MACHINE LEARNING ALGORITHMS IN FACILITATING AND ASSISTING ROCKET DESCENT AND LANDING

Ryan Shen [1], Andrew Park [2]

[1] Dos Pueblos High School, 7266 Alameda Ave, Goleta, CA 93117
[2] California State Polytechnic University, Pomona, CA 91768

ABSTRACT

This paper presents a Unity-based reinforcement learning system for simulating rocket descent and landing. Leveraging the Unity ML-Agents framework, our approach applies Proximal Policy Optimization (PPO) combined with imitation learning to balance exploration with guided behavior [8]. Unlike prior works, our system introduces vertical dynamics, randomized initial conditions to reduce overfitting, and variable environmental factors such as gravity, drag, rocket mass, and thruster power. We further refine the reward structure by incorporating precision- and time-based incentives, including a "bullseye bonus" for accuracy and a time bonus for efficiency. Experimental results show that our rocket agents achieve competitive success rates compared to existing implementations, even under more complex conditions. By extending Unity's simulation environment with both technical rigor and user-oriented design, this work contributes to advancing reinforcement learning applications in aerospace while also promoting accessibility and engagement for broader audiences interested in space exploration technologies [9].

KEYWORDS

Unity, Machine Learning, Rockets, Landing

1. INTRODUCTION

Currently, new technologies seem to be developing at a tremendous rate; AI systems have become startlingly capable in the last several years. Alongside such advancements, public fascination with space travel continues to rise. As of 2023, a majority of Americans (~55%) believe that people will routinely travel to space as tourists within the next 50 years (Pew Research Center, 2023). An even larger majority of Americans believe it is essential for the U.S. to be a leader in space exploration, which suggests a receptive audience for space-related learning experiences that are more modern and hands-on [1]. Despite this, classrooms still struggle to convert much of the interest into careers. In a study of 15,847 students, interest in an astronomy career starts comparatively high in middle school but declines at each educational level. For example, interest from male children drops from ~6% in middle school to ~1-2% by college, with even lower figures (3.9% to 0.5%) for females (Bergstrom et al, 2016). However, the same study also found extracurricular activities such as stargazing, tinkering, reading, and watching science predict stronger retention, particularly among girls. A similar paradox is observed in a study with STEM undergraduates, where students report positive views of space exploration but do not have intentions of further engaging through clubs or projects (Pyrkosz-Pacyna et al, 2022) [2]. Meta-analyses across STEM show a higher exam performance when students engage in hands-on learning rather than just listening to lectures (Freeman et al, 2014). Getting more people involved in the space sector is important to pushing forward technological progress for space travel.

David C. Wyld et al. (Eds): MLNLP, ASOFT, CSITY, NWCOM, SIGPRO, AIFZ, ITCCMA – 2025
pp. 207-222, 2025. CS & IT - CSCP 2025 DOI: 10.5121/csit.2025.152016

The three prior methodologies each advanced reinforcement learning for vehicle control but faced notable shortcomings [3]. Yusef Savid et al. applied Unity ML-Agents to race karts, showing PPO with behavioral cloning improved outcomes, though the lack of randomized spawn conditions risked overfitting. Rathod et al. trained PPO-based rocket agents with imitation learning, reaching a 95% success rate; however, their omission of atmospheric effects and reliance on discrete thruster control limited real-world realism. Hicham Bouchana et al. incorporated Thrust Vector Control and Cold Gas Thrusters, yielding 99.6% success, but still assumed discrete engine behaviors not aligned with real rockets. Our project built on these by introducing vertical dynamics, randomized initial conditions, and variable environmental factors such as gravity, drag, and rocket mass [4]. We also refined neural network architecture and enhanced reward shaping with "bullseye" and time-based bonuses. Together, these improvements increased robustness, reduced overfitting, and emphasized both precision and efficiency in rocket landings.

To better leverage the rising interest in space travel, creating and distributing a simulated experience that explores new advancements in rocketry is imperative. Showcasing the intersection of modern AI developments with rocket simulations can direct more attention on pursuing careers and innovation in the space industry [10]. We can use AI and machine learning in order to create reliable and consistent self-landing rockets which the user can in turn directly interact with. The pipeline to produce such models are created in Unity, a 3d game engine, that trains a model controlling a SpaceX Falcon 9 rocket to land on a target platform from variable starting positions and physics parameters. This serves to push forward the viability of AI for the use of landing reusable spacecraft and better understanding the impact of a model's architecture in its capacity to learn and execute such a task. The inclusion of multiple environments and adjustable variables produces a more robust solution compared to similar endeavors where models are only tailored to a more limited set of starting conditions.

In each experiment, we sought to test the effect of manipulating intrinsic and extrinsic variables on the ability of the AI to land. We ran four experiments, each testing one of four variables: gravity, drag, mass, and thrust. Prior to all modifications, we ran a control set of 300 runs using the default Earth-based parameters to get a baseline set of performance metrics. For gravity, we picked the gravities expressed by four celestial bodies: Moon, Mars, Venus, and Jupiter. As for the drag, For the intrinsic values like mass and thrust, each was modified in two increments above and below the control. The increments were 100,000 kg and 1,000,000 N for mass and thrust respectively.

2. CHALLENGES

In order to build the project, a few challenges have been identified as follows.

2.1. Neural Network Architecture Tuning

Determining the neural network's depth and width is deceptively critical. If the policy network is undersized-say, fewer than two hidden layers or fewer than 128 neurons per layer-it lacks the ability to learn complex actions as it doesn't have the representational power to approximate the landing process, causing it to plateau at a mediocre success rate. Oversizing the network, on the other hand, inflates the parameter count, slows down learning, and increases pressure on computer hardware. Worse, changing the architecture afterwards voids all previous training data, forcing a retrain. To help mitigate these issues, we can set up the agent training scene early in the

process and run various configurations of hidden units and layers to settle on the most promising candidate before proceeding to further refinements in the reinforcement step.

2.2. Accelerating PPO Training

While Proximal Policy Optimization (PPO) is relatively stable, it can still require millions of agent steps and hours of computation to result in a high-quality landing strategy [14]. Having too long a wait between steps can hinder iteration progress and put weaker systems under undue stress. To help speed things up, we can increase the number of agents running simultaneously up to what the host system can handle, while also providing player-controlled recordings of the correct actions. These recordings are captured via Unity's demonstration recorder and are enabled by activating "behavioral cloning" and Generative Adversarial Imitation Learning (GAIL) with a roughly 0.6 strength value to kickstart learning [11]. Throughout this training process, various maladaptive behaviours must be struck down with an immediate punishment value and episode termination to limit how far the agents are permitted to drift from the desired actions. We can also alter the simulation's time_scale to further accelerate progress both in iteration and data collection.

2.3. Failure Diagnostics System

Without proper diagnostic tools, failed episodes are opaque, as we simply observe a dip in the reward trends, but not why such a dip is occurring. Failure can range anywhere from crashing too hard into the ground, drifting too far away, rising up, or just missing the target entirely. To help pinpoint the overall failure trend, we can introduce a structured logging system that outputs both to the console and the user interface that counts the occurrences of each type of failure. Each agent reports their results at the end of an episode to the central logger, as identifying what the agents as a whole are missing is more important than what a singular failure happens to be. These layered diagnostics can help guide iteration refinements more effectively.

3. SOLUTION

The system is constructed out of three primary components (Figure 1). The first component is the AI brain, a PyTorch-based neural network managed by Unity ML-Agents that provides control inputs for the targeted rocket [12]. The second component is the environment, which is the world space the rocket exists and operates in, from which we can define and observe the parameter hooks for gravity, drag, mass, and thrust while also providing general intrigue for the user. The third component is the UI/UX, as various interface elements are provided to the end user to allow them to change physics parameters, change the rocket they are viewing, assume manual control, and also observe the performance of the agents in reaction to them.

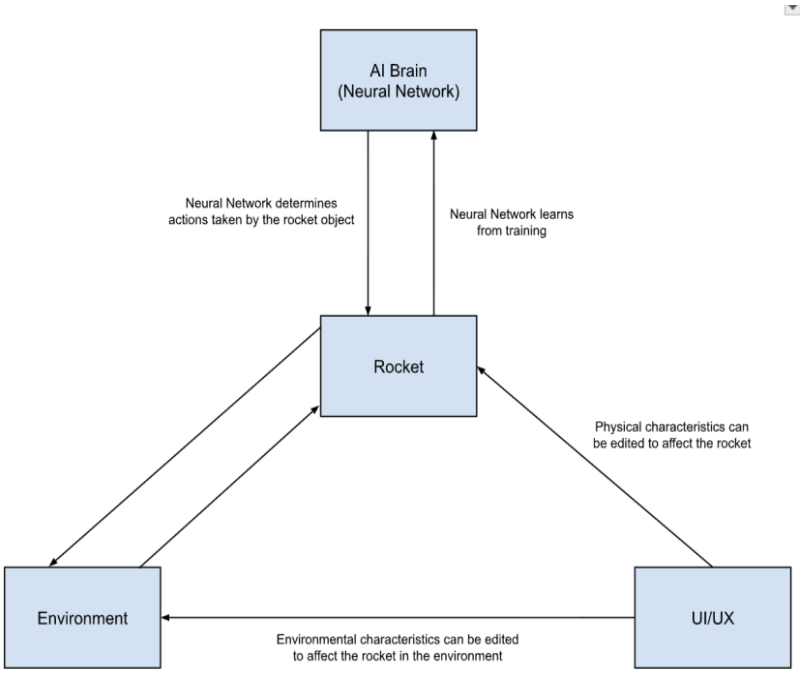


Figure 1. Overview of the solution

The AI brain utilizes PPO as the reinforcement learning algorithm and maps a 12-element observation vector (positions, rotations, destinations, target speed, and current speed) to a discrete action vector. It is implemented with two fully connected hidden layers of 512 neurons each which enables it to better comprehend the complexities of gracefully landing a rocket. The models are trained through a continuous cycle of setup, observations, attempted actions, and end-goal evaluations before then being assigned during inference to try and land the rocket in the correct location and descent velocity.

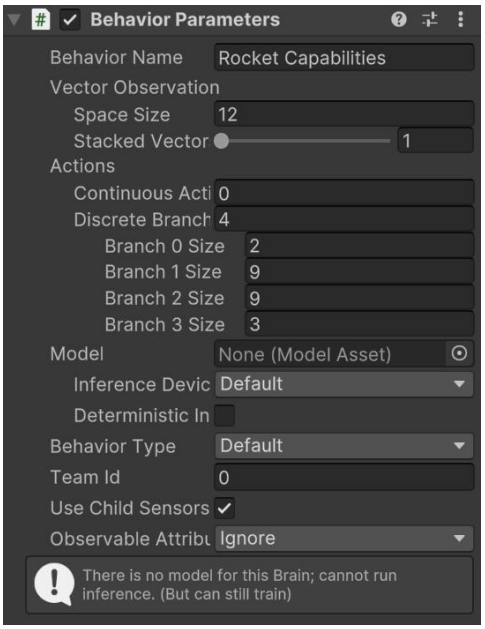
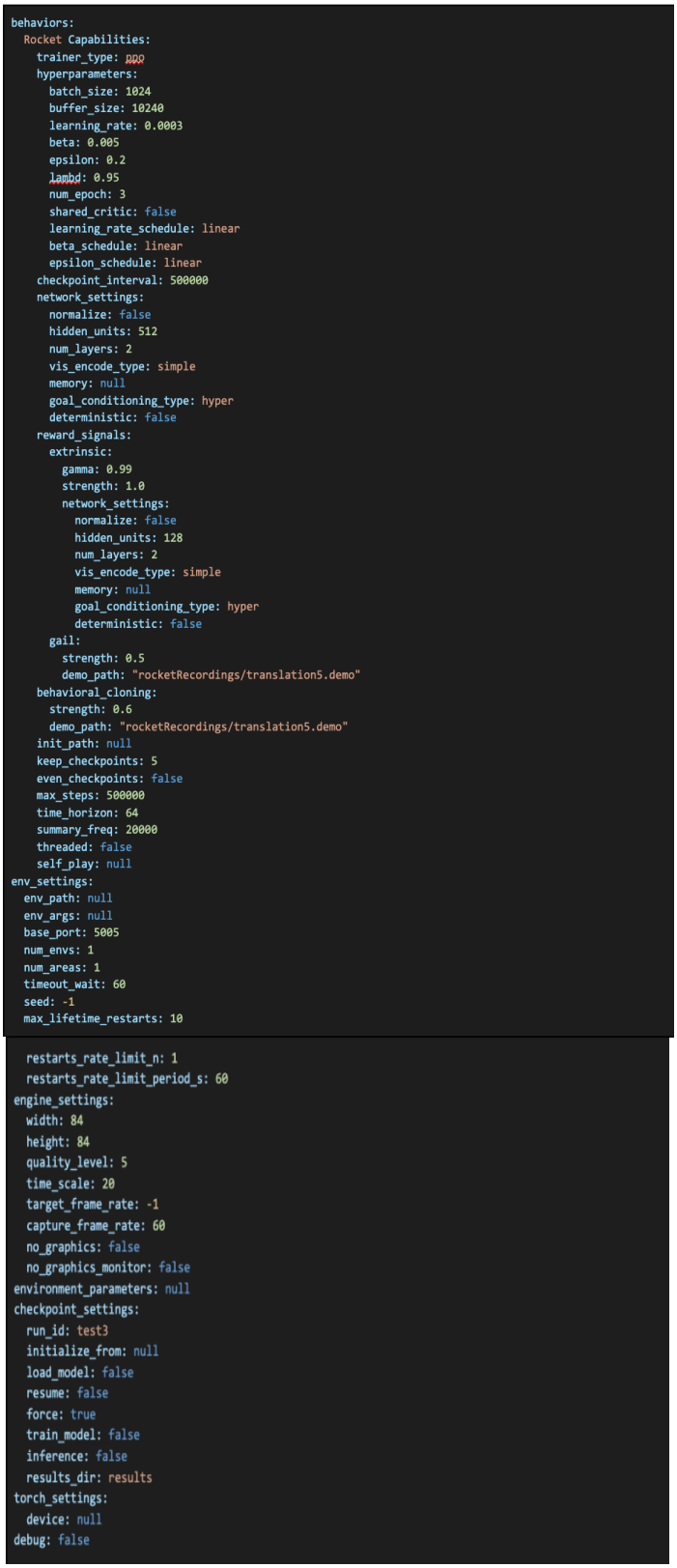


Figure 2. Screenshot of log in page



```
behaviors:
  Rocket Capabilities:
    trainer_type: ppo
    hyperparameters:
      batch_size: 1024
      buffer_size: 10240
      learning_rate: 0.0003
      beta: 0.005
      epsilon: 0.2
      lambd: 0.95
      num_epoch: 3
      shared_critic: false
      learning_rate_schedule: linear
      beta_schedule: linear
      epsilon_schedule: linear
    checkpoint_interval: 500000
    network_settings:
      normalize: false
      hidden_units: 512
      num_layers: 2
      vis_encode_type: simple
      memory: null
      goal_conditioning_type: hyper
      deterministic: false
    reward_signals:
      extrinsic:
        gamma: 0.99
        strength: 1.0
        network_settings:
          normalize: false
          hidden_units: 128
          num_layers: 2
          vis_encode_type: simple
          memory: null
          goal_conditioning_type: hyper
          deterministic: false
      gail:
        strength: 0.5
        demo_path: "rocketRecordings/translation5.demo"
    behavioral_cloning:
      strength: 0.6
      demo_path: "rocketRecordings/translation5.demo"
    init_path: null
    keep_checkpoints: 5
    even_checkpoints: false
    max_steps: 500000
    time_horizon: 64
    summary_freq: 20000
    threaded: false
    self_play: null
env_settings:
  env_path: null
  env_args: null
  base_port: 5005
  num_envs: 1
  num_areas: 1
  timeout_wait: 60
  seed: -1
  max_lifetime_restarts: 10
  restarts_rate_limit_n: 1
  restarts_rate_limit_period_s: 60
engine_settings:
  width: 84
  height: 84
  quality_level: 5
  time_scale: 20
  target_frame_rate: -1
  capture_frame_rate: 60
  no_graphics: false
  no_graphics_monitor: false
environment_parameters: null
checkpoint_settings:
  run_id: test3
  initialize_from: null
  load_model: false
  resume: false
  force: true
  train_model: false
  inference: false
  results_dir: results
torch_settings:
  device: null
debug: false
```

Figure 3. Screenshot of code 1

This YAML configuration is parsed by the `mlagents-learn` trainer at startup. Under `RocketCapabilities`, the `trainer_type` key selects PPO, after which `network_settings` defines a two-layer multilayer perceptron with 512 neurons per layer. `reward_signals` combines the standard discounted-return objective (`extrinsic`) with GAIL, which encourages the agent to imitate the recorded landings provided before the training session. `behavioral_cloning` further stabilizes early learning by minimizing the mean-squared error between the policy and the same demonstrations. To put it another way, GAIL attempts to act similarly to the demo while behavioral cloning seeks to directly copy the example. `max_steps` sets the overall training horizon, while the `time_horizon` (GAE rollout length) and `summary_freq` tune the sampling cadence. Changing any parameter will automatically propagate to the trainer without needing to recompile through Unity, which enables faster and more efficient iteration.

The environment governs the physics and visuals that the player sees and interacts with. It also acts as the interface through which the user can change world parameters like gravity or drag, along with agent parameters like mass or thrust. This enables a variety of dynamic experiments to be conducted in-scene without the need for restarting or recompiling code. The environment also enforces randomized spawn states and episode-terminating events, such as collisions, to help determine if an agent is performing the task correctly.

Figure 4. Screenshot of the system 1

```
public void OnGravityChanged()
    {
        // Change the global gravity value
        if (gravityInputField.text == "")
        {
            Physics.gravity = new Vector3(0, -defaultGravityValue, 0);
            Debug.Log($"Gravity set to: {Physics.gravity}");
        }
        else if (float.TryParse(gravityInputField.text, out float gravityValue))
        {
            Physics.gravity = new Vector3(0, -gravityValue, 0);
            Debug.Log($"Gravity set to: {Physics.gravity}");
        }
        else
        {
            Debug.LogError("Invalid gravity value entered.");
        }
    }

    public void OnDragChanged()
    {
        // change the drag of all rocket rigidbodies
        if (dragInputField.text == "")
        {
            foreach (var env in environments)
            {
                var rockets = env.GetComponentsInChildren<rocket>();
                foreach (var rocket in rockets)
                {
                    var rb = rocket.GetComponent<Rigidbody>();
                    if (rb != null)
                    {
                        rb.linearDamping = defaultDragValue;
                        Debug.Log($"Drag set to: {defaultDragValue} for rocket in {env.name}");
                    }
                }
            }
        }
        else if (float.TryParse(dragInputField.text, out float dragValue)) {
            foreach (var env in environments)
            {
                var rockets = env.GetComponentsInChildren<rocket>();
                foreach (var rocket in rockets)
                {
                    var rb = rocket.GetComponent<Rigidbody>();
                    if (rb != null)
                    {
                        rb.linearDamping = dragValue;
                        Debug.Log($"Drag set to: {dragValue} for rocket in {env.name}");
                    }
                }
            }
        }
    }

    public void OnMassChanged()
    {
        // change the mass of all rocket rigidbodies (int)
        if (massInputField.text == "")
        {
            foreach (var env in environments)
            {
                var rockets = env.GetComponentsInChildren<rocket>();
                foreach (var rocket in rockets)
                {
                    var rb = rocket.GetComponent<Rigidbody>();
                    if (rb != null)

                    {
                        rb.mass = defaultMassValue;
                        Debug.Log($"Mass set to: {defaultMassValue} for rocket in {env.name}");
                    }
                }
            }
        }
        else if (int.TryParse(massInputField.text, out int massValue))
        {
            foreach (var env in environments)
            {
                var rockets = env.GetComponentsInChildren<rocket>();
                foreach (var rocket in rockets)
                {
                    var rb = rocket.GetComponent<Rigidbody>();
                    if (rb != null)
                    {
                        rb.mass = massValue;
                        Debug.Log($"Mass set to: {massValue} for rocket in {env.name}");
                    }
                }
            }
        }
    }

    public void OnThrustChanged()
    {
        // change the mainThrust of all rocket scripts
        if (thrustInputField.text == "")
        {
            foreach (var env in environments)
            {
                var rockets = env.GetComponentsInChildren<rocket>();
                foreach (var rocket in rockets)
                {
                    rocket.mainThrust = defaultThrustValue;
                    Debug.Log($"Thrust set to: {defaultThrustValue} for rocket in {env.name}");
                }
            }
        }
        else if (int.TryParse(thrustInputField.text, out int thrustValue))
        {
            foreach (var env in environments)
            {
                var rockets = env.GetComponentsInChildren<rocket>();
                foreach (var rocket in rockets)
                {
                    rocket.mainThrust = thrustValue;
                    Debug.Log($"Thrust set to: {thrustValue} for rocket in {env.name}");
                }
            }
        }
    }
```

Figure 5. Screenshot of code 2

The script managing the environment contains a series of public physics handlers (`OnGravityChanged`, `OnDragChanged`, `OnMassChanged`, `OnThrustChanged`) that relay UI inputs to either the Unity physics engine or the rocket agents. Each handler is responsible for validating the attempted input or resetting it when the input field is cleared. `Physics.gravity` is a singleton value affecting all dynamic bodies, and changing it allows for the immediate impact to be observed by the environment. For drag, the routine enumerates all the Rigidbody components of active `rocket` instances and assigns them a new drag value. Similar routines are employed for the mass and thrust changes. These hot-swappable parameters feed into the agent's observation vector at the next physics tick and ensure the policy reacts to environmental alterations within one frame. This approach keeps UI, physics, and ML layers loosely coupled, simplifying maintenance [15].

The orbital camera controller and HUD panels round out the overall system experience by transforming otherwise opaque training cycles into a compelling and understandable learning experience. Users can pan, zoom, and observe the environment while optionally assuming control of the rocket agent to boost interactivity and allow the user to compete with the AI. The additional stats panel streams real-time success and failure incidents, which serve to help diagnose issues during the training phase and act as a performance indicator during inference, so users can gauge how well the AI can complete the task.

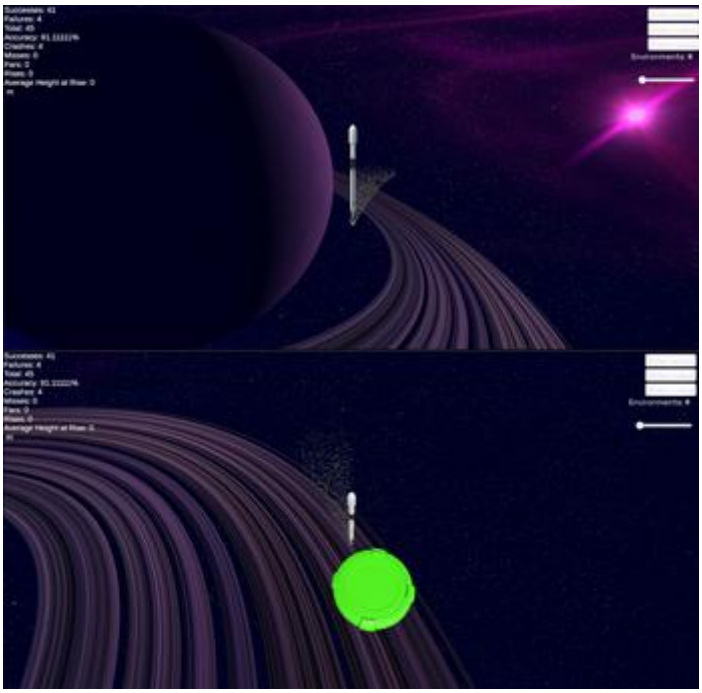


Figure 6. Screenshot of the system 2

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

public class orbitalCamera : MonoBehaviour
{

    public Transform target;
    public Vector3 offset;
    public float zoomSpeed = 12f;
    public float minZoom = 10f;
    public float maxZoom = 300f;
    public float currentZoom = 100f;
    public float xSpeed = 110f;
    public float ySpeed = 110f;

    float x;
    float y;
    Vector3 lastPosition;


    // Initial Camera Values
    Vector3 initialOffset;
    float initialZoom;
    float initialZoomSpeed;
    float initialMinZoom;
    float initialMaxZoom;
    float initialX;
    float initialY;
    float initialXSpeed;
    float initialYSpeed;

    void Awake()
    {
        initialOffset = new Vector3(0f, 35f, 0f);
        initialXSpeed = 110f;
        initialYSpeed = 110f;
        initialZoomSpeed = 12f;
        initialMinZoom = 10f;
        initialMaxZoom = 300f;

        initialZoom = 100f;
        Vector3 angles = transform.eulerAngles;
        initialX = angles.y;
        initialY = angles.x;
    }


    // Start is called once before the first execution of Update after the MonoBehaviour is created
    void Start()
    {
        Vector3 angles = transform.eulerAngles;
        x = angles.y;
        y = angles.x;
    }

    // Update is called once per frame
    void Update()
    {
        if (target == null) {
            return;
        }

        if (Input.GetKeyDown(KeyCode.F)) {
            resetCamera();
        }

        currentZoom -= Input.GetAxis("Mouse ScrollWheel") * zoomSpeed;
        Mathf.Clamp(currentZoom, minZoom, maxZoom);

        if (Input.GetMouseButton(0)) {
            x += Input.GetAxis("Mouse X") * xSpeed * 0.02f;
            y -= Input.GetAxis("Mouse Y") * ySpeed * 0.02f;
        }

        Quaternion rotation = Quaternion.Euler(y, x, 0);
        Vector3 position = rotation * new Vector3(0, 0, -currentZoom) + target.position + offset;

        transform.rotation = rotation;
        transform.position = position;
    }

    void resetCamera()
    {
        x = initialX;
        y = initialY;
        currentZoom = initialZoom;
        xSpeed = initialXSpeed;
        ySpeed = initialYSpeed;
        zoomSpeed = initialZoomSpeed;
        minZoom = initialMinZoom;
        maxZoom = initialMaxZoom;
        offset = initialOffset;

        Quaternion rotation = Quaternion.Euler(y, x, 0);
        Vector3 position = rotation * new Vector3(0, 0, -currentZoom) + target.position + offset;

        transform.rotation = rotation;
        transform.position = position;
    }
}
```

Figure 7. Screenshot of code 2

`Update()` executes once per rendered frame. The loop first checks if there is a valid target to observe and prevents the remainder of the code from running if there is not one. Then, it proceeds to check if the user pressed `F` to reset the camera before proceeding to the general control logic. The value assigned to the current zoom is attached to the scroll wheel input within predefined

bounds. The left-mouse drag is captured to rotate the spherical coordinates stored in the x (azimuth) and y (elevation) variables. The `Quaternion.Euler` method converts these angles into a rotation, then multiplies it by a negative-z vector derived from the current zoom value to obtain the camera offset before being applied to the target's world position. Given that this script runs independently of the ML-loop, it imposes zero overhead on training speed, even when `time_scale` is elevated.

4. EXPERIMENT

4.1. Experiment 1

The first element of the system we need to test is the impact of extrinsic factors from the environment, such as gravity and drag, on the Agent's accuracy.
As independent variables, gravity and drug must each be isolated to their own set of tests to avoid interference from one another. For each set, Earth-like parameters that have been used during the entire training phase are used as the control, while notable values higher and lower are chosen to be tested and compared against the control. Each value is tested against 300 runs to filter outliers and get a reasonable average.

For gravity, we selected values based on several different celestial bodies in the Solar System. On the lower end we have the Moon, followed by the planets Mars and Venus, which are closer to the control of gravity. As an extra bonus, we also included the gravity of Jupiter to observe how the system reacts to a dramatically larger stressor.

Drag values were selected in intervals of 0.001 with the exception of 0, 0.01, and 0.05 in order to observe more drastic changes as well.

Control Results:

Successes: 278, Failures: 22, Total: 300, Accuracy: 92.66666%
Crashes: 15, Misses: 7, Fars: 0, Rises: 0, Average Height at Rise: 0 m
Gravity values: control = -9.81 m/s^2,
moon = -1.625 m/s^2:
No run complete, however, the rockets all sit suspended a few inches above the platform, suggesting that they would succeed, but at the expense of using up all theoretical fuel before doing so.
mars = -3.728 m/s^2:
Successes: 300, Failures: 0, Total: 300, Accuracy: 100%
Crashes: 0, Misses: 0, Fars: 0, Rises: 0, Average Height at Rise: 0 m
venus = -8.87 m/s^2:
Successes: 290, Failures: 10, Total: 300, Accuracy: 96.66666%
Crashes: 4, Misses: 6, Fars: 0, Rises: 0, Average Height at Rise: 0 m
jupiter = -24.79 m/s^2:
Successes: 84, Failures: 216, Total: 300, Accuracy: 28%
Crashes: 79, Misses: 137, Fars: 0, Rises: 0, Average Height at Rise: 0 m
Drag: control = 0.001293 (density of air at sea level in kg/m³), other values:
0:
Successes: 274, Failures: 26, Total: 300, Accuracy: 91.33334%
Crashes: 15, Misses: 11, Fars: 0, Rises: 0, Average Height at Rise: 0 m
0.000293:
Successes: 275, Failures: 25, Total: 300, Accuracy: 91.66666%

Crashes: 14, Misses: 11, Fars: 0, Rises: 0, Average Height at Rise: 0 m
0.002293:
Successes: 281, Failures: 19, Total: 300, Accuracy: 93.66666%
Crashes: 8, Misses: 11, Fars: 0, Rises: 0, Average Height at Rise: 0 m
0.003293:
Successes: 278, Failures: 22, Total: 300, Accuracy: 92.66666%
Crashes: 11, Misses: 11, Fars: 0, Rises: 0, Average Height at Rise: 0 m
0.01:
Successes: 286, Failures: 14, Total: 300, Accuracy: 95.33334%
Crashes: 8, Misses: 6, Fars: 0, Rises: 0, Average Height at Rise: 0 m
0.05:
Successes: 300, Failures: 0, Total: 300, Accuracy: 100%
Crashes: 0, Misses: 0, Fars: 0, Rises: 0, Average Height at Rise: 0 m
Highest accuracy, but likely excessive fuel consumption

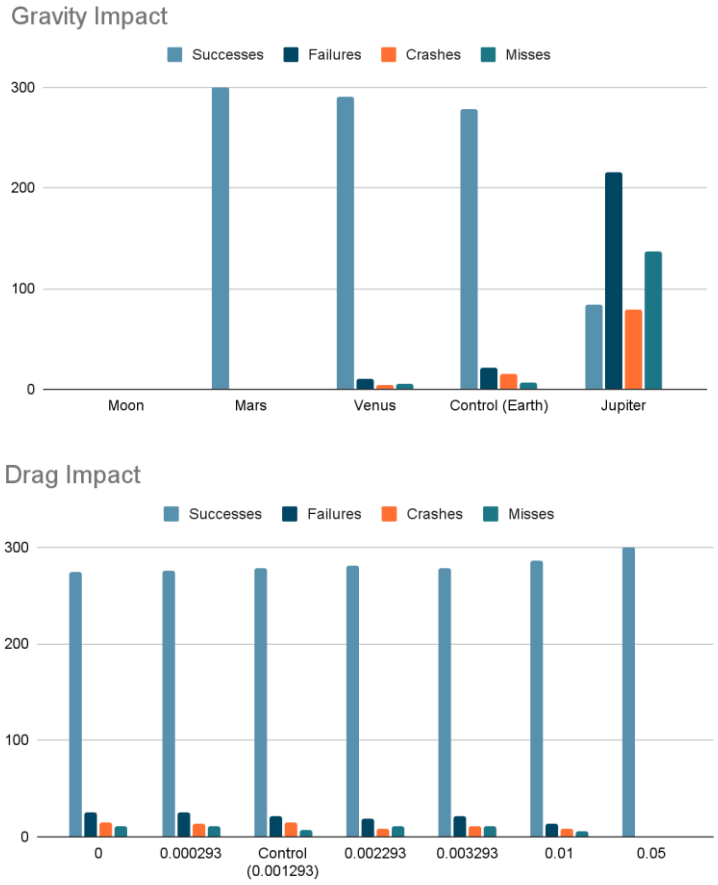


Figure 8. Figure of experiment 1

Generally speaking, the performance of the rocket agents were better, the lower the gravity likely due to easier maneuverability as less force was required to make the necessary adjustments. Compared to the control success rate of 92.67%, in Venus it rose to 96.67% and then achieved a 100% success rate for Mars. However, a weird behavior was observed once we got to Moon gravity as no run was able to complete at all; the rocket would instead just perpetually hover a couple inches above the platform. This is likely due to the amount of force the agent exerted through the thrusters being too strong relative to the gravitational force thus preventing it from

making progress towards the ground. On the other hand, when testing a gravity value higher than Earth, we observed a significant drop in the performance. When simulating Jupiter's gravity, the rocket was only able to succeed 28% of the time with around ⅔ of the failures being attributed to "misses" and the remainder to "crashes." Misses are defined as when the rocket reaches ground level but fails to touch the target platform. The fact that the majority of failures were in this category indicates that the gravity was too strong for the lateral thrusters to have enough time to move the rocket into position. Crashes are when the rocket does manage to touch the platform but is still doing so at a velocity that would incur damage. In the case of drag, the general accuracy trended higher as more drag was introduced to the system. The implication here is that the rocket became less prone to drifting, allowing for the rocket to make more precise adjustments. However, it should be noted that higher drag does force more time and fuel expenditure required to land. Between the two extrinsic variables, changes in gravity were much more significant.

4.2. Experiment 2

Another group of elements we tested for the system are intrinsic properties, such as different rocket masses and primary thruster power.

Much like the previous experiment, the rocket mass and thrust must also each be isolated to properly measure their impact. The same control results are able to be used for this set of scenarios. Two values in either direction are tested to the control value are tested. We continue to test each value against 300 runs to filter outliers and get a reasonable average.

For mass, increments of 100,000 kg were explored in either direction while thruster force values were changed in increments of 1,000,000 to observe the optimal ranges.

Control Results:

Successes: 278, Failures: 22, Total: 300, Accuracy: 92.66666%
Crashes: 15, Misses: 7, Fars: 0, Rises: 0, Average Height at Rise: 0 m
Masses: control = 549054 kg, two increments of 100,000 kg in either direction
349054 kg:
Successes: 300, Failures: 0, Total: 300, Accuracy: 100%
Crashes: 0, Misses: 0, Fars: 0, Rises: 0, Average Height at Rise: 0 m
449054 kg:
Successes: 300, Failures: 0, Total: 300, Accuracy: 100%
Crashes: 0, Misses: 0, Fars: 0, Rises: 0, Average Height at Rise: 0 m
649054 kg:
Successes: 248, Failures: 52, Total: 300, Accuracy: 82.66666%
Crashes: 38, Misses: 14, Fars: 0, Rises: 0, Average Height at Rise: 0 m
749054 kg:
Successes: 195, Failures: 105, Total: 300, Accuracy: 65%
Crashes: 74, Misses: 31, Fars: 0, Rises: 0, Average Height at Rise: 0 m
Thrust: control = 10,000,000 two increments of a million newtons in either direction
8,000,000 N:
Successes: 234, Failures: 66, Total: 300, Accuracy: 78%
Crashes: 37, Misses: 29, Fars: 0, Rises: 0, Average Height at Rise: 0 m
9,000,000 N:
Successes: 253, Failures: 47, Total: 300, Accuracy: 84.33334%
Crashes: 27, Misses: 20, Fars: 0, Rises: 0, Average Height at Rise: 0 m
11,000,000 N:
Successes: 292, Failures: 8, Total: 300, Accuracy: 97.33334%

Crashes: 7, Misses: 1, Fars: 0, Rises: 0, Average Height at Rise: 0 m
12,000,000 N:
Successes: 300, Failures: 0, Total: 300, Accuracy: 100%
Crashes: 0, Misses: 0, Fars: 0, Rises: 0, Average Height at Rise: 0 m

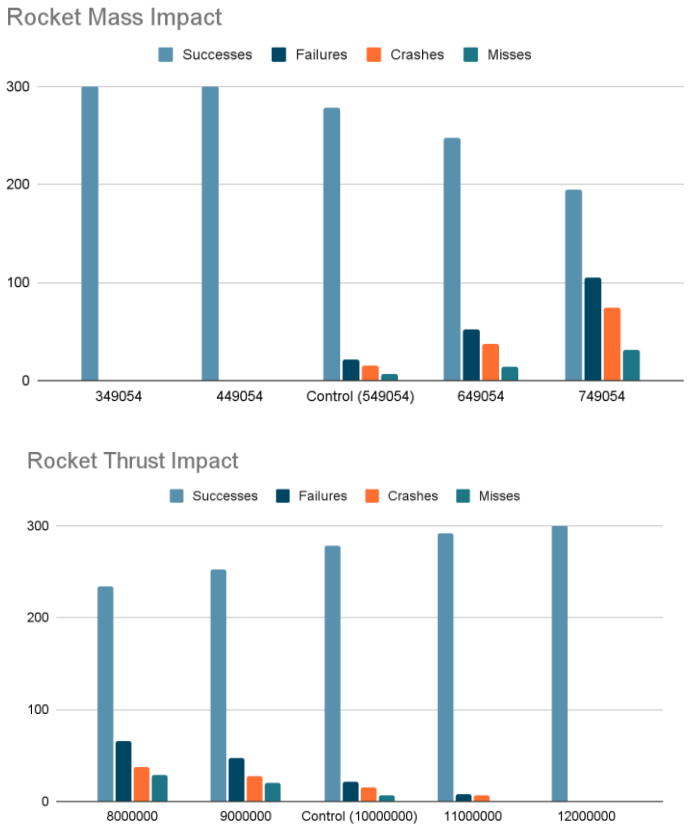


Figure 9. Figure of experiment 2

Both the mass and thruster tests yielded a clear trend in the general performance of the rocket agents. In the case of mass, accuracy only started to suffer once the rocket started to get heavier than the control weight, dropping from 92.67% success to 82.67% and 65% at each larger increment respectively. On the flipside, there was a consistent increase in success rising to 100% in the lower mass increments.This is a good sign, as generally rockets do not increase in weight throughout a flight. The lower mass means the rocket has less inertia to overcome. As for the thrust, the trend went in the opposite direction. When it continued to increase, the rocket got progressively more accurate, which is likely due to tighter control over drifting afforded to the rocket by the stronger thruster. However, at higher values, the rocket would have to be switching between on/off incredibly fast, which is impractical in real real-life applications. Both trends coincide with one another as decreasing the mass or increasing the thruster power, both result in a rocket that is easier to maneuver.

5. Related Work

Yusef Savid et al. also put together a system under the Unity ML-Agents framework, but to teach agents how to drive race karts around a track [5]. Their implementation achieved a reasonable performance, hitting a mean reward of 0.761 without obstacles and 0.0681 with. While there is a

significant drop in the mean reward upon the introduction of foreign obstacles, the fact that adding behavioral cloning as a pre-training option allowed the agents to still achieve a positive mean reward, which aligns with the effectiveness we observed when providing the same references to our own agents. Their conclusion that Proximal Policy Optimization, particularly with behavioral cloning, provided the best results matches the evolution we observed in the rocket agents. One potential risk acknowledged by the authors is the current lack of randomization in the initial spawn position and orientation of the car, which could lead to overfitting issues. Our rocket agents expand on the application of using Unity's ML agent system to drive vehicles by introducing a vertical axis to the environment and also implementing a range of initial starting conditions to better avoid the risk of overfitting.

The simulation created by Rathod, A. et al. bore a much more similar resemblance to the system we designed as both are focused on teaching PPO-based rocket agents within the Unity engine how to safely land [6]. Their implementation achieved a very high success rate of 95% which was heavily boosted by the presence of curriculum and imitation learning, further corroborating the effectiveness of PPO-based solutions backed up with human references. With that being said, the lack of atmospheric effects and an unrealistic assumption of engines being able to instantly turn on and off via a discrete control system means that further testing is required before any real-world deployment. While our system shares the same control limitations and works under a smaller elevation range, we expand on the concept through the introduction of variable environmental factors such as gravity and drag in addition to intrinsic variables like the rocket's mass and thruster power. We were also able to squeeze out a similar success rate in spite of these conditions by also leveraging some tweaks to the model architecture such as tuning up the size of each layer of the neural network as well as the number of layers. From a thematic standpoint, we also placed a greater emphasis on visual elements like the skybox which would normally be superfluous but serves to provide a more polished experience as the program is also intended to be directly downloaded by others and interacted with to promote accessibility and engagement with space media.

Hicham Bouchana et al. also approaches the task of teaching Unity agents to land a rocket, but with the added component of Thrust Vector Control (TVC) to augment maneuverability [7]. They also incorporate Cold Gas Thrusters (CGT), which are essentially side thrusters located near the first stage of the rocket much like the ones we integrated into our own agents to facilitate lateral movement and stability. Their solution achieved a very high success rate of 99.6% which mirrors the high success rates observed across the board for PPO-based agent solutions. This system, however, also has the limitation of assuming discrete controls which does not fully translate to real life conditions at this time. Our solution expands on the incentive structure of the rocket agent to more explicitly factor in landing time and precision into the rewards the agents can achieve; progress reward formulas were tuned into the desired curve. Figure 10 illustrates the "bullseye bonus," which is the exponential reward given the closer the rocket is to the center of the platform upon a successful landing. Figure 11 is the additional time bonus provided based on the amount of time it took to complete the operation. Higher starting points were tuned to give a slightly higher reward yield to further push the agents to prefer faster runs. Given that we are operating within the 300-700 meter elevation band, the curve was structured to only give a meaningful reward within about 70 seconds.

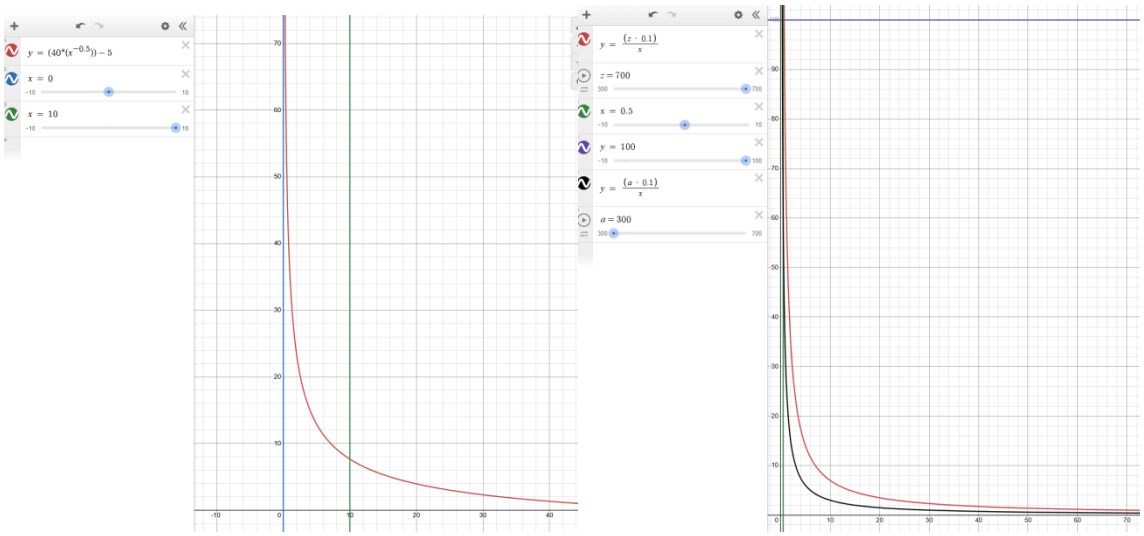


Figure 10. Figure of paper 3

6. CONCLUSIONS

The general scope of the AI model's utility still has significant room to grow. While the rocket is capable of performing spins and rotations, the agents themselves were not trained enough to properly take advantage of these controls which limit the conditions it can adapt to. The current agent's focus is on the final vertical landing phase but can also be expanded to perform the earlier stages of descent starting at reentry to form a more complete solution. This would require a significant increase to the starting height at around 4.6 km for the final vertical landing phase and a separate model to handle the reentry phase which begins at around 60-70 km. The increased difficulty of such an expansion can be improved through the introduction of more adaptable controls like TVC.

Machine learning algorithms play a pivotal role in enabling safe and precise rocket descent and landing by optimizing trajectory control, predicting dynamic conditions, and adapting in real time. Their integration enhances reliability, reduces risk, and supports sustainable space exploration, marking a critical advancement in aerospace engineering and autonomous flight systems.

REFERENCES

[1] Savid, Yusef, et al. "Simulated autonomous driving using reinforcement learning: A comparative study on unity's ML-agents framework." Information 14.5 (2023): 290.

[2] Xue, Shuai, et al. "Research on intelligent control method of launch vehicle landing based on deep reinforcement learning." Mathematics 11.20 (2023): 4276.

[3] Bouchana[1], Hicham, et al. "AI-Guided Rocket Landing: Navigating Precision Descent Strategies." Proceedings of the International Conference on Emerging Intelligent Systems for Sustainable Development (ICEIS 2024). Vol. 184. Springer Nature, 2024.

[4] Kennedy, Brian, and Alec Tyson. "Americans' views of space: US role, NASA priorities and impact of private companies." (2023).

[5] Bergstrom, Zoey, Philip Sadler, and Gerhard Sonnert. "Evolution and Persistence of Students' Astronomy Career Interests: A Gender Study." Journal of Astronomy & Earth Sciences Education 3.1 (2016): 77-92.

[6] Pyrkosz-Pacyna, Joanna, Katarzyna Cieślak, and Marcin Zwierżdżyński. "Space Exploration Perception and Engagement Intention Among STEM University Students–Results from a Mixed Method Study." The New Educational Review 69.3 (2022): 191-205.

[7] Freeman, Scott, et al. "Active learning increases student performance in science, engineering, and mathematics." Proceedings of the national academy of sciences 111.23 (2014): 8410-8415.

[8] Wang, Yuhui, Hao He, and Xiaoyang Tan. "Truly proximal policy optimization." Uncertainty in artificial intelligence. PMLR, 2020.

[9] Yang, Chi-Wen, et al. "Unity 3D production and environmental perception vehicle simulation platform." 2016 International Conference on Advanced Materials for Science and Engineering (ICAMSE). IEEE, 2016.

[10] McLennan, Stuart, et al. "An embedded ethics approach for AI development." Nature Machine Intelligence 2.9 (2020): 488-490.

[11] Ho, Jonathan, and Stefano Ermon. "Generative adversarial imitation learning." Advances in neural information processing systems 29 (2016).

[12] Hu, Benyi, et al. "PyRetri: A PyTorch-based library for unsupervised image retrieval by Deep Convolutional Neural Networks." Proceedings of the 28th ACM International Conference on Multimedia. 2020.

[13] Raut, Umesh, et al. "Unity ML-Agents: Revolutionizing Gaming Through Reinforcement Learning." 2024 2nd World Conference on Communication & Computing (WCONF). IEEE, 2024.

[14] Wu, Zifan, et al. "Coordinated proximal policy optimization." Advances in Neural Information Processing Systems 34 (2021): 26437-26448.

[15] Jordan, Michael I., and Tom M. Mitchell. "Machine learning: Trends, perspectives, and prospects." Science 349.6245 (2015): 255-260.