Neural-Navigator
Training an ML-Agent to Drive a Car in Unity

Date: 12/04/2024

Steven Broaddus and Erik Nilsson

1. Introduction

With the advent of machine learning techniques in real-world applications, it has become imperative that computer engineers who want to stay at the cutting edge of technology are able to learn how to properly utilize these techniques for their own use. One of the most versatile machine learning techniques that has emerged in recent years is reinforcement learning due to the ability for agents utilizing this technique to learn how to properly complete any task given proper training and enough time. Many applications have seen convenient packages or plugins that make it easy for users to integrate reinforcement learning techniques into the application's environment, and Unity is no different. The `ML-Agents` package for Unity provides a convenient way to provide reinforcement learning within any environment, and understanding how to properly utilize it is crucial for any machine learning engineer who wants to take advantage of the Unity engine.

This project aims to implement a Unity environment with integrated reinforcement learning based agents to provide the researchers with an understanding of how to properly utilize the ML-Agents package within Unity. The goal environment is to train a model that can effectively drive a somewhat realistic car across any track using a checkpoint system alongside various components of the ML-Agents Unity package. In order to use ML-Agents, users must run the `mlagents-learn.exe` executable that must be installed within a Python environment alongside the Unity project. The executable must be running within a separate terminal when the Unity project starts for the two applications to be able to communicate. Since this process takes a lot of time to implement, the researchers plan to provide a video detailing how the agents can train through the package while also providing a final Unity project with multiple trained brains with varying levels of efficiency to exemplify how the training has worked within the environment.

2. Experimental Methodology

To begin understanding how to properly utilize the ML-Agents package in Unity, the researchers determined that they should follow a tutorial on ML-Agents within a much more simplified environment to get a grasp on how it properly works. The tutorials followed for this initial implementation can be found at [3]. This initial experiment consisted of a simple block agent, a square environment enclosed by walls, and a goal "egg" that the agent must collect. This is a simplified environment where the agent only receives its own position and the position of the goal as observations, and can only move by directly translating its X and Y positions. Shown in Figure 1 is the prefab for the environment.

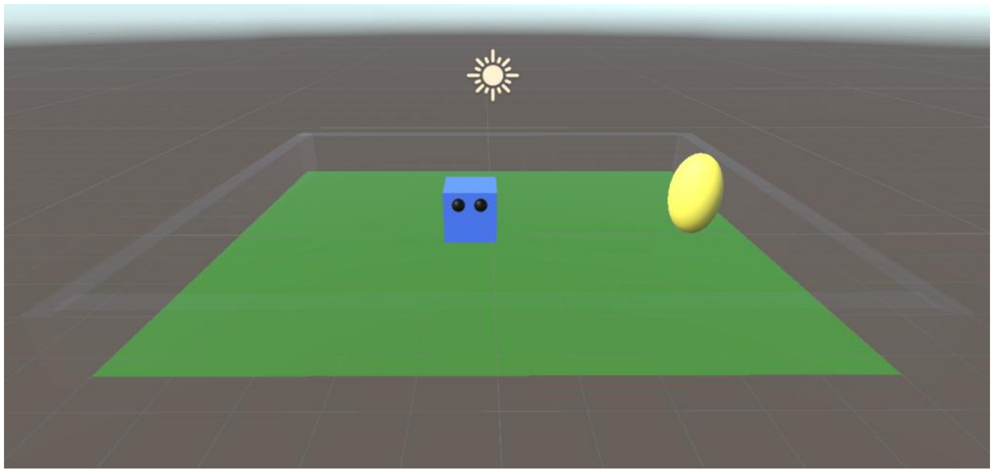


Figure 1: "Hungry Mort" initial ML-Agents testing environment.

Once the simplistic environment was created, with tags and colliders given to the appropriate `GameObject`s, the ML-Agents package was integrated into the project. To do so, the researchers followed the installation instructions found at the ML-Agents GitHub Repo Web Docs. This consisted of installing the correct Python version, creating a virtual environment, and installing the required Python packages to use the ML-Agents package (`torch` and `mlagents`). Once this was installed, the default configuration for the ML-Agents package within Unity, once applied with its integrated components to the simple agent was enough to get it to train after scripting the observations and the rewards for the agent. Once successfully integrated into the project, the team was able to clone the environment multiple times to increase the speed at which the agents effectively learned. This resulted in parallel training, exemplified within Figure 2.

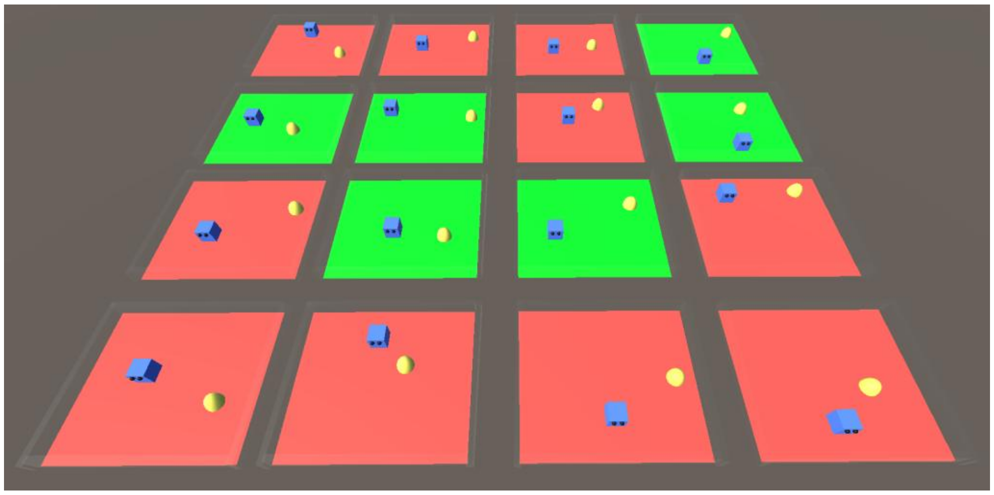
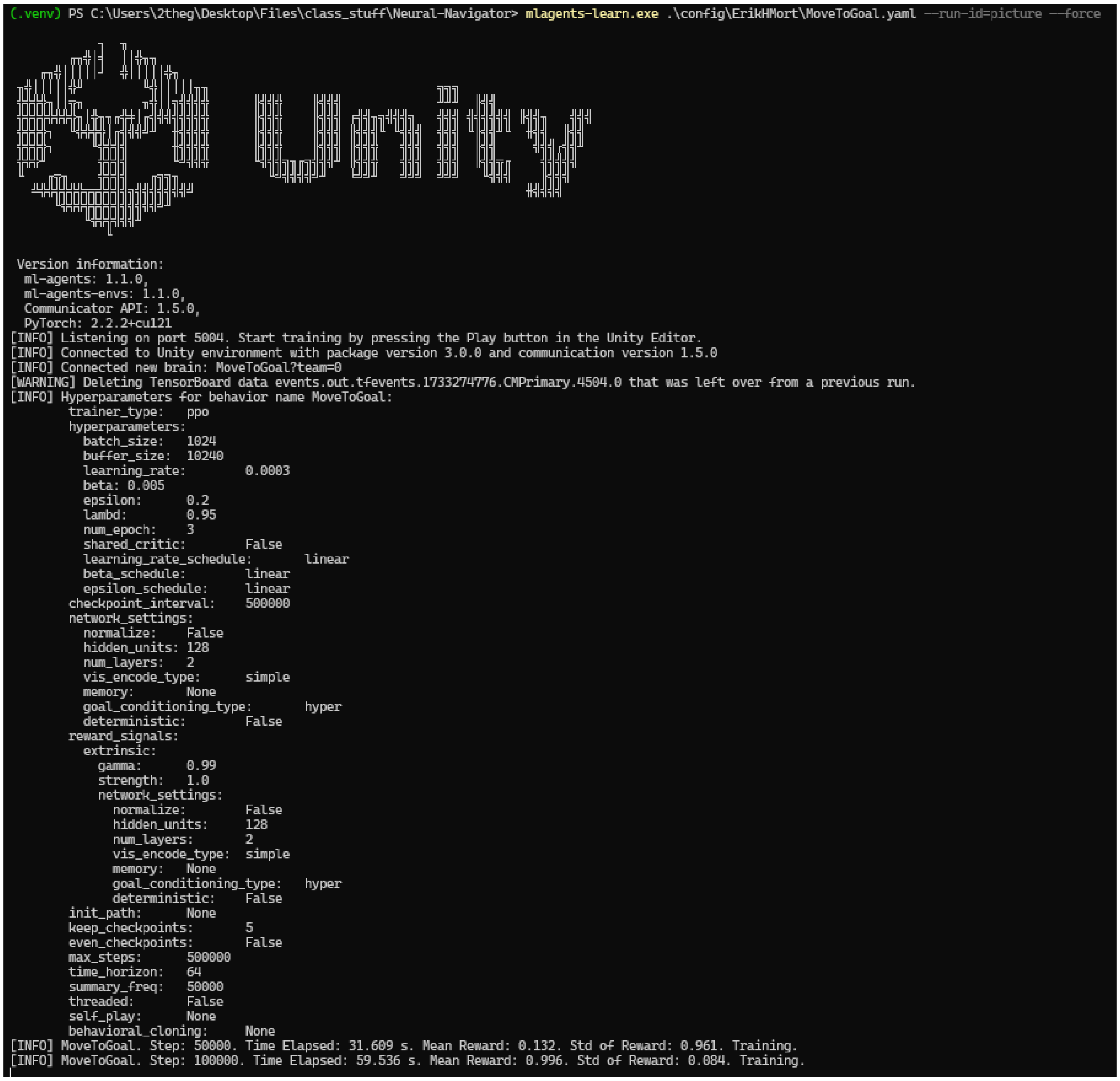


Figure 2: "Hungry Mort" training environments.

This successfully ran multiple instances of the agent to learn how to effectively collect the goal. In order to train the agents, the `mlagents-learn.exe` program must be run in a separate window. This results in a server being run and exposed through a port. The output of running this program can be seen in Figure 3.



```
(.venv) PS C:\Users\2theg\Desktop\Files\class_stuff\Neural-Navigator> mlagents-learn.exe .\config\ErikHMort\MoveToGoal.yaml --run-id=picture --force

 Version information:
  ml-agents: 1.1.0,
  ml-agents-envs: 1.1.0,
  Communicator API: 1.5.0,
  PyTorch: 2.2.2+cu121
[INFO] Listening on port 5004. Start training by pressing the Play button in the Unity Editor.
[INFO] Connected to Unity environment with package version 3.0.0 and communication version 1.5.0
[INFO] Connected new brain: MoveToGoal?team=0
[WARNING] Deleting TensorBoard data events.out.tfevents.1733274776.CMPrimary.4504.0 that was left over from a previous run.
[INFO] Hyperparameters for behavior name MoveToGoal:
        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: 128
          num_layers:   2
          vis_encode_type:      simple
          memory:       None
          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:   None
              goal_conditioning_type:   hyper
              deterministic:    False
        init_path:      None
        keep_checkpoints:       5
        even_checkpoints:       False
        max_steps:      500000
        time_horizon:   64
        summary_freq:   50000
        threaded:       False
        self_play:      None
        behavioral_cloning:     None
[INFO] MoveToGoal. Step: 50000. Time Elapsed: 31.609 s. Mean Reward: 0.132. Std of Reward: 0.961. Training.
[INFO] MoveToGoal. Step: 100000. Time Elapsed: 59.536 s. Mean Reward: 0.996. Std of Reward: 0.084. Training.
```

Figure 3: `mlagents-learn.exe` program running in terminal alongside Unity project.

Within Figure 3, it can be seen at the bottom that the mean reward increased substantially close to the max of one for the environment. This demonstrated the agent was successfully able to train to complete the task.

Once the team successfully implemented the training of the agents in the simple environment, the researchers moved on to implementing the car controller that will be used within the final project. In order to understand how to implement a car controller that mimicked many of the controls of a realistic car, the researchers followed the tutorial playlist at [5] which detailed how to create a car with a `RigidBody` that could interact with its environment like a real car would. This tutorial went in-depth, providing realistic methods for control including a steering curve, wheel colliders for the car prefab to provide physics-based feedback, engine noise, proper engine simulation, and proper gear systems. However, to provide varying levels of complexity for the agents to train at, the researchers implemented various versions of the car controller with different levels of complexity. The "Simple Car" consisted of a car controller that took in gas (also break) input and steering input through Unity's New Input System and applied a steering curve and physics-based acceleration via Unity's physics engine with `RigidBody`s to control a somewhat realistic car. Another "Complex Car" was created with the goal of mimicking more of the functionality of a realistic car as aforementioned. With the successful training of a model using the Simple Car, the Complex Car will be destined for future work.

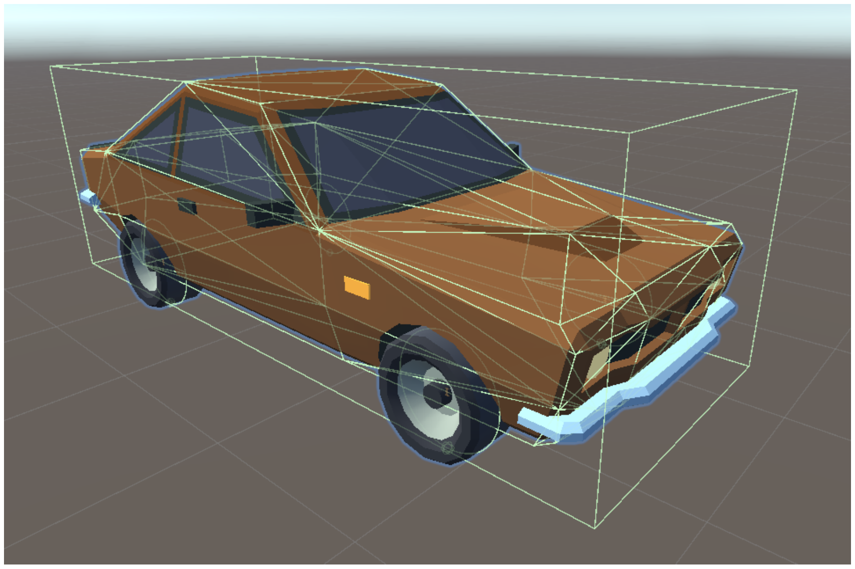


Figure 4: "Simple Car" prefab with a box collider (disabled in practice), two mesh colliders for the body, and four wheel colliders for the Unity physics engine to work with.

Shown in Figure 4 is the prefab for the simple car, taking an a free car prefab from the Free low poly car pack by Nebula on the Unity Asset Store.

In order to begin training the car model, the team first implemented several tracks of varying complexity using the free package Simple Roads by Stepan Drunks on the Unity Asset Store. In addition to created several tracks, the team added checkpoints in order of progression to the track to act as a reward mechanism, following the part of the Unity ML-Agents tutorial Simple Checkpoint System by Code Monkey that was referenced in the simple ML-Agents car tutorial at [3]. The simplest and first loop track created with checkpoints can be seen in Figure 5.

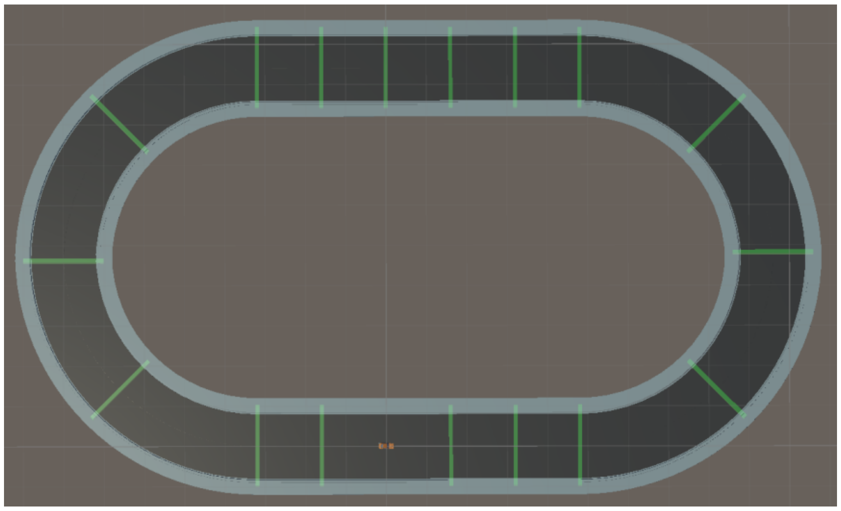


Figure 5: Simple left-looping oval track with checkpoints for training.

In order to see the walls within the track, the team then added the “Ray Perception Sensor 3D” component alongside the “Behavior Parameters” component and the agent script component using the ML-Agents package to provide rays that return tag indices upon hitting colliders. Through this, the agent can see surrounding walls and objects as needed. Shown in Figure 6 is a car agent with the attached ray sensors.

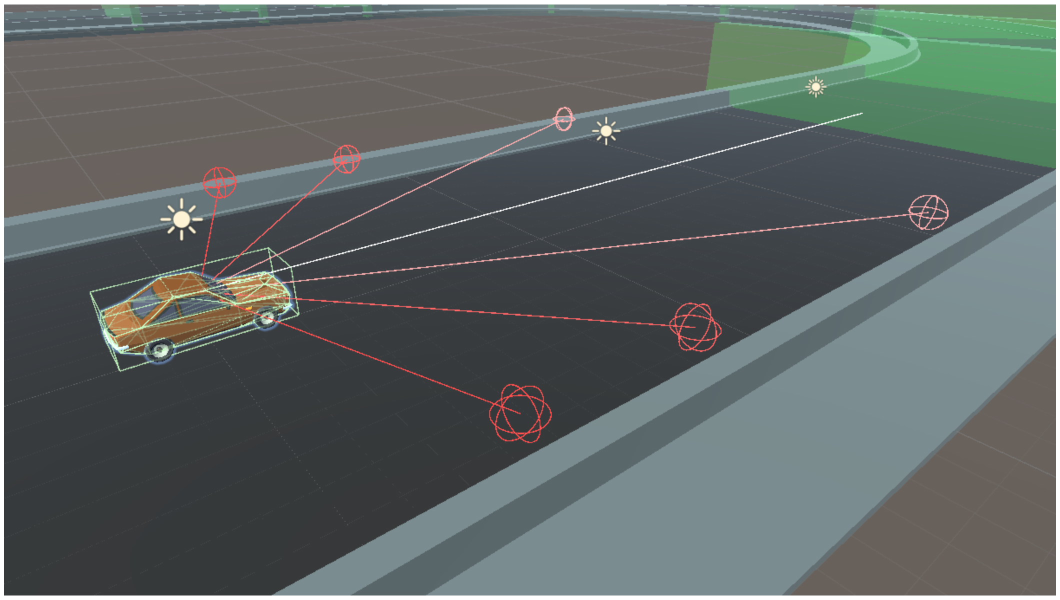


Figure 6: Ray Perception Sensor 3D component on car agents.

Once the agents were fully scripted and customized with a robust reward system of checkpoints and walls, the project was successfully set up with only training required to successfully develop the models.

3. Results

3.1. Completed Functionality

With the successful setup of the training environment and ML Agent, the following functionality is presented within the completed project:

- Scenes with the "Hungry Mort" training environment for an ML-Agents introduction
- Two car controllers with varying levels of complexity
    - "Simple Car" controller
        - Simple steering and gas input through the New Input System
        - \`RigidBody\` physics-based behavior with wheel/mesh colliders

        - Used for training
    - “Complex Car” controller
        - Realistic steering curves, slip angles, gear systems, engine simulations, an RPM gauge, engine sound, and horsepower curve
        - Wheel collider friction coefficients designed for drifting
        - Enter scene “ComplexCar” if interested. `Shift` for manual gear shift up when min. RPM is reached (5500
        - Not used for training (future work)
- Several tracks with varying levels of complexity
- Checkpoint system for ML Agents
- Ray Perception Sensors with tags and colliders for additional agent observations
- Visualizations for training via `tensorboard` (provided through the ML-Agents package)
- Effective trained models for the `LoopLeft`, `LoopRight`, and `Figure8` tracks
    - Models experimented with millions of steps (~9 hours)
    - Models trained within a headless environment to permit more agents (used 150 agents for major models)

3.2. Original Techniques Beyond Tutorials

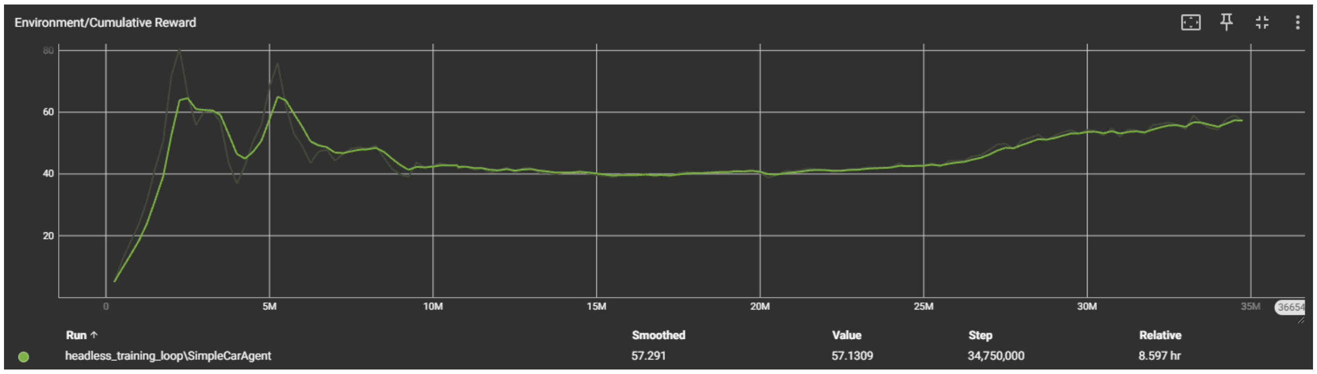
Although the team generally followed tutorials for the majority of the base implementations (everything wasn’t that easy, and downloadable project files weren’t complete), the team also implemented some original or further developed techniques beyond the tutorials. While the novelty of the techniques in a broad sense is not considered, the team proudly presents the following original implementations:

- Custom made tracks using the Simple Roads by Stepan Drunks Unity asset

- All prefabs and `GameObject`s were either based on a free Unity asset not in any of the tutorials, or self-made
- The AI Car tutorial by Code Monkey did not use the more complex Car Controller by Nanousis Development, which required significantly more training
- The checkpoints tutorial was incorrect for multiple agents. There were bugs that were not noticed due to the simplistic nature of the tutorial controller
- The trained models are completely original and developed through ML-Agents. The team could not find another tutorial with ML-Agents used for this kind of car controller
    - The team's models learned to navigate with elevation
    - The team utilized headless training to mass train agents, which also required learning how to build projects within Unity
- The statistics UI window within Unity was not based on any tutorials
- The complex car tutorials were followed to create a complex car, which is beyond the scope of the minimum aim of the project
- The imitation learning experimentation was based on Code Monkey's tutorial, and was also beyond the aimed scope of the project

3.3. Model Evaluation

When training the models, ML-Agents goes through steps over time, which is dependent on the engine time scale, the amount of agents, and other confounding factors. Regardless, this results in an incomprehensibly expansive discrete spectrum of models to select from. To make this usable for humans, ML-Agents takes checkpoints throughout the training process alongside the final model compiled when the training stops. However, how does one understand the performance of the models? This is accomplished through summaries of the checkpoint models' performances with the `Mean Reward` and the `Std. of Reward` being the primary metrics for comparison. These are effectively summarized through graphs within a built in `tensorboard` application that consolidates the reward values into something a human could interpret. The primary means of evaluating the models within this project were through a cumulative reward graph and a cumulative reward histogram. These visualizations showcase the general movement of rewards over the course of training, which is sufficient for this simple project. Shown in Figures 7 & 8 are the graphs from the large training experiment for the `LoopLeft` model.



Run ↑	Smoothed	Value	Step	Relative
headless_training_loop\SimpleCarAgent	57.291	57.1309	34,750,000	8.597 hr

Figure 7: Cumulative reward against training steps for `LoopLeft` model training experiment.

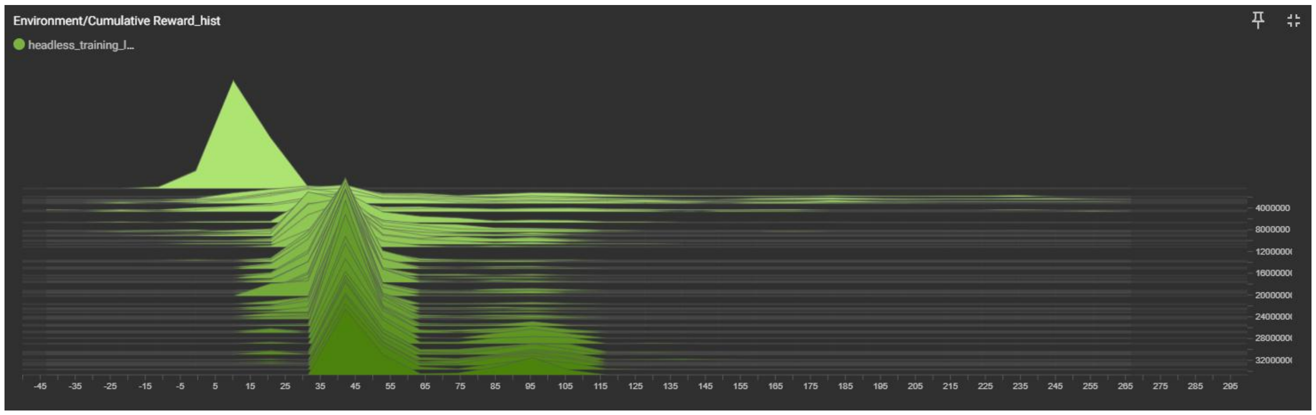


Figure 8: Cumulative reward histogram for `LoopLeft` model. Each histogram (steps/time progresses downwards) represents the distribution of rewards for the agents at the step count. The distribution shifting right over time signifies productive training.

When training the models, the team encountered some fundamental issues within reinforcement learning, with the primary issue being that more training doesn't always equate to better performance (see overfitting and/or double descent). In fact, although the team trained the `LoopLeft` model for approximately 32M steps, the most effective model was trained at 2.5M steps, demonstrating the need to be aware of proper reinforcement learning techniques.

Despite the challenges encountered, the team successfully developed models that could traverse tracks that resemble the ones that they trained on, with the potential for scaling complexity. Trained models are also available for future transfer learning, with great potential for quickly adapting to tracks with new features. Shown in Figure 9 portrays an instantaneous moment taken of the final effective models successfully navigating through a variety of tracks.

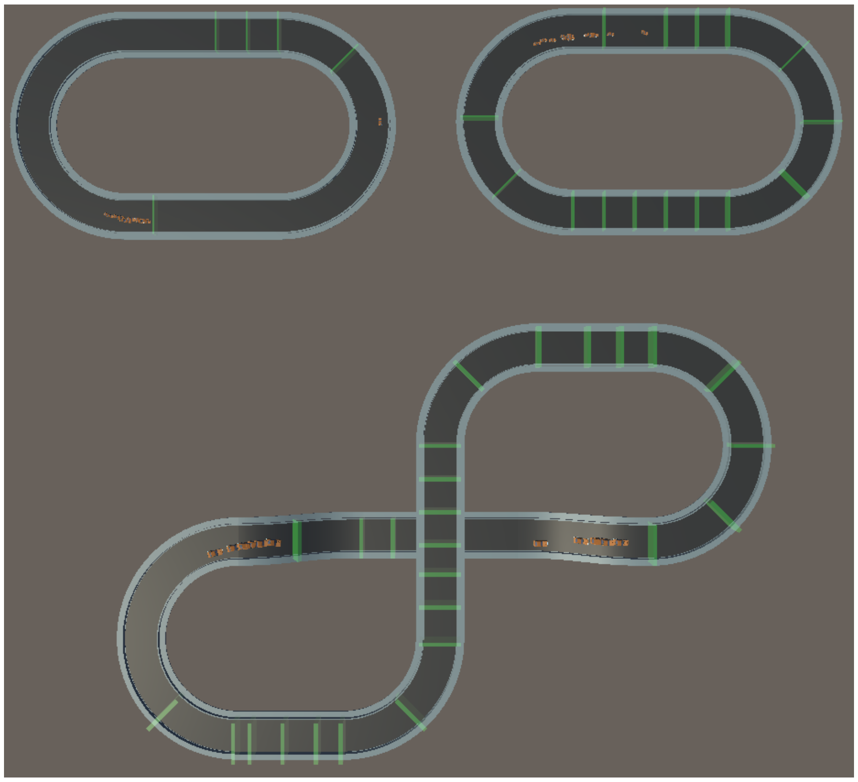


Figure 9: Agent cars running inference only on the trained effective models for tracks with varying complexity. Top left loop track is left turning, with the other being right turning.

Shown in Figure 10 is a side perspective view of the cars successfully navigating through the more complex figure 8 track.

With the successful implementation of the ML-Agents package within the Unity project, the team has completed the project. The models are able to effectively navigate through the tracks despite the increased complexity, and demonstrate how the ML-Agents package can be used to implement reinforcement learning techniques within a Unity environment.

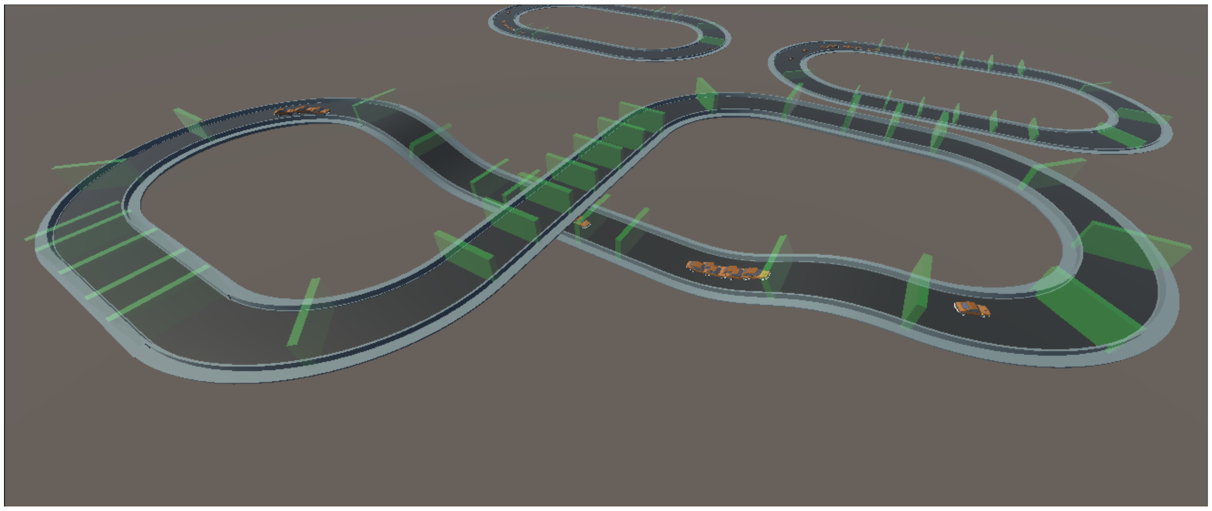


Figure 10: Agent cars successfully navigating through the complex figure 8 track.

4. Discussion & Conclusion

4.1. Researcher Contributions

4.1.1. Steven's Work

Steven's Work

Hungry Mort (Simple MLAgents Test)

- We both implemented our own versions for learning purposes

Car and track prefabs

- Both complex and simple car controllers

Car controllers, input systems, and offset cameras (for players)

ML-Agent scripts for car agents

Modified car agent controllers to utilize checkpoints

Camera switching system

Trained Overnight Model of Simple Agent (LoopLeft track)

Although the work for the project was primarily done simultaneously (lots of overlap), there were some notable delegations of work that differed between both researchers. Once both researchers were past the initial learning stages for ML-Agents, which included following the ML-Agents endorsed tutorials by Code Monkey, Steven proceeded to focus on implementing the car controllers, the car agents, and integrating them into the systems required for ML-Agents to have effective reinforcement learning for varying levels of complexity. This process consisted of finding free Unity assets (please refer to Experimental Methodology for further details), and then following Nanousis Development's car tutorials to implement two car controllers with varying complexity to prepare for the implementation of reinforcement learning. This was an in-depth process, which required a more developed understanding of how car physics is approximated and can be replicated within Unity.

Once the car controllers, prefabs, and scenes were prepared for utilization, Steven focused on integrating the ML-Agents package into the agents. This required adding components to the `GameObject`s to provide observations, creating and adding a script to inherit and implement the `Agent` class within ML-Agents, and modifying values in the configuration, inspector, and scripts to get a viable model. Note that this was also simultaneous.

To polish the final product for submission, Steven prepared several scenes and placed checkpoints so the system could effectively work. Steven also added a camera system and corrected positioning to provide for a more convenient user experience. Due to the intensive computational power and time required to experiment with long-trained model, both researchers trained models to learn how to drive in the track.

4.1.2. Erik's Work

Erik's Work

- Hungry Mort (Simple MLAgents Test)
- Hungry Mort Imitation Learning Test
- Checkpoint System for Agents on Track
- Debugged Simple Car Agent
    - Game Object Properties
    - Interactions with other Track Components
- Tuned Simple Car Agent Model
    - Observations and Reward Criteria
    - Training Parameters
- Implemented Headless Training on Built Environment
- Trained Overnight Model of Simple Agent (Figure 8 track)

Erik mainly focused on Model Optimization and MLAgent's Capability Research. Above is a list of all major changes he contributed as the project developed. All team members initially followed the basic Food Seeking Agent (Hungry Mort) tutorial with the goal of MLAgent's Familiarization. Erik took this Food Seeking Agent tutorial one step further by following an Imitation learning tutorial [6]. This was to develop skills and understanding of the tools for later optimization of Car Agent Models.

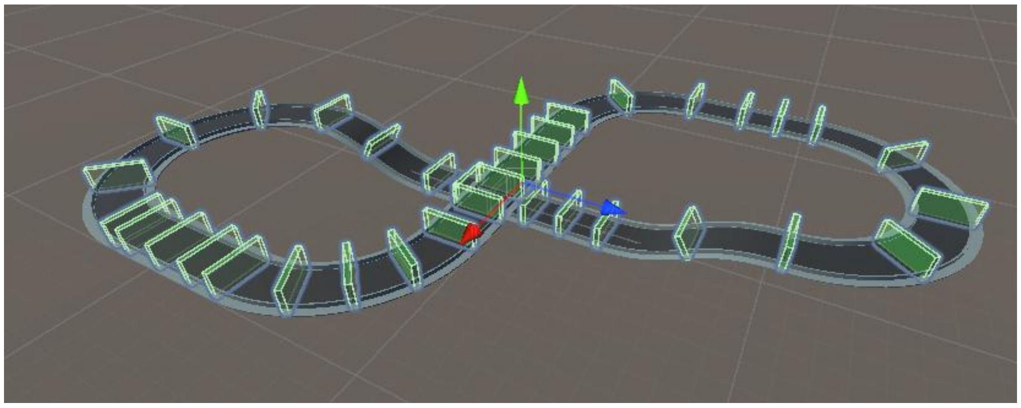


Figure 11: Checkpoints (Green Rectangles) on figure 8 track

Additionally, Erik created a checkpoint system for training multiple Car Agents on the Same Track. This entails keeping track of each car's progress through checkpoint segments. The system will notify Agents as they pass through a checkpoint if it was the correct "next" checkpoint or an incorrect one. This functionality can be found within: "CheckpointSingle.cs", "TrackCheckpoints.cs", and "SimpleCarAgent.cs".

Following Steven's initial creation of the tracks and bare bones Car Agent, Erik debugged several issues in Car -> Environment interaction. These were mostly Unity Environment issues like tag, layer standardization, and recognition between collisions. Next, the Observation and Reward systems were reworked and debugged. Ray Sensors which radiated out the front of the car were hooked up to the Agent Observation parameters for the model along with state information of the car. State information includes current speed, current wheel angle, direction to next checkpoint, and current direction.

The crux of this project lay in fine tuning model parameters and training settings. Erik did much of this with the help of Steven, reading MLAgent's documentation and progressively tweaking the model structure until a stable model was converged upon. Once this point was reached, Erik added Headless Server training to allow for multiple environments with many agents to train at once. Headless training is also significant in that it uses a compiled unity executable (instead of the editor window) and doesn't require graphics. This new method yielded immense computational benefits and vastly improved training speed.

Lastly, Erik used headless training on his Personal Computer overnight to create a Simple Agent Car model which is optimized for the figure 8 track. It is important to note that while this training ultimately didn't converge on a "perfect" solution, it laid the groundwork for reasonable time expectations of convergence in our Car Agent Scenario. Below are several figures representing progress over 12 training hours.

The figure above shows a Reward Score Histogram for Agents at each training

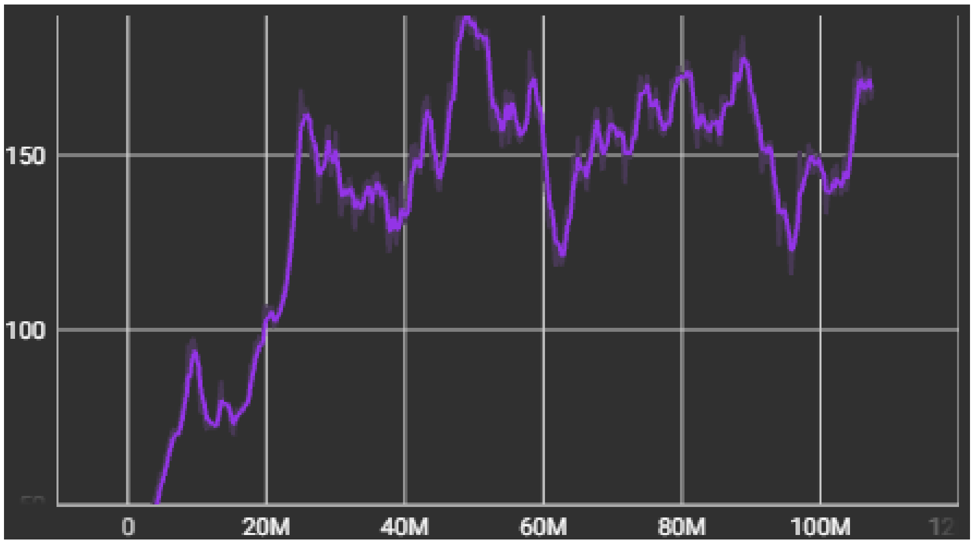


Figure 12: Environment/Cumulative Reward Graph (Figure 8 Track)

checkpoint. Larger peaks indicate a high number of agents at that score. Low peaks imply fewer agents. As you move down the graph, the model gets more and more optimized and thus scores higher values.

Here we can see an interesting pattern where as the model tries to optimize for further along the track, the overall performance has a sharp decline and hits the same waveform as previous training to climb back up to the top.

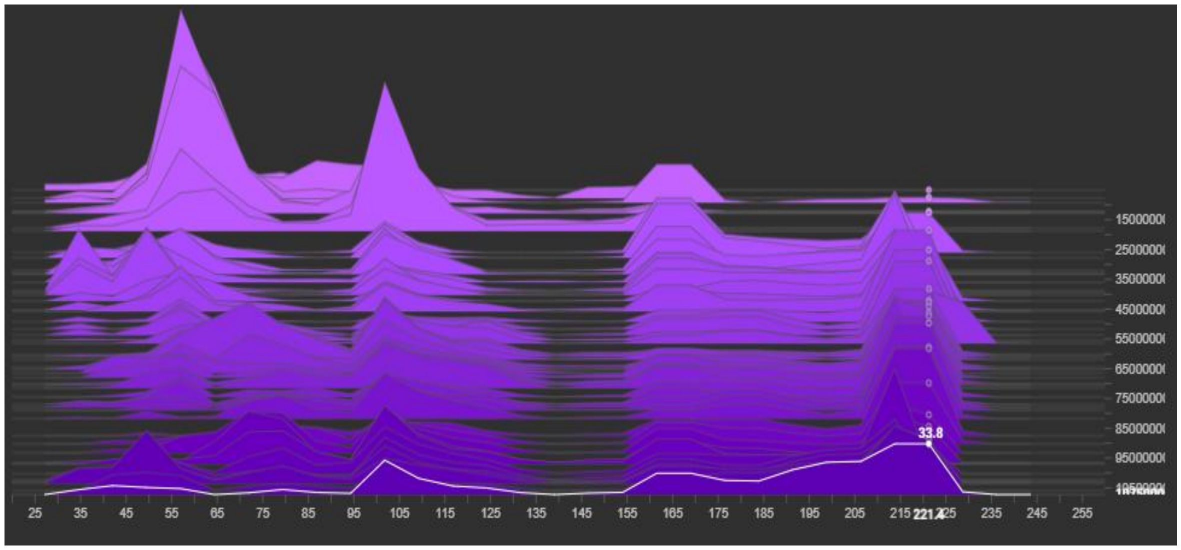


Figure 13: Reward Score Histogram for Simple Car Agent Training (Figure 8 Track)

4.2. Future Work

This project was a basic introduction to the machine-learning based tools available within Unity. The following lists details prospective ideas the researchers have for future work on this or a similar project:

- Add more in-game visualizations that reflect graphs in `tensorboard`
- Switch to a checkpoint-less spline reward system that can be (somehow) conveniently implemented into any track

- Implement a mouse-based selection control to isolate and visualize trainings/statistics of independent agents
- Add obstacles to tracks
- Train models on complex car
- Tire trails, get models to learn to drift, music, and animations

The researchers thoroughly enjoyed this project and look forward to further developing their understanding of data science and computer graphics.

REFERENCES

[1] “AI Learns to Drive a Car! (ML-Agents in Unity),” YouTube, https://www.youtube.com/watch?v=2X5m_nDBvS4 (accessed Nov. 12, 2024).

[2] “Training an unbeatable AI in Trackmania,” YouTube, https://www.youtube.com/watch?v=Dw3BZ6O_8LY (accessed Nov. 12, 2024).

[3] “Machine learning ai in Unity (ML-agents),” YouTube, https://www.youtube.com/playlist?list=PLzDRvYVwl53vehwiN_odYJkPBzcqFw110 (accessed Nov. 12, 2024).

[4] “Spider Evolution Simulator,” YouTube, https://www.youtube.com/watch?v=SBfR3ftM1cU (accessed Nov. 12, 2024).

[5] “Car Controller,” YouTube, https://www.youtube.com/playlist?list=PL0JXhw1odpJLTRBDdv4ybtYkuD1lEcF-N (accessed Dec. 3, 2024).

[6] “Teach your AI! Imitation Learning with Unity ML-Agents!” https://youtu.be/supqT7kqpEI?si=zHciYkJ7ruAtEKfO (Accessed Dec 4, 2024)