Capstone Project Report

By Nathaniel Biddle

Capstone: Exploration Of Unity As A Physics Simulator And Unity ML-Agents As A Reinforcement Learning Sandbox, Used To Train A Quadcopter To Autonomously Navigate An Obstacle Course

Abstract

In this solo capstone project, I explored the capability of Unity Game Engine as a robotics simulation environment and trained a quadcopter within the simulation to autonomously navigate one or more obstacle courses, compared and contrasted results based on the Proximal Policy Optimization (PPO) deep reinforcement learning algorithm with and without Generative Adversarial Imitation Learning (GAIL) and Behavioral Cloning (BC). I tested multiple reward methodologies against multiple paradigms of perceptive capability for the robot, testing location knowledge of obstacles and the robot's own location within the simulation, as well as utilizing raycasting as a method of obstacle detection, and analyze outcomes in different scenarios. The goal was to explore the efficacy of the Unity ML-Agents package and Unity as a simulation environment while attempting to train the quadcopter to effectively navigate, to model the dynamics that controlled the robot via thrust commands (using a continuous action space), and to create an effective training paradigm to achieve that. Much was learned in this capstone project about the Unity ML-Agents package and deep reinforcement learning general, and the model was fairly performant, but robustness can be increased with greater training time or utilization of greater compute. The conclusion is that Unity is a useful simulator for physics and for reinforcement learning training and that continuous action spaces require large compute or large amounts of time in order to build robust behaviors for a neural net.

Introduction

Various simulation environments are widely used for training robots in a cost-effective and scalable way, such as MujoCo, ROS Gazebo, and Nvidia Isaac Sim. Increasingly so too is the Unity game engine, which utilizes Nvidia PhysX engine as the default means of simulating physics behavior. In the context of robotics, Unity makes available the ML Agents plugin, which allows you to integrate with your local TensorFlow or OpenAI Gym environment in order to train behavior within Unity. The exploration of Unity as an engine within which to train robots has the potential to make robotics more accessible to those that have not done robotics work, but have experience with Unity, for example. Quadcopters are a type of 4-propellered drone and I will attempt to train the quadcopter to autonomously navigate an obstacle course utilizing the ML Agents Unity package. Autonomous Drones are increasingly becoming

mainstream in both defense sectors and consumer commercial sectors. Defense sector (including commercial defense companies) entrants into the space include companies such as Anduril and consumer commercial companies include companies such as Archer Aviation. In the future, there may be drone deliveries and there will certainly be various autonomous agents roaming the very cities we live in as time goes on.

For my capstone project for my Master of Engineering in Robotics and Intelligent Autonomous Systems, I chose to simulate the dynamics and control of an autonomous drone within Unity to capstone my experience in learning about the use of these areas of study in creating autonomous systems. I utilized the Proximal Policy Optimization algorithm, which originally came from OpenAI in 2017, a new type of policy gradient method for reinforcement learning (John Schulman, 2017). I set up the simulation and chose the reward function and hyperparameters through a mix of trial and error and intuitive judgment based on considerations of behavioral outcomes based on chosen reward conditions. This was used in training the autonomous drone to navigate from a start line to a finish line with randomized obstacles spawning per training episode that the drone attempted to avoid.

Software And Environment Setup

Unity ML Agents is a package for the Unity Game Engine that allows you to utilize TensorFlow, a machine learning platform, and run deep reinforcement learning algorithms within a simulated environment. It uses the Nvidia PhysX Physics Engine to simulate physics interactions. Setting up the environment for training the drone consisted of installing the appropriate Unity version (2019.3.xf), installing the ML Agents Unity Package, TensorFlow, Conda for environment management (it allows siloed python versions and other required libraries), and several other associated libraries.

Quadcopter Dynamics Modeling

The dynamics of the quadcopter will be modeled, allowing the quadcopter to autonomously navigate utilizing continuous thrust commands. Modeling the dynamics is done to better approximate the real world, increasing the applicability of the training to the real world.

For the purposes of the control system used in this reinforcement learning project, we apply direct thrust vertically, horizontally, and along the depth axis, each of which correspond respectively to the following (Sharma, 2024) (note that within Unity, the X Axis is horizontal, the Y Axis is vertical, and the Z Axis represents depth):

- Vertically/Y Axis (Throttle): This simulates equivalent increases in RPM of all 4 motors of the quadcopter to overcome the force of gravity and rise in the case of rising vertically, and

equivalent decreases of RPM below the threshold which equals the force of gravity in order to lower vertically.
    - Mathematically, this can be represented as:
        - $F_{\text{total}} = F_1 + F_2 + F_3 + F_4 = m(g + a_y)$
        - Where F_i is is the upward force generated by the rotational motion of one of the motors, g is the force of gravity, and a_y is the desired vertical acceleration upward or downward.
        - The Hovering condition occurs when a_y is set to 0.
- Horizontally/X Axis (Roll): This simulates equivalent increases in both the left and right 2 motors respectively, with increases in RPM of the left 2 motors relative to the right causing movement horizontally to the right and increases in RPM of the right 2 motors relative to the left causing movement horizontally to the left.
    - Mathematically, this can be represented as:
        - $\Delta F_{\text{roll}} = (F_1 + F_4) - (F_2 + F_3)$
        - $a_x = \frac{\Delta F_{\text{roll}}}{m}$
        - Where a_x is the translational acceleration along the x-axis left or right
- Depth/Z Axis (Pitch): This simulates equivalent increases in the back 2 or front 2 motors, with RPM increases of the front 2 motors relative to the back 2 causing movement backward and increases in the back motors relative to the front causing movement forward.
    - Mathematically, this can be represented as:
        - $\Delta F_{\text{pitch}} = (F_3 + F_4) - (F_1 + F_2)$
        - $a_z = \frac{\Delta F_{\text{pitch}}}{m}$
        - Where a_z is the translational acceleration along the z-axis forward or backward

In Unity, this was done utilizing the RigidBody component (which represents within the physics simulation a body which respects the laws of rigid body dynamics) and its method Rigidbody.AddForce().

Training Of Autonomous Drone

Unity ML-Agents allows for creation of and iteration on reward functions for your autonomous agents. Reward paradigms, algorithm choice, and level of exposure to environmental information are aspects of a training paradigm that can affect training success, reliability, and/or speed. Obstacle course scene characteristics in Unity will be defined and then trained within. Multiple episodes of training and multiple reward functions often must be tested to find an effective reward function, utilizing both intuition and best practice. Through effective

reward functions, over many episodes, the autonomous drone will be trained to navigate the obstacle course.

Several methodologies were utilized in order to effectively train the autonomous agent to navigate the obstacle course. I began with trying pure reinforcement learning using the PPO algorithm, prior to any use of behavioral cloning methodologies. Through 7,000,000 time steps, this was fruitful to some degree, but did not converge on the desired behavior, so I introduced curriculum learning, behavioral cloning, and generative adversarial imitation learning in order to increase training speed for my available compute power.

Training Observations

Unity ML-Agents handles observations via a provided method of the built-in 'sensor' object. The method is called 'AddObservation()' and you can pass into it any vector or float value. A vector is treated as the number of values equal to the dimensions of the vector, so a Vector3 is treated as 3 separate observations; floats are treated as singular float values. At each time step after an action has been carried out by the autonomous agent, observations are collected again, with the control and training process consisting of 'Observation'>'Action'>'Reward'>'Observation'>…, so on and so forth until training is concluded. The observations are mapped to outcomes of actions to refine the neural networks mapping of input observations to outputted actions. Observations included the drone's velocity, position, as well as a set of raycast observations which mapped distance to intersection with the raycast relative to the autonomous agent, as defined in the DroneAgent.cs class.

Continuous Action Space

The training paradigm that I used in this project utilized a continuous action space, allowing the autonomous agent to carry out actions that apply a random value (that gains rules over time) between -1.0 and 1.0, in the case of the quadcopter agent simulation, in all 3 axes by providing thrust in those directions. There are pros and cons to this approach. One of the main benefits of using a continuous action space is that it allows for more fine-grained behavioral outcomes due to the greater granularity of movement. A downside, however, is that as a result of the large range of values that can be applied, training can take orders of magnitude longer than it would in a discrete action space, a discrete action space only allowing for some specific set of values from a list/array. This led to a wanting robustness in obstacle avoidance in the current result.

Reward Function

Initially, I attempted to add undue complexity to the reward function. My first attempts included the following:

- A negative penalty for moving backward past a specific point in the z-axis
- A negative penalty for any collision
- A negative penalty constantly and scaled based on proximity to objects that have colliders, calculated based on raycasts emanating from the autonomous drone agent
- A constant, per time step, negative penalty, to penalize not moving
- A reward for proximity to the finish line z-axis barrier
- A reward for reaching the finish line without collisions

This led to some unexpected behaviors. The ratios of negative rewards to positive rewards caused conflicts leading to the autonomous agent learning to crash as soon as possible, as that optimized the least negative reward, never allowing it to move forward with enough consistency to get the larger award that it would get as it approached the finish line. In order to reduce complexity and get a more straightforward and reliable result, as well as increase training speed, I removed a few portions of the reward function, leaving me with:

- A reward for forward progress and penalty for backward progress, as measured by a "previous" and "currentZ" variable
- A reward for reaching the finish line without collisions
- A penalty for several boundary violations (in x, y, and z) at specific positions in meters (the unit in Unity)
- A penalty for a collision

The result of this was that there was no outcome that would maximize reward except by navigating forward toward the finish line.

The reward function that was utilized in the final paradigm is mathematically representable as:

- Progress Reward (pos/neg):

$$R_{\text{progress}} = progress_{var}$$

- Goal Reward

$$R_{\text{goal}} = 5000 \quad \text{if } z <= z_{\text{goal}}$$

- Early Retreat Penalty

$$R_{\text{boundary}} = -50 \quad \text{if } z > 2$$

- Collision And Boundary Penalty

$$R_{\text{collision}} = -50$$

$$R_{\text{boundary}} = -50 \quad \text{if } -15 < x < 15$$

$$R_{\text{boundary}} = -50 \quad \text{if } -1.75 < y < 20$$

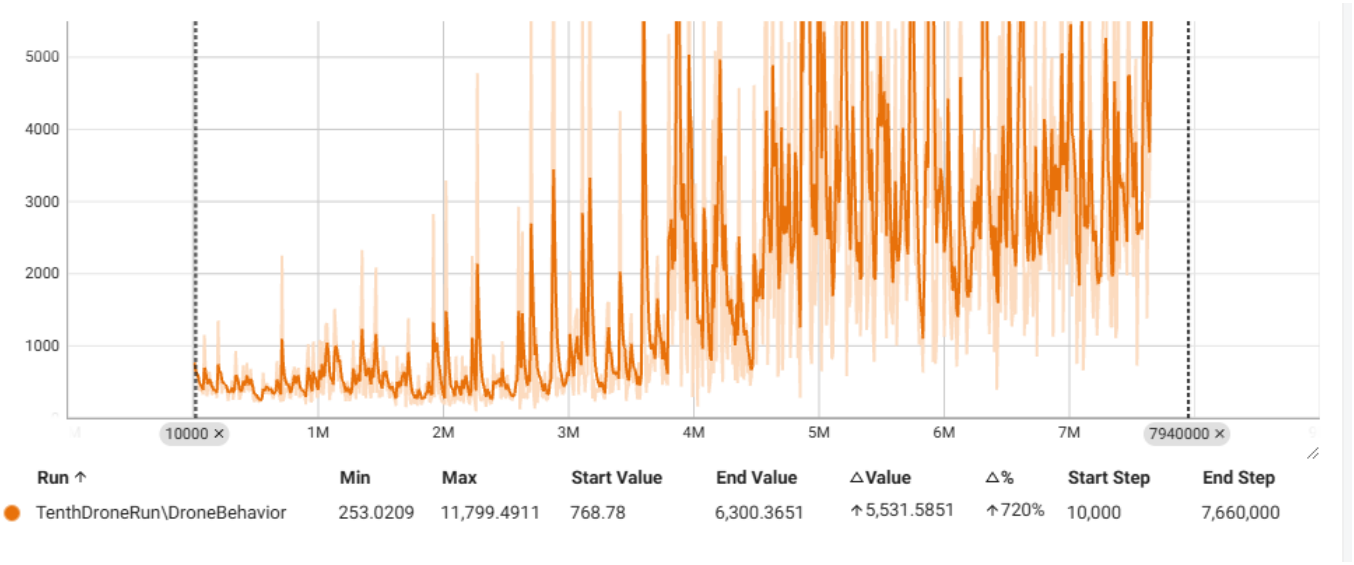
- Total Reward:

$$R_{\text{total}} = R_{\text{progress}} + R_{\text{goal}} + R_{\text{boundary}} + R_{\text{collision}}$$

And on collisions or early retreats, the episode is ended via Unity ML-Agent's provided EndEpisode() method, which implicitly drives the agent to avoid that behavior due to the fact that reward is maximized by the episode not ending and the agent reaching the goal.

Curriculum Learning

Early methodology that attempted to both reach the finish line and avoid obstacles were not quickly fruitful and led to more unexpected behaviors, so a curriculum learning approach was adopted. The drone agent was first trained, instead, on an empty course, where the only crash points were the floor, ceiling, and walls. This allowed the agent to learn to move toward the finish line in the z-axis while maintaining appropriate hover and sideways movement. I initially trained the agent to overshoot the desired z-axis position, so that it had a policy that moved toward the end goal consistently. A neural net model was generated for that behavior.



Run ↑	Min	Max	Start Value	End Value	△Value	△%	Start Step	End Step
TenthDroneRun\DroneBehavior	253.0209	11,799.4911	768.78	6,300.3651	↑5,531.5851	↑720%	10,000	7,660,000

The above graph shows the training of the neural net for the agent in the first step of the curriculum, training it to simply navigate an empty course. This was intended to build a foundation from which future learning could adapt. I trained it several different times, previous attempts utilizing the less straightforward reward function described in the Reward section above, with this result being the best base result for the first behavior. This ran for 7.66 million steps and you can see that the reward increased over time, with a maximum of 11,799 (for this earlier paradigm, there were increasing rewards as you got closer to the z-axis target position), though some instability remained. Some instability is common in reinforcement learning and can be further reduced through additional training cycles and fine-tuning of the reward function and/or hyperparameters, something to be considered for future projects.

Once the ability to move forward toward the finish line was trained, the next part was to add in the cubes. I utilized a "CubeSpawner.cs" script in order to achieve this, which randomly generated cubes of random volumes within a specified range, as well as in random locations within a specified range within the course. I trained successfully for up to 5 cubes (with marginal accuracy in obstacle avoidance, but somewhat useful, as seen in the video attached to the project) of randomized dimension in the 25 meter course; the randomized location meant that the cubes could spawn in front of a linear path or outside of a linear path; this meant that if there was a straight path, the agent had to act accordingly as well as if the linear path was obfuscated by blocks.

Behavioral Cloning

Behavioral cloning allows a user to carry out multiple episodes of training using their own human dexterity and manual input, which can significantly increase training speed, as the reinforcement learning can utilize these examples. Behavioral Cloning "trains the Agent's policy to exactly mimic the actions shown in a set of demonstrations" (ML-Agents Overview, 2024). It is a very useful methodology for increasing training speed without having access to massive compute. The Unity Input System was utilized for human input utilizing a video game controller for the purposes of the project and recording of the demonstrations that the autonomous agent was trained on. The movement of joysticks mapped to application of force.

Generative Adversarial Imitation Learning

Generative Adversarial Imitation Learning (GAIL) is a learning policy with similarities to behavioral cloning that attempts to have a demonstrated behavior and outcome competed against, where the autonomous agent attempts to behave the way that your demonstrations do. When this is carried out, the difference between it and behavioral cloning is that in GAIL there are two neural networks rather than one, with a second one, the discriminator, that is taught to try and figure out if a given observation or action is carried out by an autonomous agent being trained or by a demonstration. The agent is rewarded for "tricking" the discriminator into thinking that what it did was actually the demonstration; this, in turn, drives convergence on the behavior in the demonstration (ML-Agents Overview, 2024).

PPO Training Configuration

Various hyperparameters and configurations can be utilized with OpenAI Gym and Unity ML-Agents' integration with it in order to define the specific neural network architecture that you would like to use for your training. The following training configuration was used to define the way the machine learning environment trains:

PPO Hyperparameters

Batch_size	2048
Buffer_size	5000
Learning_Rate	1.0e-5
Beta	5.0e-4

Epsilon	2
Lambd	.99
Num_Epoch	10
Learning_Rate_Schedule	Linear
Beta_Schedule	Constant
Epsilon_Schedule	Linear

Neural Network Settings

Normalize	true
Hidden_units	256
Num_layers	3

Imitation Learning Settings

Type	GAIL	Behavioral Cloning
Demo	<attached_to_submission_of_final_deliverable>	<attached_to_submission_of_final_deliverable>
Strength	1.0	1.0
Use_actions	True	NA

Results and Conclusions

Adding Behavioral Cloning (BC) and Generative Adversarial Imitation Learning (GAIL) increased speed of the training, cutting the number of time steps required to get a result for the task. It took 7,000,000 steps to successfully be able to simply move forward toward the goal, and then only 2,000,000 utilizing that as a base in order to create some marginal ability to dodge obstacles, a much more complicated behavior. Additional training will be carried out to increase obstacle avoidance further.

Additionally, using a curriculum learning approach was effective in training one behavior first that was a necessary behavior for further behaviors. These methods outperform a standard Proximal Policy Optimization (PPO) reinforcement learning paradigm by giving a reference point for the autonomous agent to train against and, in the case of GAIL, for a discriminator to compare autonomous agent behavior with as an adversarial game that drives convergence toward the demonstrated behavior.

Future Scope

The most useful future scope will be to first iterate on the model and do additional training cycles and fine-tune the reward function further to get a more robust obstacle avoidance methodology, as the result was marginal in the category of obstacle avoidance, but successful in purely navigating toward a finish line from a start line. Beyond that, I can then move from the simulated environment into a real-world testing environment for a physical drone to test simulation against reality. This tests assumptions and can expose model deficiencies that may exist as a result of utilizing a simulation for previous testing.

One improvement that could be carried out, but wasn't for the purposes of this project, is the addition of additional actions, such as z-axis rotation. Along with increasing the number of obstacles that must be avoided, in future simulations this could be trained on; adding an additional continuous action increases training time requirements, but also makes it so that the autonomous agent has a far more robust set of actions. There could be some exploration into whether or not GAIL and BC work better with specific human input methodologies, as I did not put much consideration other than heuristic consideration into reaching the end goal utilizing the video game controller. It could be that specific methods of providing input via the joystick provide better training results. This could be another avenue of further exploration.

Another consideration for future development might be to attempt a pure vision-only model, which uses only camera data as inputs to map to output actions, similar to what is used by companies such as Tesla for autonomous navigation. This has potential to be more robust in varying circumstances, can lead to the building of an internal world model, and can potentially allow for more cost-effectiveness in terms of required hardware to allow for successful autonomous for a drone. Finally, a more granular physics simulation can allow for more robust translation into the real-world, something that can be iterated on prior to moving into a physical environment (or upon confirming current validity in a physical setting). More robust simulation of aerodynamics and/or specific considerations related to power sources, heat, etc. could provide better translation to physical environments. In this vein, comparison with other physics simulation environments such as MujoCo, ROS Gazebo, and Nvidia Isaac Sim should be done to see which simulation environment has the most robust simulation features.

This project increased my understanding of Unity ML-Agents and deep reinforcement learning for autonomous agents, a main control system training methodology for robotics and intelligent autonomous systems, the degree program within which I'm enrolled. It built upon my learning in rigid body dynamics and exercised my machine learning and robotics software engineering skillset. Quality control was emphasized in testing against randomized environments in a randomized way to attempt to increase robustness of the model intelligent autonomous system for the autonomous drone agent being trained against, noting that additional robustness should be strived for with further training. I had to engage in project management to ensure that I met deadline requirements, adjusting scope as necessary while still engaging deeply with the material. It was an effective exploration of Unity as a physics simulation environment that is cross-functional with game development that is being increasingly used for robotics training and physics simulation, including through partnerships with companies such as Nvidia and OpenAI (Unity, n.d.). Through simulations like these and through engaging with reinforcement learning literature and experimenting with various training paradigms, a greater understanding of the pros and cons of various approaches and a more intimate knowledge of Robotics and Intelligent Autonomous Systems was cultivated.

References

John Schulman, F. W. (2017, Aug 28). *Proximal Policy Optimization Algorithms.* Retrieved from Arxiv: https://arxiv.org/abs/1707.06347

ML-Agents Overview. (2024). Retrieved from Unity-Technologies Github ML-Agents Repo: https://unity-technologies.github.io/ml-agents/ML-Agents-Overview/#gail-generative-adversarial-imitation-learning

Sharma, S. (2024). *Drone Development from Concept to Flight.* Birmingham, UK: Packt Publishing.

Unity. (n.d.). *Physics Solutions For Game Development*. Retrieved from Unity.Com: https://unity.com/solutions/programming-physics

Acknowledgements: I utilized a 3d model purchased from the Unity Asset Store for the drone, but no scripts included with it; all functional coding was carried out by me and it was just for the enjoyment of the model. This is the 'Drone_Parent' model found in Assets/PBR Racing Drone/Prefabs in the associated Unity projected included at link: https://drive.google.com/drive/folders/1MPqSRZAh0f-k44wmo5pNAC1rGG5Jh9Uz?usp=sharing