRreward = 0;
   testEpisodeCount = 0;
}

/// <summary>
/// Throw projectile if current step is the throw projectile step
/// </summary>
public override void AcademyStep()
{
        currentStepInterval++;
        if (throwProjectiles && currentStepInterval == projectileThrowInterval)
        {
            currentStepInterval %= projectileThrowInterval;
            projectileThrower.ThrowProjectile(bodies[Random.Range(0, bodies.Count)]);
        }
    }

    public void CollectReward(float r)
    {
        if (isTesting)
        {
            testReward += r;
        }
    }

    public void CollectCumulativeReward(float c)
    {
        if (isTesting)
        {
            if (testEpisodeCount < testEpisodeLimit)
            {
                testCRreward += c;
                testEpisodeCount += 1;
            }

            if (testEpisodeLimit > 0 && testEpisodeCount >= testEpisodeLimit)
            {
                //limit reached, terminate
                ExitPlayMode();
            }
        }
    }

    /// <summary>
    /// Terminate the test session
    /// </summary>
    public void ExitPlayMode()
    {
        EditorApplication.isPlaying = false;
    }

    /// <summary>
    /// Print out reward data for review when test session is terminated
    /// </summary>
    void OnApplicationQuit()
    {
        if (isTesting)
        {
            //print data
            Debug.Log("episodes completed: " + testEpisodeCount);
            Debug.Log("threw projectiles: " + throwProjectiles);
            Debug.Log("mass: " + bodyMass + ", friction: " + forelegFriction + ", bounce: " +
forelegBounciness);
            Debug.Log("total reward: " + testReward);
            Debug.Log("avg reward: " + testReward / ((float)testEpisodeCount));
            Debug.Log("total c reward: " + testCRreward);
            Debug.Log("avg c reward: " + testCRreward / ((float)testEpisodeCount));
            Debug.Log("Done testing, now terminating!");
        }
    }
}
```

CrawlerAdversaryAgent.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using MLAgents;

/// <summary>
/// Adversary agent that applies forces to its host agent
/// author: Phong Nguyen
/// with minor code segments from Unity ML-Agents
/// </summary>
public class CrawlerAdversaryAgent : Agent
{
    bool isNewDecisionStep;
    int currentDecisionStep;

    //host agent modifiers
    [SerializeField] CrawlerAgent hostAgent;
    JointDriveController hostJdController;
    [SerializeField] float maxAcceleration = 5f;
    float maxForce;
    Vector3 forceDirection;
    [SerializeField] float maxFriction;
    [SerializeField] float maxBounciness;
    [SerializeField] float maxMass;
    [SerializeField] float minMass;
    [SerializeField] bool visualizeForce = false;
    [SerializeField] Transform forceSprite; //sprite used for visualization

        //host limbs
        [System.Serializable] struct Limb
    {
        [SerializeField] public Transform tf;
        [SerializeField] public Rigidbody rb;
        [SerializeField] public Collider col;
    }
    [SerializeField] Limb[] hostLimbs;
    struct LimbValues
```

```
{
    public float mass;
    public float staticFriction;
    public float dynamicFriction;
    public float bounciness;
}
List<LimbValues> hostLimbDefaults;

public override void InitializeAgent()
{
    if (visualizeForce)
    {
        forceSprite.gameObject.SetActive(true);
    }
    else
    {
        forceSprite.gameObject.SetActive(false); //disable until needed
    }

    // Save limb default values
    hostLimbDefaults = new List<LimbValues>();
    for (int i=0; i < hostLimbs.Length; i++)
    {
        LimbValues newValues = new LimbValues();
        newValues.mass = hostLimbs[i].rb.mass;
        newValues.staticFriction = hostLimbs[i].col.material.staticFriction;
        newValues.dynamicFriction = hostLimbs[i].col.material.dynamicFriction;
        newValues.bounciness = hostLimbs[i].col.material.bounciness;
        hostLimbDefaults.Add(newValues);
    }

    // Calculate max force
    maxForce = maxMass * maxAcceleration; //F=ma
}

/// <summary>
/// Collect observations from a specific body part
/// Observations are from Unity ML-Agents project's CrawlerAgent class
/// </summary>
public virtual void CollectObservationBodyPart(BodyPart bp)
{
    var rb = bp.rb;
    AddVectorObs(bp.groundContact.touchingGround ? 1 : 0);
    AddVectorObs(rb.velocity);
    AddVectorObs(rb.angularVelocity);

    if (bp.rb.transform != hostAgent.body)
    {
        Vector3 localPosRelToBody = hostAgent.body.InverseTransformPoint(rb.position);
        AddVectorObs(localPosRelToBody);
        AddVectorObs(bp.currentXNormalizedRot); // Current x rot
        AddVectorObs(bp.currentYNormalizedRot); // Current y rot
        AddVectorObs(bp.currentZNormalizedRot); // Current z rot
```

```
            AddVectorObs(bp.currentStrength / hostAgent.jdController.maxJointForceLimit);
        }

        //collision data
        AddVectorObs(bp.groundContact.lastContactPositionX);
        AddVectorObs(bp.groundContact.lastContactPositionY);
        AddVectorObs(bp.groundContact.lastContactPositionZ);
        AddVectorObs(bp.groundContact.lastContactNormalX);
        AddVectorObs(bp.groundContact.lastContactNormalY);
        AddVectorObs(bp.groundContact.lastContactNormalZ);

        //height data
        if (bp.groundContact.checkForHeight)
        {
            AddVectorObs(bp.groundContact.GetHeightFromGround());
        }
    }

    /// <summary>
    /// Collect observations from the host agent
    /// Observations are the same as in Unity ML-Agents project's CrawlerAgent class
    /// </summary>
    public override void CollectObservations()
    {
        hostAgent.jdController.GetCurrentJointForces();
        // Normalize direction vector to help generalize
        AddVectorObs(hostAgent.dirToTarget.normalized);

        // Forward & up to help with orientation
        AddVectorObs(hostAgent.body.transform.position.y); //height
        AddVectorObs(hostAgent.body.forward);
        AddVectorObs(hostAgent.body.up);
        foreach (var bodyPart in hostAgent.jdController.bodyPartsDict.Values)
        {
            CollectObservationBodyPart(bodyPart);
        }

        // Center of mass to aid with balance
        hostAgent.CalculateCenterOfMass();
        AddVectorObs(hostAgent.centerOfMass.x);
        AddVectorObs(hostAgent.centerOfMass.y);
        AddVectorObs(hostAgent.centerOfMass.z);
    }

    /// <summary>
    /// Select one of the host agent's limbs and apply forces to it
    /// </summary>
    public override void AgentAction(float[] vectorAction, string textAction)
    {
        // Perform actions only if this is a new decision step
        if (isNewDecisionStep)
        {
            ResetLimbs();
            // Get ref to host agent’s body parts
            var bpDict = hostAgent.jdController.bodyPartsDict;

            int i = -1;
            // Select a limb to interrupt
            int j = Mathf.Clamp(j, 0, hostLimbs.Length - 1);
            hostLimbs[j].rb.mass = Mathf.Clamp(Mathf.Abs(vectorAction[++i]) * maxMass,
minMass, maxMass);
            // Apply force with resulting acceleration restricted to max value
            forceDirection.x = vectorAction[++i];
            forceDirection.y = vectorAction[++i];
            forceDirection.z = vectorAction[++i];
            forceDirection = Vector3.Normalize(forceDirection);
            hostLimbs[j].rb.AddForce(Mathf.Clamp(Mathf.Abs(vectorAction[++i])*maxForce /
hostLimbs[j].rb.mass,0,maxAcceleration*hostLimbs[j].rb.mass)*forceDirection,
ForceMode.Impulse);
            //F=ma, clamp F/m so that it is between 0 and maxAcceleration

            if (visualizeForce)
            {
                forceSprite.position = hostLimbs[j].col.transform.position;
                forceSprite.LookAt(hostLimbs[j].col.transform.position + forceDirection);
            }

            // Change friction and bounciness values
            hostLimbs[j].col.material.dynamicFriction = Mathf.Clamp(Mathf.Abs(vectorAction[++i]) *
maxFriction, 0, maxFriction);
            hostLimbs[j].col.material.staticFriction = Mathf.Clamp(Mathf.Abs(vectorAction[++i]) *
maxFriction, 0, maxFriction);
            hostLimbs[j].col.material.bounciness = Mathf.Clamp(Mathf.Abs(vectorAction[++i]) *
maxBounciness, 0, maxBounciness);

        }
    }

    void ResetLimbs()
    {
        // Reset limb values
        for (int i = 0; i < hostLimbs.Length; i++)
        {
            hostLimbs[i].rb.mass = hostLimbDefaults[i].mass;
            hostLimbs[i].col.material.staticFriction = hostLimbDefaults[i].staticFriction;
            hostLimbs[i].col.material.dynamicFriction = hostLimbDefaults[i].dynamicFriction;
            hostLimbs[i].col.material.bounciness = hostLimbDefaults[i].bounciness;
        }
    }

    public override void AgentReset()
    {
        ResetLimbs();
        isNewDecisionStep = true;
        currentDecisionStep = 1;
    }
```

```
}
```

CrawlerAgent2.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using MLAgents;

/// <summary>
/// Crawler agent that can be attacked by an adversary
/// author: Phong Nguyen
/// with minor code segments from Unity ML-Agents
/// </summary>
public class CrawlerAgent2 : CrawlerAgent
{
    public Vector3 centerOfMass = Vector3.zero;
    protected float totalMass = 0f;

    //heights
    float bodyHeight;
    float foot0Height;
    float foot1Height;
    float foot2Height;
    float foot3Height;

    [SerializeField] public CrawlerAdversaryAgent adversary;

    //tester is an Academy that can run tests and print out reward data
    Testable tester;

    protected override void OnEnableHelper(Academy academy)
    {
        base.OnEnableHelper(academy);
        if (tester == null && academy is Testable)
        {
            tester = academy as Testable;
        }
    }

    public override void InitializeAgent()
    {
        base.InitializeAgent();

        //Calculate total mass
        totalMass = 0f;
        foreach (var bp in jdController.bodyPartsDict.Values)
        {
            totalMass += bp.rb.mass;
        }
    }
```

```
/// <summary>
/// Calculate center of mass of the agent
/// </summary>
public void CalculateCenterOfMass()
{
   centerOfMass = Vector3.zero;
   totalMass = 0f;
   foreach (var bp in jdController.bodyPartsDict.Values)
   {
      centerOfMass += bp.rb.worldCenterOfMass * bp.rb.mass;
      totalMass += bp.rb.mass;
   }
   centerOfMass /= totalMass;

   /**
   //visualize center of mass
   if (com != null)
   {
      com.position = centerOfMass;
   }
   */
}

/// <summary>
/// Add relevant information on each body part to observations.
/// </summary>
public override void CollectObservationBodyPart(BodyPart bp)
{
   base.CollectObservationBodyPart(bp);

   //collision data
   AddVectorObs(bp.groundContact.lastContactPositionX);
   AddVectorObs(bp.groundContact.lastContactPositionY);
   AddVectorObs(bp.groundContact.lastContactPositionZ);
   AddVectorObs(bp.groundContact.lastContactNormalX);
   AddVectorObs(bp.groundContact.lastContactNormalY);
   AddVectorObs(bp.groundContact.lastContactNormalZ);

   //height data
   if (bp.groundContact.checkForHeight)
   {
      AddVectorObs(bp.groundContact.GetHeightFromGround());
   }
}

public override void CollectObservations()
{
   base.CollectObservations();

   // Center of mass
   CalculateCenterOfMass();
   AddVectorObs(centerOfMass.x);
```

```
    AddVectorObs(centerOfMass.y);
    AddVectorObs(centerOfMass.z);
}

/// <summary>
/// Overrides the current step reward of the agent and updates the episode
/// reward accordingly.
/// </summary>
/// <param name="reward">The new value of the reward.</param>
public override void SetReward(float reward)
{
    base.SetReward(reward);
    if (adversary != null)
    {
        adversary.SetReward(-reward);
    }
}

/// <summary>
/// Increments the step and episode rewards by the provided value.
/// </summary>
/// <param name="increment">Incremental reward value.</param>
public override void AddReward(float increment)
{
    base.AddReward(increment);
    if (adversary != null)
    {
        adversary.AddReward(-increment);
    }
}

/// <summary>
/// Loop over body parts and reset them to initial conditions.
/// </summary>
public override void AgentReset()
{
    base.AgentReset();

    tester.CollectReward(GetReward());
    tester.CollectCumulativeReward(GetCumulativeReward());
}

/// <summary>
/// Sets the done flag to true for itself and adversary.
/// </summary>
public override void Done()
{
    base.Done();
    if (adversary != null)
    {
        adversary.Done();
    }
}
```

```
}
```

GroundContact2.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

namespace MLAgents
{
    /// <summary>
    /// This class gives the agent information about whether or not this Ground Contact is
touching the ground.
    /// It also stores the last contact position, the normal of the last surface contacted, and the
transform’s height above the ground.
    /// author: Phong Nguyen
    /// with minor code segments from Unity ML-Agents
    /// </summary>
    [DisallowMultipleComponent]
    public class GroundContact2 : GroundContact
    {
        public float lastContactPositionX = 0f; //position of last contact, relative to this transform
        public float lastContactPositionY = 0f;
        public float lastContactPositionZ = 0f;
        public float lastContactNormalX = 0f; //surface normal of last collider that made contact
        public float lastContactNormalY = 0f;
        public float lastContactNormalZ = 0f;
        protected List<Collider> collidedColliders; //colliders that are currently in contact

        public bool checkForHeight = false;
        float spherecastRadius = .15f;
        [SerializeField] Transform spherecastOrigin;
        [SerializeField] LayerMask spherecastLayerMask;
        const float skinWidth = .05f; //distance added to spherecast radius to reduce clipping
        const float MAX_HEIGHT = 500f; //max height spherecast is capable of detecting

        void Start()
        {
            collidedColliders = new List<Collider>();
            Collider col = GetComponent<Collider>();
            if (col is SphereCollider)
            {
                spherecastRadius = ((SphereCollider)col).radius * transform.lossyScale.x -
skinWidth;// - skinWidth;
            }
            else if (col is CapsuleCollider)
            {
                spherecastRadius = ((CapsuleCollider)col).radius * transform.lossyScale.x -
skinWidth;// - skinWidth;
            }
        }
        /// <summary>
        /// Check for collision with ground, and save the last contact position to pass as an
observation
        /// </summary>
        override void OnCollisionEnter(Collision other)
        {
            if (other.transform.CompareTag(Ground))
            {
                touchingGround = true;
                if (!collidedColliders.Contains(other.collider))
                {
                    collidedColliders.Add(other.collider);
                }
                SetLastContactPosition(this.transform.InverseTransformPoint(other.GetContact(0).p
oint));
                SetLastContactNormal(other.GetContact(0).normal);
            }
        }

        /// <summary>
        /// Check for end of ground collision and reset flag appropriately.
        /// Also zeroes out last contact position and contact normal if appropriate.
        /// </summary>
        override void OnCollisionExit(Collision other)
        {
            if (other.transform.CompareTag(Ground))
            {
                if (collidedColliders.Contains(other.collider))
                {
                    collidedColliders.Remove(other.collider);

                    //reset contact point values if no colliders in contact
                    if (collidedColliders.Count == 0)
                    {
                        touchingGround = false;
                        SetLastContactPosition(Vector3.zero);
                        SetLastContactNormal(Vector3.zero);
                    }
                }
            }
        }

        /// <summary>
        /// Update current collision contact point values
        /// </summary>
        void OnCollisionStay(Collision other)
        {
            if (other.transform.CompareTag(Ground))
            {
                SetLastContactPosition(this.transform.InverseTransformPoint(other.GetContact(0).p
oint));
                SetLastContactNormal(other.GetContact(0).normal);
                //visualize surface normal
                //Debug.DrawRay(other.GetContact(0).point, other.GetContact(0).normal * 1f,
Color.red);
            }
        }

        void SetLastContactPosition(Vector3 pos)
        {
            lastContactPositionX = pos.x;
            lastContactPositionY = pos.y;
            lastContactPositionZ = pos.z;
        }

        void SetLastContactNormal(Vector3 n)
        {
            lastContactNormalX = n.x;
            lastContactNormalY = n.y;
            lastContactNormalZ = n.z;
        }

        /// <summary>
        /// Get distance from ground for this ground contact
        /// </summary>
        /// <returns>
        /// Returns the height from the ground if detected, otherwise returns MAX_HEIGHT
        /// </returns>
        public float GetHeightFromGround()
        {
            if (touchingGround)
            {
                return 0f;
            }

            if (spherecastOrigin != null)
            {
                return GetHeightFromGround(spherecastOrigin.position);
            }
            else
            {
                return GetHeightFromGround(transform.position);
            }
        }
        protected float GetHeightFromGround(Vector3 origin)
        {

            RaycastHit hit;
            if (Physics.SphereCast(origin, spherecastRadius, Vector3.down, out hit,
MAX_HEIGHT, spherecastLayerMask))
            {
                return hit.distance;
            }
            else
```

```
            {
                return MAX_HEIGHT;
            }
        }
    }
}
```

GroundGenerator.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

/// <summary>
/// This script randomly generates terrain for the agent to traverse, alternating between flat and
sloped terrain
/// author: Phong Nguyen
/// </summary>
public class GroundGenerator : MonoBehaviour
{
    [SerializeField] Transform groundBase; //reference to prefab
    [SerializeField] Transform target; //reference to game object
    [SerializeField] float minAngle;
    [SerializeField] float maxAngle;
    [SerializeField] float minSegmentLength; //min length of a segment
    [SerializeField] float maxSegmentLength; //max length of a segment
    [SerializeField] float maxLength; //max length of entire ground along X axis

    bool currentlyGeneratingSlope = false;
    float currentLength = 0; //current x distance of total ground
    Vector3 currentPosition = Vector3.zero; //pos of current segment
    float currentAngle; //angle of current segment
    Transform newTransform;
    Vector3 currentScale = Vector3.one; //scale of current segment
    float len;
    float lenX;
    float lenY;

    // Start is called before the first frame update
    void Start()
    {
        if (groundBase == null || target == null)
        {
            //error, these need to be set
            Debug.Log("Error. groundBase or target are not set");
            return;
        }

        currentPosition = transform.position;
        currentPosition.x += -maxLength / 2f;
        // always start with flat for agent to land on
        GenerateStartSegment();
        while (currentLength < maxLength)
        {
            GenerateNextSegment();
        }
        //place target at end
        currentPosition.y += 5f;
        target.position = currentPosition;
    }

    /// <summary>
    /// Generate the initial flat ground segment
    /// </summary>
    void GenerateStartSegment()
    {
        //choose slope angle
        currentAngle = 0;
        newTransform = Instantiate(groundBase, currentPosition, Quaternion.Euler(0, 0,
currentAngle));
        //adjust segment length
        len = 25f;
        lenX = len;
        lenY = 0;
        currentScale = newTransform.localScale;
        currentScale.x = len;
        newTransform.localScale = currentScale;
        currentPosition.x += lenX;
        currentPosition.y += lenY;
        currentLength += lenX;
    }

    /// <summary>
    /// Generate next sloped ground segment
    /// </summary>
    void GenerateNextSegment()
    {
        //choose slope angle
        currentAngle = Random.Range(minAngle, maxAngle);
        newTransform = Instantiate(groundBase, currentPosition, Quaternion.Euler(0, 0,
currentAngle));
        //adjust segment length
        len = Random.Range(minSegmentLength, maxSegmentLength);
        lenX = len * Mathf.Cos(Mathf.Deg2Rad*currentAngle);
        lenY = len * Mathf.Sin(Mathf.Deg2Rad * currentAngle);
        currentScale = newTransform.localScale;
        currentScale.x = len;
        newTransform.localScale = currentScale;
        currentPosition.x += lenX;
        currentPosition.y += lenY;
        currentLength += lenX;
    }
}
```

ProjectileScript.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

/// <summary>
/// This class represents a projectile that can be thrown.
/// A thrown projectile disables itself when its duration ends.
/// Attach this to a prefab with a Collider and Rigidbody.
/// author: Phong Nguyen
/// </summary>
public class ProjectileScript : MonoBehaviour
{
    float duration = 5f; //duration projectile stays active
    float timer = 0f;
    CrawlerAcademy.ProjectileThrower parent;

    void OnEnable()
    {
        //reset timer
        timer = duration;
    }

    public void Init(CrawlerAcademy.ProjectileThrower thrower)
    {
        parent = thrower;
    }

    void Update() {
        //decrement timer, cleanup when time is 0
        timer -= Time.deltaTime;
        if (timer <= 0f)
        {
            if (parent != null)
            {
                parent.CleanupProjectile(gameObject);
            }
        }
    }
}
```

Testable.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

/// <summary>
/// Interface for classes that print test data
/// author: Phong Nguyen
/// </summary>
public interface Testable
{
   void CollectReward(float r);
   void CollectCumulativeReward(float c);
   void ExitPlayMode();
}
```