TEAM: PARK IT RIGHT!

CSCI 527 - Applied Machine Learning for Games

Engineering Design Document

Authors:
Keerthana Nandanavanam (nandanav@usc.edu)

Krishna Manoj Maddipatla (km69564@usc.edu)

Nidhi Chaudhary (nidhicha@usc.edu)

Sumanth Mothkuri (mothkuri@usc.edu)

April, 2021

Table of Contents

List of Figures

List of Tables

1 Goal

The focus of this project is to develop an agent that is adept in obstacle avoidance and seamless navigation. The project makes use of a multi-level car navigation game [1] in Unity where a car must navigate through fixed and moving obstacles and park at the highlighted spot. The goal is to train the agent for two levels of the game in Unity using different machine learning algorithms, tuning different hyperparameters and evaluating the results on the tensorboard. The most challenging aspect is navigation through moving obstacles through a portal when the target is located on a different storey.

2 Background and Motivation

We are in a generation of automation where most of the commonplace tasks are carried out by well trained automated systems. One such diurnal task is driving a vehicle and parking it at the appropriate parking spot. This problem opens doors to introduce an automated parking system where an artificially intelligent agent tries to overcome obstacles in its way and park the car in the right spot.

The history of self driving cars goes back to the early 1920s when experiments were started to turn the fantasy of autonomous driving into reality. The introduction of DARPA's grand challenge in 2004 caused a tremendous uprising in this technology [2]. Tesla, Waymo, BMW and many other mega corporations are now actively investing in this trend. Out of the many design considerations for such an autonomous vehicle, having a good parking system that helps in navigating through obstacles, identifying the right spot and parking the vehicle is of paramount importance. We drew inspiration from this exact problem and strived to develop a machine learning agent capable of doing the same using reinforcement learning. The fact that this technology can also be used for vacuum cleaners which can navigate through household items and clean the surface thoroughly has strengthened our motivation to work towards this project. Generally, it could be used for any obstacle avoidance and navigation system and hence it can have multiple applications from medicine to defense industry.

3 Prior Work

3.1 Obstacle Avoidance and Navigation Systems

There has been significant research in the field of obstacle avoidance and navigation systems using reinforcement learning [3] and since this problem is the parent problem of our use case, we decided to experiment further on the research done in this domain. Proximal Policy Optimization (PPO) algorithm was found to be showing great results for the problem. So, we also drew inspiration from it, and incorporated PPO as our base algorithm.

3.2 Parking occupancy detection using CNN

There has been a lot of related work carried out in the area of parking occupancy detection systems using Convolutional Neural Networks and Support Vector Machines (SVM) [4]. The classifier is being trained and tested by features learned by deep CNN from public datasets (PKLot) having different illuminance and weather conditions.

3.3 Autonomous Vehicle Control using Reinforcement Learning

A lot of promising research has been conducted using reinforcement learning for strategic decision making [5]. The autonomous exploration of a parking lot is simulated and the controls of the vehicles are learned via deep reinforcement learning [6]. A neural network agent is trained to map its estimated state to acceleration and steering commands to reach a specific target navigating through the obstacle course. Training was performed by a proximal policy optimization method with the policy being defined as a neural network. This paper also motivated us to look at PPO as our base algorithm.

3.4 Policy Gradient based Reinforcement Learning

A policy gradient based reinforcement learning approaches for self driving cars in a simulated highway environment has been implemented and tested. The research showed that reinforcement learning is a strong tool for designing complex behavior on traffic situations, such as highway driving, where multiple objectives are present, such as lane keeping, keeping right, avoiding incidents while maintaining a target speed [7].

3.5 Self-Driving Cars Using CNN and Q-Learning

This paper focuses on learning of self-driving vehicles irrespective of how the hardware is established [8]. The two approaches used were supervised learning and deep reinforcement learning. Supervised learning using Convolutional Neural Networks was done for feature extraction and reinforcement learning helped the car learn from its experiences. Training was done in a constrained simulated environment mimicking some real life situations such as obstacles and road signs.

4 Game Overview

Unity is one of the most popular game development engines that provides built in features like physics, 3D rendering, collision detection without having to reinvent the wheel for developing a game. Thus, we decided to go ahead with the Unity environment as it is a great simulation environment and offers cross platform development. Additionally, Unity offers ml-agents package which enables simulations to serve as environments for training agents. They provide implementations (based on PyTorch) to train intelligent agents, and also provide great support for reinforcement learning, imitation learning and many other methods.

We found an open source 3D game developed in Unity and made modifications to it according to our use cases. The game environment is described in the following sections.

4.1 Level 1

Level 1 of the game consists of a bounded arena with a car starting at an arbitrary position and a parking spot appearing at another random position. The goal location or parking spot is highlighted in red color. The car has to first identify the highlighted spot and then navigate towards it through three obstacles that are placed in the center of the arena as depicted in Fig. 1.

4.2 Level 2

Level 2 unlike level 1 is more complex. It consists of a bounded arena similar to level 1 but with moving obstacles and portals to navigate to different storeys. The car will start at an arbitrary position and a parking spot will appear on the same storey or a path will be highlighted to another

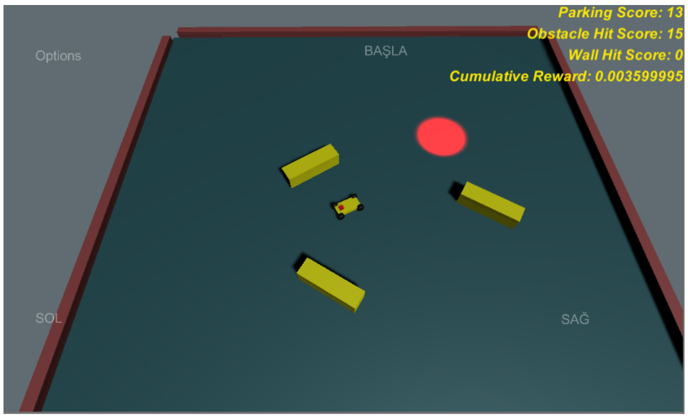


Figure 1: Level 1 of the game

storey. The car has to identify the highlighted parking spot or storey entrance and navigate through moving obstacles in the arena to reach the parking spot. The objects in the yellow color in the Fig. 2 are obstacles which move parallel to the walls of the arena and the object is green color is the portal that can help navigate to different storeys.

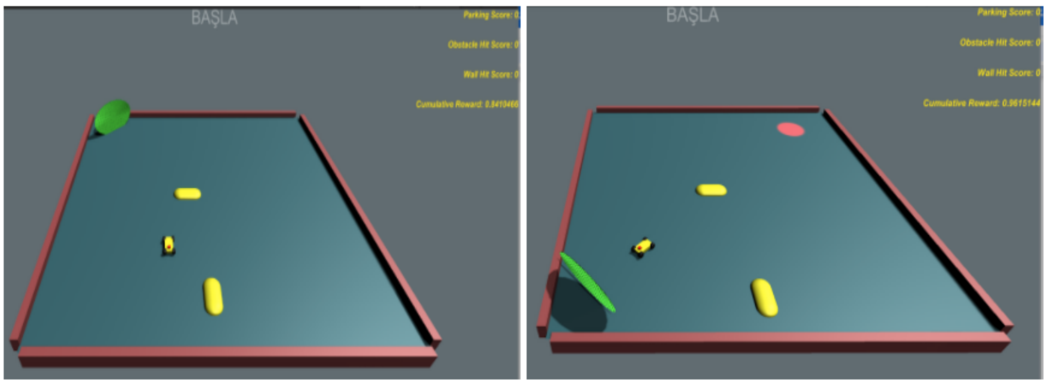


Figure 2: Level 2 of the game - On the left is first storey and on right is the second storey with parking spot highlighted in red color

5 Project Timeline

Having explored numerous games, we decided to go ahead with the open source game we found in Unity - Car Parking [1], because of the vast number of applications for this type of an agent in the real world. Now that the game was selected, we needed to familiarize ourselves with Unity and game development. Setting up the “mlagents” package was the next step as it helped in training an agent to play the game. The game was modified to suit the training environment and once the training pipeline was set up, we tuned the hyperparameters to ensure that the agent learns the best optimal policy for parking in the highlighted spot. Pre-midterm efforts were focused on Level 1 of the game and theoretical research, and post mid-term efforts were focused on Level 2 of the game having moving obstacles and portal navigation to different storeys. Fig. 3 gives a complete overview of the project timeline throughout the semester.

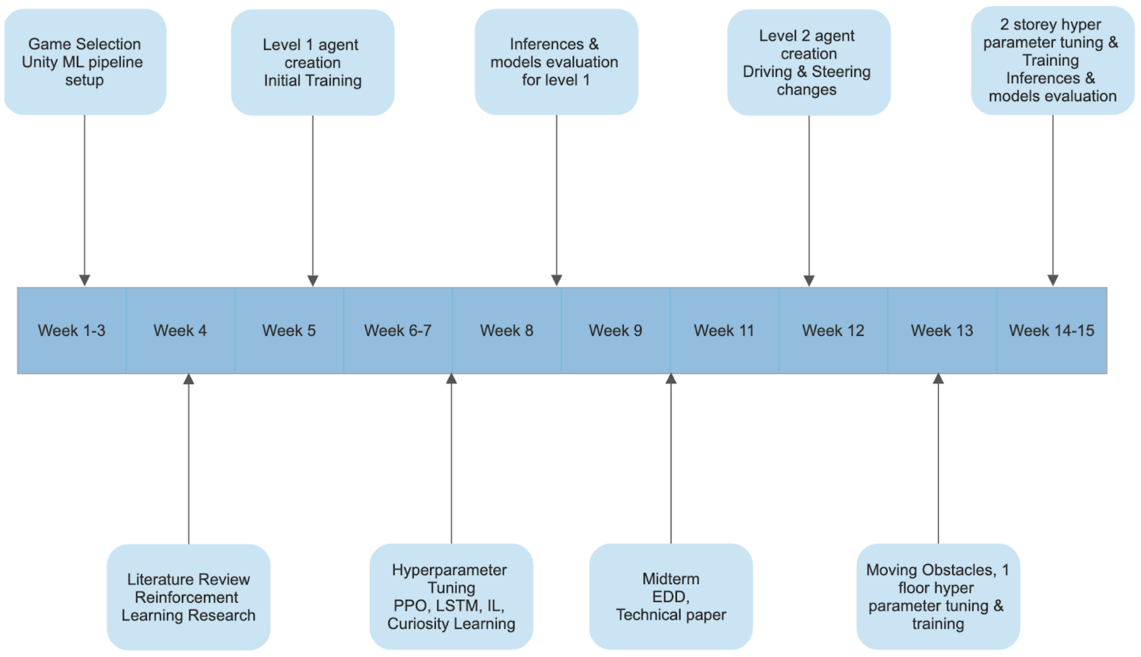

Game Selection Unity ML pipeline setup
Level 1 agent creation Initial Training
Inferences & models evaluation for level 1
Level 2 agent creation Driving & Steering changes
2 storey hyper parameter tuning & Training Inferences & models evaluation
Week 1-3
Week 4
Week 5
Week 6-7
Week 8
Week 9
Week 11
Week 12
Week 13
Week 14-15
Literature Review Reinforcement Learning Research
Hyperparameter Tuning PPO, LSTM, IL, Curiosity Learning
Midterm EDD, Technical paper
Moving Obstacles, 1 floor hyper parameter tuning & training

Figure 3: Timeline explaining the work done at different stages of the project

6 Research

6.1 Reinforcement Learning

Neural Networks have shown a great potential for decision making in game playing agents. However, a simple Neural Network is a supervised ML. It takes an input which is propagated through the layers of the network and produces an output, which is then compared with the actual label and the errors are back-propagated till the network converges. Now, in supervised learning, it's difficult to get the training data. A human player will have to play for multiple hours and data frames will have to be generated from the games played to be fed to the system. Since this is a very tedious, time consuming and error prone task, we decided to move ahead with reinforcement learning.

Reinforcement learning is the training of machine learning models to make a sequence of decisions. The agent learns to achieve a goal in an uncertain, potentially complex environment. In reinforcement learning, artificial intelligence faces a game-like situation. The computer employs trial and error to come up with a solution to the problem. To get the machine to do what the programmer wants, the artificial intelligence gets either rewards or penalties for the actions it performs. Its goal is to maximize the total reward [9].

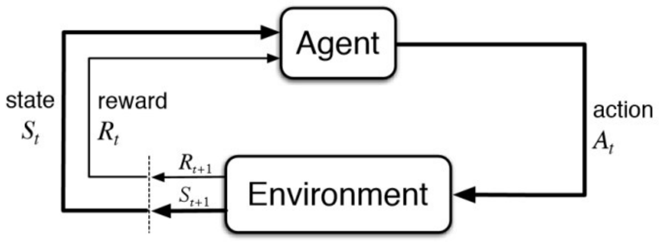


Figure 4: Reinforcement Learning

In reinforcement learning, the end goal for the Agent is to discover a behavior (a Policy) that maximizes a reward.

There are different types of reinforcement learning:

6.1.1 Q-Learning

In Q-Learning, Q values are iteratively updated in the environment based on actions and states using value iteration. Though it is good for an environment with small space, it is not feasible for a gaming environment.

6.1.2 Deep Q Learning

Since there are enormous states in a game environment, we can't use a Q-value table to get the best action sequence. Instead, we use Deep Q Learning. In Deep Q learning, we use neural networks(CNN) to approximate the Q-function for each state action pair in a given environment. But DQN are sensitive to hyperparameter changes and suffer from instability problems.

6.1.3 Proximal Policy Optimization

Proximal Policy Optimization (PPO) learns online unlike experience replay by Deep Q-Networks. PPO strikes a balance between ease of implementation, sample complexity, and ease of tuning, trying to compute an update at each step that minimizes the cost function while ensuring the deviation from the previous policy is relatively small. Unlike DQN, PPO doesn't suffer from instability problems [10]. PPO Objective function is defined in the Fig. 5

$$L^{CLIP}(\theta) = \hat{E}_t[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\varepsilon, 1+\varepsilon)\hat{A}_t)\ ]$$

- θ is the policy parameter
- $\hat{E}_t$ denotes the empirical expectation over timesteps
- r_t is the ratio of the probability under the new and old policies, respectively
- $\hat{A}_t$ is the estimated advantage at time t
- ε is a hyperparameter, usually 0.1 or 0.2

Figure 5: PPO Objective Function

We have decided to move ahead with PPO due to following reasons:

- Training data is generated based on the current policy rather than relying on static data. If the agent learned a poor policy in the first run, then it would cause a cascading effect as it keeps on learning. Moreover, due to the continuous change of rewards and observations, learning is not stable. PPO overcomes this drawback in addition to not being very sensitive to hyper parameter tuning.
- It involves collecting a small batch of experiences interacting with the environment and using that batch to update its decision-making policy.

6.2 Imitation Learning

The idea behind imitation learning is to mimic human behaviour to perform a given task. An expert demonstrates how to perform a task and the agent is trained to follow the expert's way of carrying out the task thereby, leveraging human intelligence efficiently. Under the hood, the agent tries to learn a mapping function between observations and actions and tries to learn the optimal policy by following, imitating the expert's decisions. This decreases the need to programmatically teach the agent by reward function to perform a task.

6.2.1 Behavioral Cloning

This is the simplest form of Imitation Learning where the agent tries to replicate the expert's actions as it is using supervised learning. Since we want the agent to do more than just mimic the expert i.e, explore the arena, understand the policy, judge the scenario better while parking etc, we decided not to move forward with this technique.

6.2.2 GAIL (Generative Adversarial Imitation Learning)

This state-of-the-art, model free imitation learning method aims to directly extract a policy from data, as if it were obtained by reinforcement learning following inverse reinforcement learning. We show that a certain instantiation of our framework draws an analogy between imitation learning and generative adversarial networks, from which we derive a model-free imitation learning algorithm that obtains significant performance gains over existing model-free methods in imitating complex behaviors in large, high-dimensional environments. Because of these advantages, we have decided to use GAIL in training our agent [11].

7 Environment Setup

The open source game used in this project is downloaded from the link here. Unity Environment has been used for further game modifications and for training the agent as it provides a good support for PPO as well as imitation learning. The ml-agents package from Unity is open source and helps developers develop intelligent game playing agents. Our team members had both Mac and Windows laptops and hence we had to set up the game environment in both the machines.

7.1 Unity Setup

7.1.1 Mac

- Unity Hub version - 2.4.2
- Unity version - 2020.2.1
  The open source Car Parking Game was developed in the 2019.4.1 version of Unity. Due to compatibility issues with Mac Big Sur OS, the game was migrated to the latest version of Unity on Macintosh.
- mlagents package version for training - 1.0.6
- Python - 3.9

7.1.2 Windows

- Unity Hub version - 2.4.2
- Unity version - 2019.4.1
  The open source Car Parking Game was developed in the 2019.4.1 version of Unity. Hence even though there is a latest version of Unity available, we stuck to the older version of Unity in which the game was developed.
- mlagents package version for training - 1.0.6
- Python - 3.8.3
  Ml-agents package does not work with the 3.9 version of python due to version compatibility issues in Windows machines. Hence, in Windows machines, unlike Mac, we used the 3.8.3 version of python.

7.2 Google Colab

- Since training takes hours on local machines, it is a good option to switch the training process onto cloud architecture so that multiple training instances could be run. Moreover, since google colab provides a wonderful GPU/TPU support, it would be a great speed up for the training process [12].
- Google colab setup has following steps:
  - Clone ml-agents repository onto google colab environment
  - Install and activate ml-agents environments
  - Create a headless linux server build of the game. This required us to add support for Linux builds in Windows machines(this feature is not supported by Mac OS)
  - Deploy headless server build and provide respective permissions to all the executables.
  - Adding training configuration details to a file
  - Enable tensorboard
  - Start the training process using the command:
    ```
    !mlagents-learn ./config.yaml --run-id=$run_id --env=$env_name --no-graphics
    ```
  - After training completion, download the summaries and models
- It was noticed that there was not any significant gain in the training time. This was because mlagents isn't set up to leverage GPUs. Unless we have a scene that heavily uses visual observations and does a lot of model updates with a really large neural network, which is not the case with our game, we won't benefit from using a GPU for training the model.
- The link to our colabatory can be found here.

8 Methodology

8.1 Flow Chart

In reinforcement learning, the agent is expected to read observations from the game environment and perform actions on it. A reward, either positive or negative is assigned to the agent based on the action it perfoms on the environment.

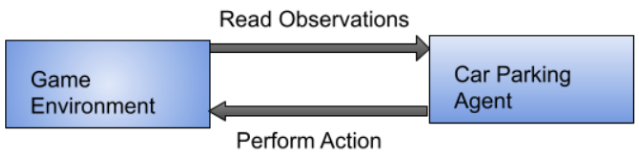


Figure 6: High Level Design

In Level 1 of the game, the observations are the direction to the target and relative distances to the walls, obstacles and target. The game is programmed in such a way that the car always accelerates with a constant speed of 400 units. Changing the direction of the car is the action performed by the agent. It can either turn left, turn right or perform no action which means that the car does not change direction.

In level 2 of the game, Unity's RayPerceptron3D component is used for reading the observations from the game environment. Along with the observations read for the level 1 of the game, the portal location is also read from the game environment to navigate to different storeys. The action space for level 2 has been improved to perform two actions unlike level 1 where we perform only one action. First, the car can either accelerate forward or perform no action, meaning the car stays in its current position. Second, similar to level 1, the car can change direction (either turn left, turn right or no change in direction) to navigate to the parking spot.

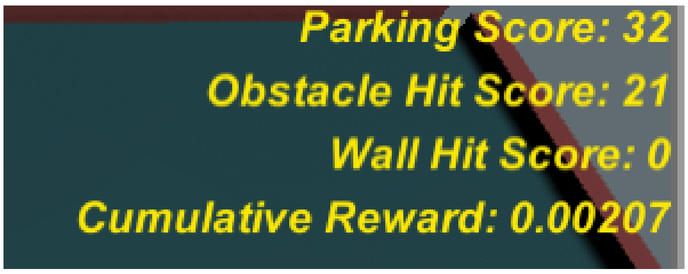
8.2 Game Modifications

Some new features have been added to the game and some existing features have been modified to ease training and make the game compatible to be trained with the ml-agents package.

8.2.1 Scoreboard

While an AI agent is training, it is important to keep track of the performance of the agent. For the purposes of debugging and performance analysis, following metrics were added to the game.

- Parking Score: Number of times agent parked the car in the highlighted spot
- Wall Hit Score: Number of times agent hit the walls
- Obstacle Hit Score: Number of times agent hit the obstacles
- Cumulative reward: In each episode, we try to see if the agent is moving towards the goal (positive rewards) or away from the goal (negative rewards). This helps in debugging different scenarios and also to identify if the agent is making correct inferences.



Parking Score: 32
Obstacle Hit Score: 21
Wall Hit Score: 0
Cumulative Reward: 0.00207

Figure 7: Scoreboard

8.2.2 Navigation

The open source game from Unity had touch screen controls, to change the direction of the car. The touch screen controls have been modified to keyboard controls. The right arrow or "d" button on the keyboard is used to change the direction to right and the left arrow or "a" button is used to change the direction to left. In level 2 of the game, more controls for driving the agent have been added. Along with changing the direction of the car, the agent can also accelerate or brake and stay in the current location. The forward arrow or "w" button is used to accelerate the agent and when this key is released, the agent stops accelerating in the forward direction. Using keyboard controls we can leverage the mlagents package for training the agent which cannot be done with touch screen controls as there is no support for touch screen in ml-agents package currently.

8.2.3 Fixed Starting Point

In the open source game, the car that should be parked can start at any random location. We have fixed the starting point to a single location for level 1 and level 2 of the game to reduce the training time. A random start location exponentially increases the training time and the hardware of our machines does not support training for such longer times.

8.2.4 Fixed but Multiple Goal Points

In the modified game, there are some predefined goal locations where the agent is expected to park the vehicle. For each episode, a goal location is randomly selected from the set of the 3 goal locations for level 1 and 1 or 2 goal locations for level 2 for the agent to park. The higher the number of goal locations, the more is the training time for training the agent. Since, level 2 is tremendously complex in terms of environment and moving obstacles, it was trained for a single goal location for two storeys.

Environment	**Training time to learn correct policy to park**
Level 1 (one storey, fixed obstacles, 3 goals)	12 hours (5M steps)
Level 2 (one storey, moving obstacles, 2 goals)	24 hours (10M steps)
Level 2 (two storeys, moving obstacles, 1 goal)	6 days (50M steps)

Table 1: Approximate training time to learn parking policy based on environments using PPO and GAIL

8.2.5 Collision Objects

In the open source game, only the obstacles in the arena are treated as collision objects. The game was modified to treat both the walls and obstacles as collision objects. Whenever the agents collides with either the wall or an obstacle, a negative reward is assigned to the agent, the episode ends and the agent is starting again at the start location to park at the same highlighted spot.

8.3 Scripts

8.3.1 Car Agent script

The Car agent script extends the Agent class from mlagents package and is responsible for collecting observations, passing them to its decision-making policy, and receiving an action vector in response. There are two different agents created for level 1 and level 2 of the game. The script for level 1 is called CarAgent and the script for level 2 is called CarAgentForLevel2. These scripts override the default behavior of the agent in the mlagents class. The different methods that are overridden in our project are OnEpisodeBegin(), OnActionReceived(), CollectObservations() and Heuristic().

8.3.1.1 OnEpisodeBegin

The OnEpisodeBegin() method is responsible for resetting the parameters used in an episode. In our case, the best distance that the agent has been from the goal location is reset back to 30 units. The reason we chose 30 units is because our arena dimensions are 15 * 15 and the maximum displacement between the start location and the goal location cannot be more than 30 units.

8.3.1.2 OnActionReceived

The OnActionReceived() method implements the actions that the agent can take, such as moving to reach a goal or interacting with its environment. In level 1 of the game, if the agent receives 0 in the actions buffer, then it does change the direction. If it receives 1, then it turns right and if it receives 2, then it turns left.

In level 2 of the game, the action buffer size is increased to 2 because the agent performs 2 actions. The first value in the action buffer is used for changing direction similar to level 1 and the second value is for driving the agent. If the value in the second location is 0, then the agent does not move forward and if the value is 1, then the agent accelerates with an acceleration of 100 units. The speed for navigating forward is 800 units. Rewards are given to the agent for every action performed and this is explained in detail in the Reward System section.

8.3.1.3 CollectObservations

The CollectObservations() method is used to collect the vector observations of the agent for every step. The agent observation describes the current environment from the perspective of the agent. The observations collected by the agent are discussed in detail in the Agent Observation Space section.

8.3.1.4 Heuristic

An agent calls this Heuristic() function to make a decision when you set its behavior type to HeuristicOnly or Default. This method has no purpose while training the agent but can be used for debugging purposes while not training.

8.3.2 Game Controller script

The Game Controller script is responsible for controlling the game. It resets the game environment after the end of an episode, randomly chooses a new goal location from the set of available goal locations for the next episode, destroys the previously parked car instance from the previous episode and resets the reward to 0.

8.3.3 Reward System

The Reward System is the key to train an agent properly to learn a policy. Different reward functions have been shaped for both the algorithms by trial and error while training the agents. Table. 2 summarizes the rewards and penalties that were assigned to the agent for every action it takes.

S.No	**Condition**	**Level 1 Reward [PPO]**	**Level 1 Reward [PPO + GAIL]**	**Level 2 Reward [PPO + GAIL]**
1.	Hit the wall [Episode Ends]	-0.5	-0.5	-0.5
2.	Hit an obstacle [Episode Ends]	-0.5	-0.5	-0.5
3.	Car Parked [Episode Ends]	+5	+5	+5
4.	Within 2.5 units of distance to the goal location	+0.00008	+0.00003	+0.00003
5.	Best current distance to the goal location	+0.00002	+0.00002	+0.00002
6.	Moving towards the goal but not the best distance to the goal in the current episode	-0.00004	+0.00001	+0.00001
7.	Moving away from the goal	-0.00008	-0.00002	-0.00002
8.	Within 2 units of distance to the wall	-0.005	-0.005	-0.005
9.	Within 2 units of distance to the obstacle	N/A	-0.005	-0.005
10	Move through portal towards target	N/A	N/A	+0.5
11.	Move through portal away from target	N/A	N/A	-0.1

Table 2: Reward System

8.3.4 Agent observation space

8.3.4.1 Level 1 Observation Space

For level 1, an observation space of 27 was added for the agent, which consists of the following 9 parameters, each consisting of x,y and z coordinates:

- *Direction to the target* : Relative and normalized Vector3 value
- *Relative distance to the target* : Normalized Vector3 value
- *Relative distance to obstacle 1* : Normalized Vector3 value
- *Relative distance to obstacle 2* : Normalized Vector3 value
- *Relative distance to obstacle 3* : Normalized Vector3 value
- *Relative distance to wall 1* : Normalized Vector3 value
- *Relative distance to wall 2* : Normalized Vector3 value
- *Relative distance to wall 3* : Normalized Vector3 value
- *Relative distance to wall 4* : Normalized Vector3 value

8.3.4.2 Level 2 Observation Space

For level 2, a combination of CollectObservations() and RayPerceptron 3D is used for collecting the data from the game environment.

The CollectObservations() method is used to identify

- *Direction to the target* : Relative and normalized Vector3 value and
- *Relative distance to the target* : Normalized Vector3 value

RayPerceptionSensorComponent3D is the new way to use raycast observations in Unity ML-Agents. RayPerceptionSensorComponent automatically adds observations to your agent, so you no longer need to add any RayPerception code to CollectObservations(). The following figure shows the configuration of RayPerceptionSensorComponent3D for our agent.

Rays Per Direction specifies how many raycasts will be performed on either side of the centerline. By default, there will be at least one raycast that points straight forward.

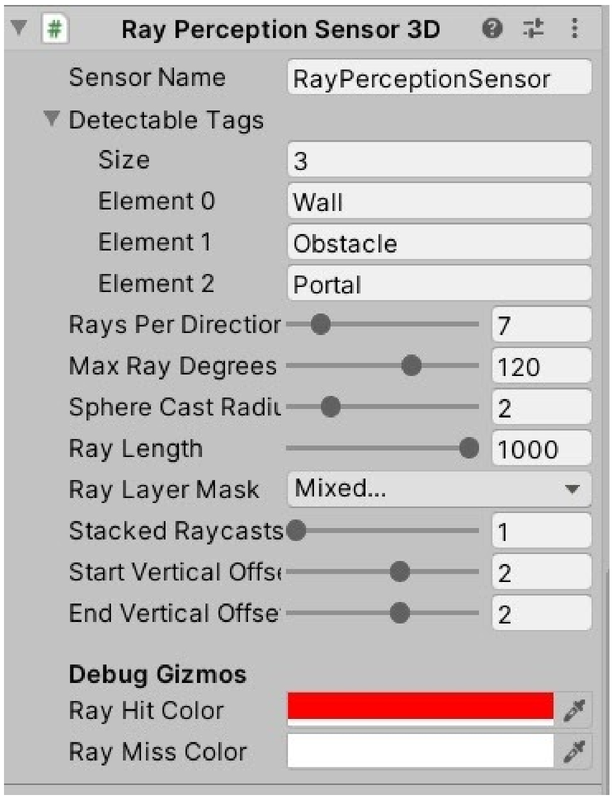
Max Ray Degrees specifies the angle at which to spread out the raycasts. A value of 120 degrees will result in rays spread over 60 degrees to the left and right of the centerline.

Sphere Cast Radius specifies how large of a sphere to cast along the ray. Rather than using a thin line for detection, anything within this radius of the ray will count as a hit.

Ray Length specifies how far to project each ray for sight. The value of 1000 means it will not see anything beyond 1000 meters.

8.3.5 Decision Requester

The DecisionRequester component provides a way to trigger the agent decision making process. A DecisionPeriod is defined as the frequency with which the agent requests a decision. For level 1, we set the DecisionPeriod to 3 which means that the Agent will request a decision every 3 Academy steps. For level 2, we set the decision period to 1 so that the agent makes a decision at every step due to high variability of the environment.



Ray Perception Sensor 3D

Sensor Name	RayPerceptionSensor
Detectable Tags	
Size	3
Element 0	Wall
Element 1	Obstacle
Element 2	Portal
Rays Per Directior	7
Max Ray Degrees	120
Sphere Cast Radiu	2
Ray Length	1000
Ray Layer Mask	Mixed...
Stacked Raycasts	1
Start Vertical Offs	2
End Vertical Offse	2

Debug Gizmos

Ray Hit Color	🟥
Ray Miss Color	⬜

Figure 8: RayPerceptionSensorComponent3D Configuration

8.3.6 Behavior Parameters

How a Policy makes its decisions depends on the Behavior Parameters associated with the agent.

8.3.6.1 Vector Observations

- "Space Size" represents the observation space size. For level 1 this value was 27. For level 2, it is 6.
- "Stacked Vectors" represents the number of previous vector observations that will be stacked and used collectively for decision making. This value was set to 1 for level 1. For level 2, the value is set to 3.

8.3.6.2 Actions - The actions performed by the agent

Space Type - Actions for an agent can be either Continuous or Discrete. In our case the action space type is Discrete.

Action	Action Buffer Value	Action Description	Applicable for Level
Steer	0 1 2	No change in the direction Take a right turn Take a left turn	Level 1, Level 2
Drive	0 1	No acceleration Accelerate	Level 2

Table 3: Actions for Level 1 and Level 2

8.4 Training Process

We used the ml-agents package provided by Unity to train our model. The training process required following steps:

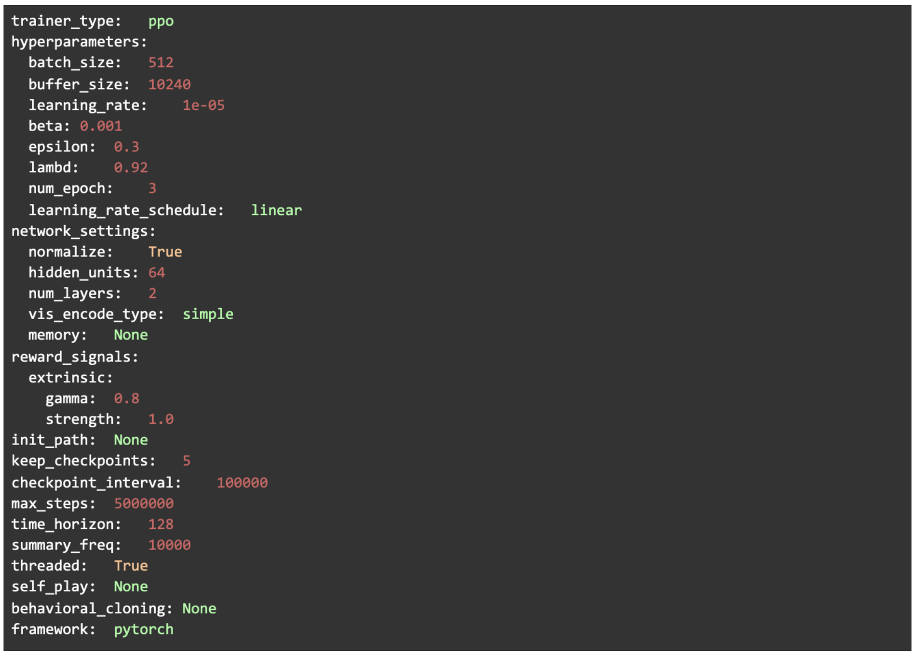
1. Activate the virtual environment consisting of all the dependencies
2. Changes to our Batmobile (Car Agent) prefab: Adding decision requester and making decision process to heuristic only.
3. Create a configuration file
4. Start training process using command:
   mlagents-learn <path to config yaml file> –run-id= <unique run id>
5. After training is completed, visualize the results at the tensorboard

8.4.1 Training for Level 1

Level 1 consists of fixed obstacles and single storey. Two variants of the game were trained using Proximal Policy Optimization algorithm for 1 goal location and Proximal Policy optimization combined with Generative Adversarial Imitation Learning and LSTM for 3 goal locations. The hyperparameters and results are described in the following sections.

8.4.1.1 Training the agent with PPO

The agent was trained using following parameters using Proximal Policy Optimization:



```
trainer_type:   ppo
hyperparameters:
  batch_size:   512
  buffer_size:  10240
  learning_rate:    1e-05
  beta: 0.001
  epsilon:  0.3
  lambd:    0.92
  num_epoch:    3
  learning_rate_schedule:   linear
network_settings:
  normalize:    True
  hidden_units: 64
  num_layers:   2
  vis_encode_type:  simple
  memory:   None
reward_signals:
  extrinsic:
    gamma:  0.8
    strength:   1.0
init_path:  None
keep_checkpoints:   5
checkpoint_interval:    100000
max_steps:  5000000
time_horizon:   128
summary_freq:   10000
threaded:   True
self_play:  None
behavioral_cloning: None
framework:  pytorch
```

Figure 9: Hyperparameters for PPO

We decided to use Long short-term memory recurrent network architecture since we have a discrete action space, and LSTM networks perform better in such conditions. Also, we decided to keep the learning rate low(1e-05), batch and buffer size high, for more stable updates. To make the agent explore more, we kept epsilon as 0.3 so that more deviations are allowed from the learned policies.

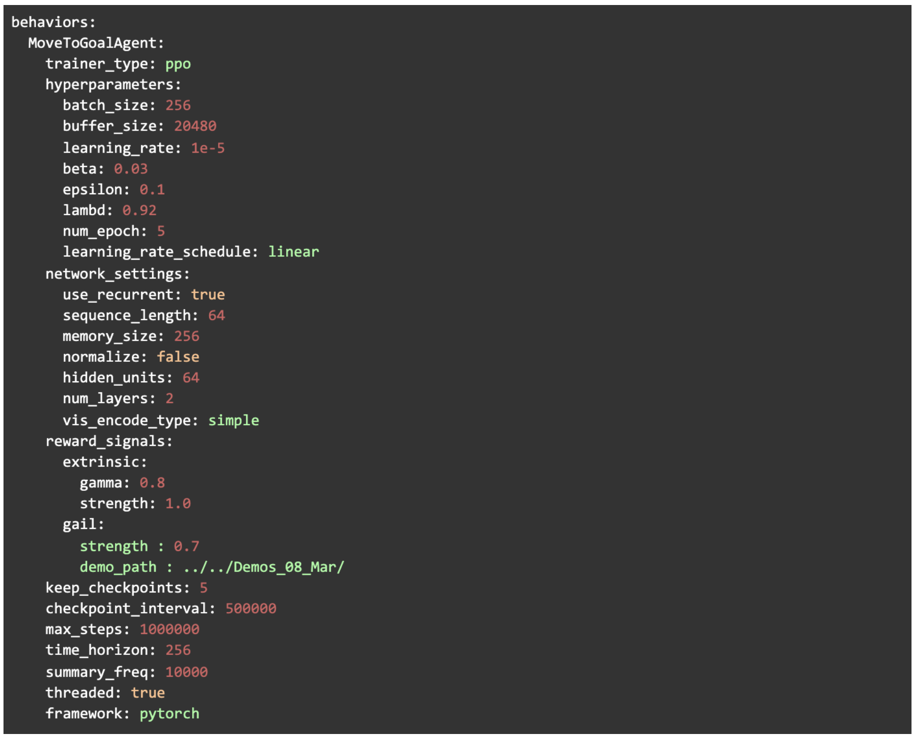
8.4.1.2 Training the agent with PPO + Imitation learning(GAIL)

8.4.1.2.1 Expert Demonstrations

GAIL uses two neural networks effectively. One is the *"generator"* that the agent will use to generate new data points and the other is called the *"discriminator"*. The discriminator determines if the actions or observations are by the agent or the expert demonstration. If the discriminator classifies that an action came from the agent, a reward is assigned. This way, two optimizations are working in parallel to give us a faster and better agent. To record a demonstration, add a "Demonstration Recorder" component to the agent and record a good number of demonstrations for the agent to learn the policy well.

8.4.1.2.2 GAIL parameters

The agent was trained with PPO in combination with GAIL using the following hyperparameters. In addition to PPO parameters, GAIL strength and path to expert demonstrations folder is added.



```
behaviors:
  MoveToGoalAgent:
    trainer_type: ppo
    hyperparameters:
      batch_size: 256
      buffer_size: 20480
      learning_rate: 1e-5
      beta: 0.03
      epsilon: 0.1
      lambd: 0.92
      num_epoch: 5
      learning_rate_schedule: linear
    network_settings:
      use_recurrent: true
      sequence_length: 64
      memory_size: 256
      normalize: false
      hidden_units: 64
      num_layers: 2
      vis_encode_type: simple
    reward_signals:
      extrinsic:
        gamma: 0.8
        strength: 1.0
      gail:
        strength : 0.7
        demo_path : ../../Demos_08_Mar/
    keep_checkpoints: 5
    checkpoint_interval: 500000
    max_steps: 1000000
    time_horizon: 256
    summary_freq: 10000
    threaded: true
    framework: pytorch
```

Figure 10: Hyperparameters for GAIL

8.4.1.3 Trained Neural Network Model

The model has following layers:

1. **Input Layer:** A vector observation consisting of 27 neurons (refer section 8.3.4.1)
2. **Hidden Layers:** There are two hidden layers with sigmoid activation function and one layer with softmax activation function along with multiple functions such as concatenation, multiplication, addition, subtraction and slicing.

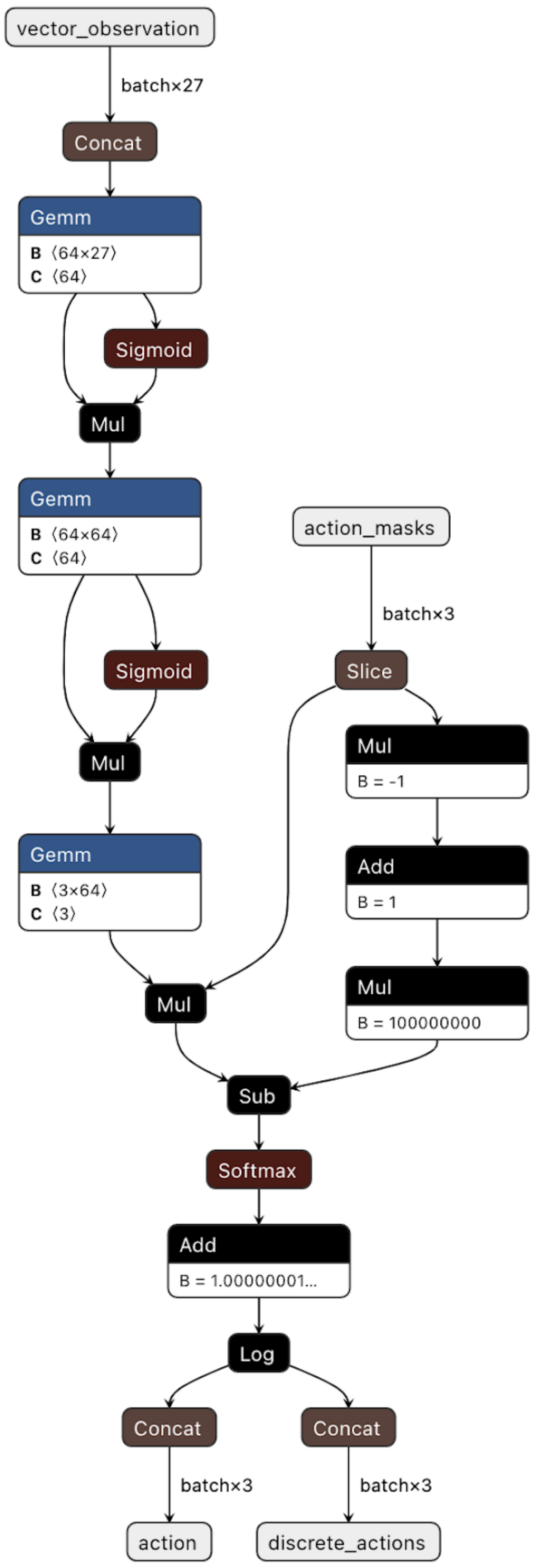


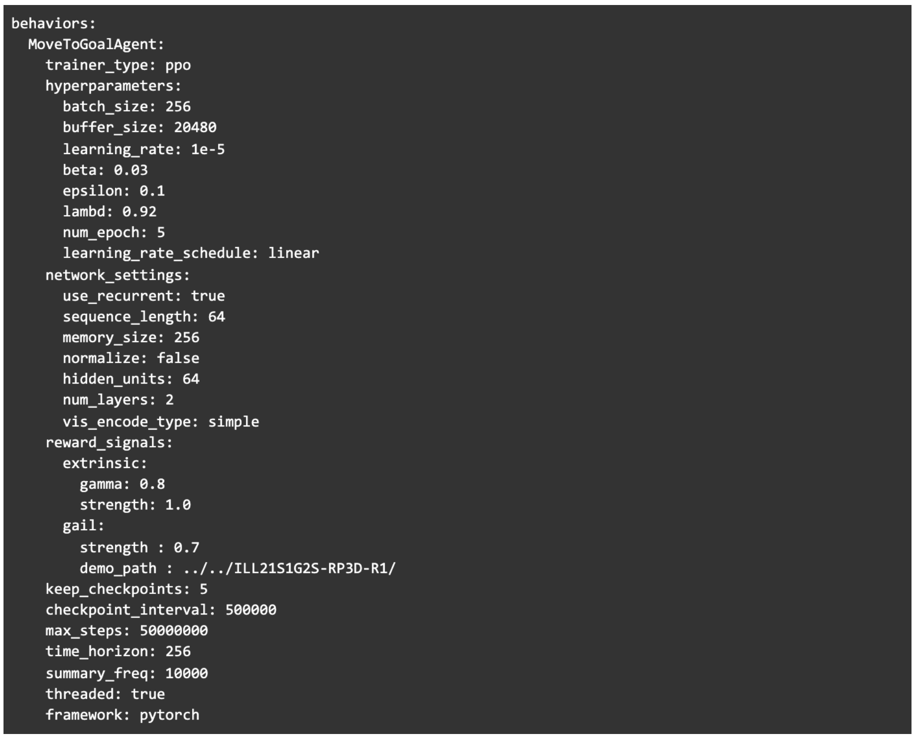
Figure 11: Trained Neural Network Model for Level 1

3. **Output Layer:** The output layer consists of 3 neurons, one for each action (refer section 8.3.6.2)

8.4.2 Training for Level 2

8.4.2.1 Training the agent with PPO + Imitation learning(GAIL)

Since PPO with GAIL showed promising results for Level 1 of the game, we decided to stick with it for Level 2 as well. Though we tried other algorithms such as Soft Actor Critic(SAC) and Curiosity Learning with sparse reward systems, PPO with GAIL along with a dense reward system gave us the best results. After tuning with different hyperparameters, we found an ideal configuration that was giving us most stable results. The final hyperparameter on which we trained our agent for level 2 for 50 million steps is described below:



```
behaviors:
  MoveToGoalAgent:
    trainer_type: ppo
    hyperparameters:
      batch_size: 256
      buffer_size: 20480
      learning_rate: 1e-5
      beta: 0.03
      epsilon: 0.1
      lambd: 0.92
      num_epoch: 5
      learning_rate_schedule: linear
    network_settings:
      use_recurrent: true
      sequence_length: 64
      memory_size: 256
      normalize: false
      hidden_units: 64
      num_layers: 2
      vis_encode_type: simple
    reward_signals:
      extrinsic:
        gamma: 0.8
        strength: 1.0
      gail:
        strength : 0.7
        demo_path : ../../ILL21S1G2S-RP3D-R1/
    keep_checkpoints: 5
    checkpoint_interval: 500000
    max_steps: 50000000
    time_horizon: 256
    summary_freq: 10000
    threaded: true
    framework: pytorch
```

Figure 12: Level 2 Hyperparameters for GAIL

8.4.2.2 Trained Neural Network Model

The layers of the network can be described as below:

1. **Input Layer:** An input observation consisting of 93 neurons (6 from observation collection and 87 from ray perceptrons). Refer to the section 8.3.4.2.
2. **Hidden Layers:** 2 hidden layers consisting of 64 neurons each.
3. **Output Layer:** The output layer consists of 3 neurons, one for each action (refer section 8.3.6.2).

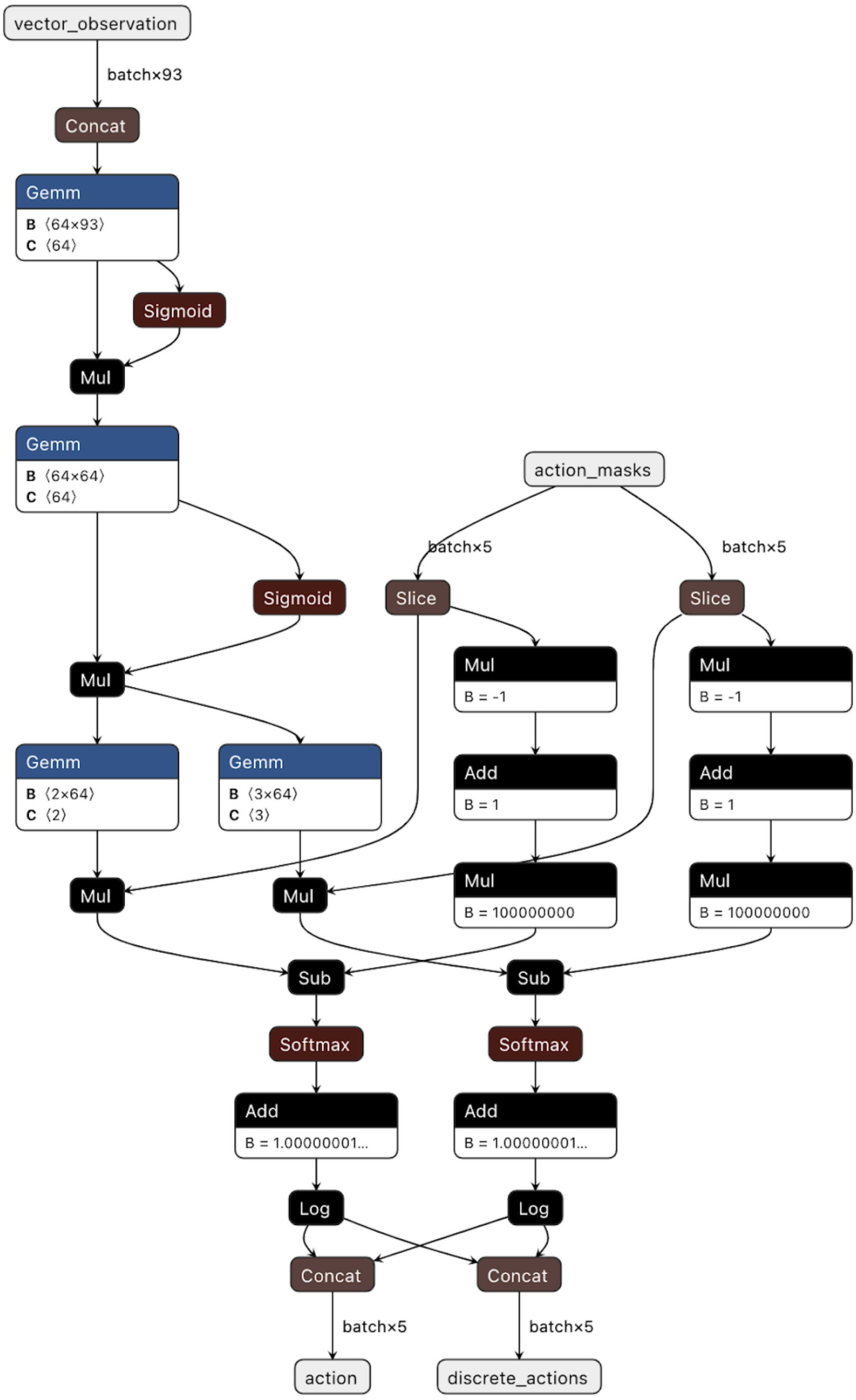


Figure 13: Trained Neural Network Model for Level 2

9 Results and Evaluation

9.1 Cumulative Rewards

For level 1, the cumulative reward as shown in Fig. 14 keeps on increasing with the number of steps for both PPO and PPO with GAIL. This means that our agent learnt a good policy and kept on accumulating more positive rewards over time. We ran our model for 5M steps for both the cases.

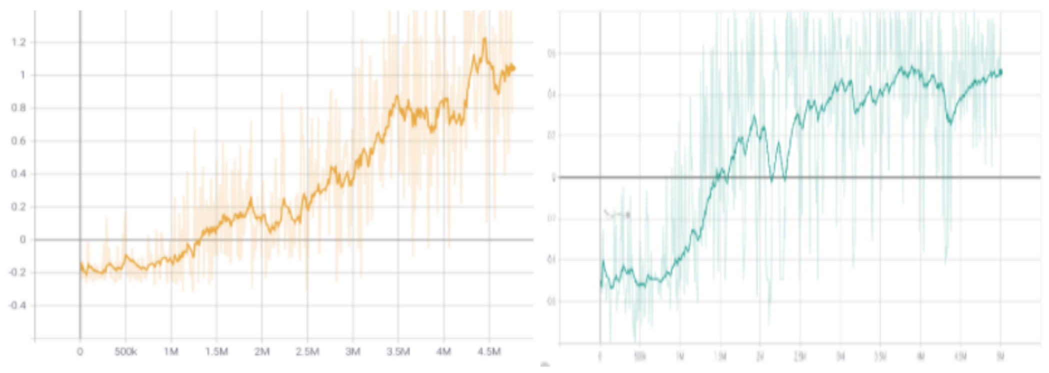


Figure 14: Cumulative Reward for Level 1. Cumulative reward for PPO (left); Cumulative reward for PPO with GAIL (right)

For level 2, it required significantly more steps to learn a good and stable policy because the obstacles were not in a fixed location and the environment changed to two storeys. We trained the agent for 50M steps over a span of 4 days on our local machine and in the end, the agent learnt to go through the portal to the second floor and find the highlighted spot to park the car. The cumulative reward as shown in Fig. 15 is an increasing function as expected for good agents.

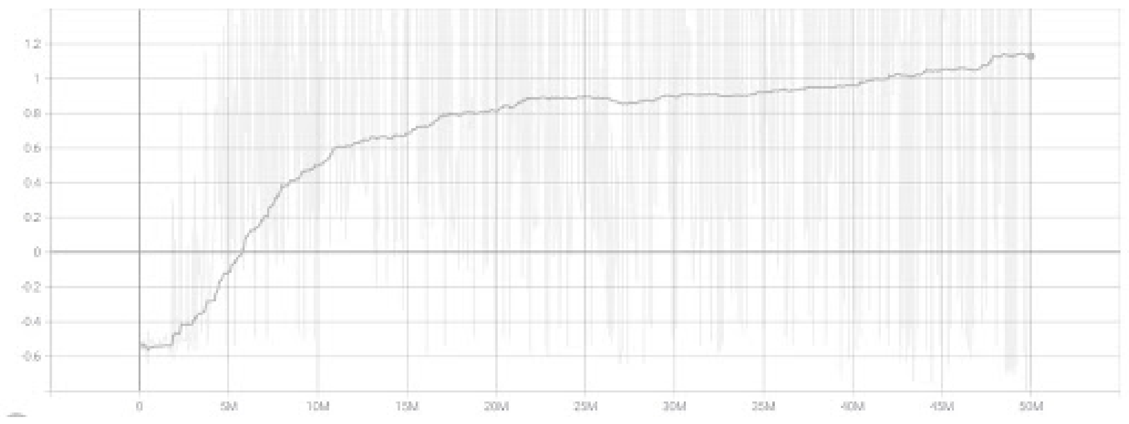


Figure 15: Cumulative Reward for Level 2

9.2 Policy Loss

The policy loss fluctuates throughout the training process, but it is less than 1. It means that the agent is trying to learn the optimal policy. For both level 1 as in Fig. 16 and level 2 as in Fig. 17, policy loss fluctuated but was always stayed less than 1.

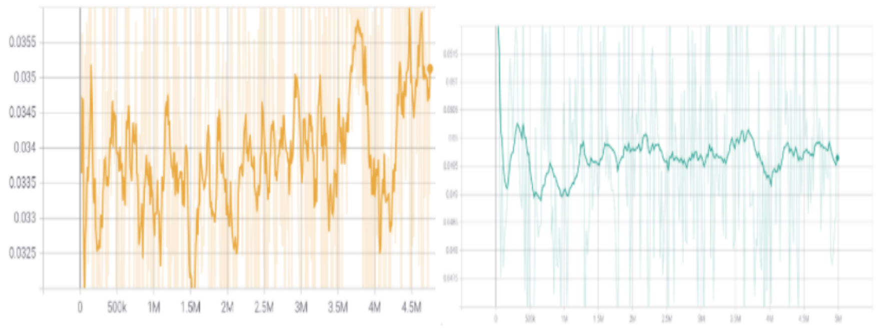


Figure 16: Policy Loss for Level 1

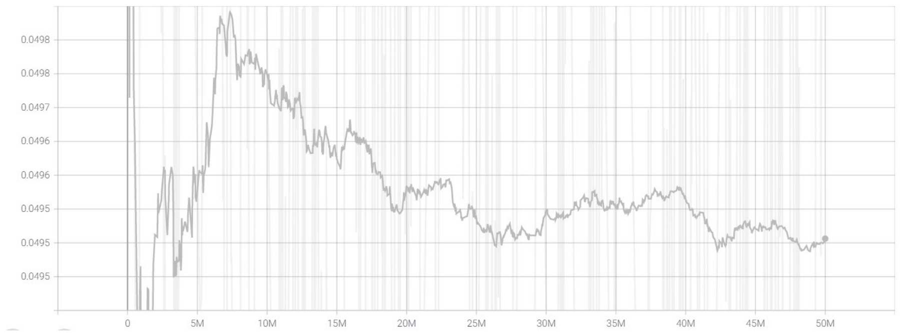


Figure 17: Policy Loss for Level 2

9.3 Entropy

Entropy shows how randomly the decisions of the model are. It should slowly decrease during a successful training process. The entropy of our system is decreasing continuously over steps for both the algorithms. Initially the decisions of the agent are random, but as it learns an optimal policy, the decisions become more informed. Fig. 18 and Fig. 19 outlines the entropy for level1 and level2 respectively. In both the cases, there is a steep decrease in the entropy of the models.

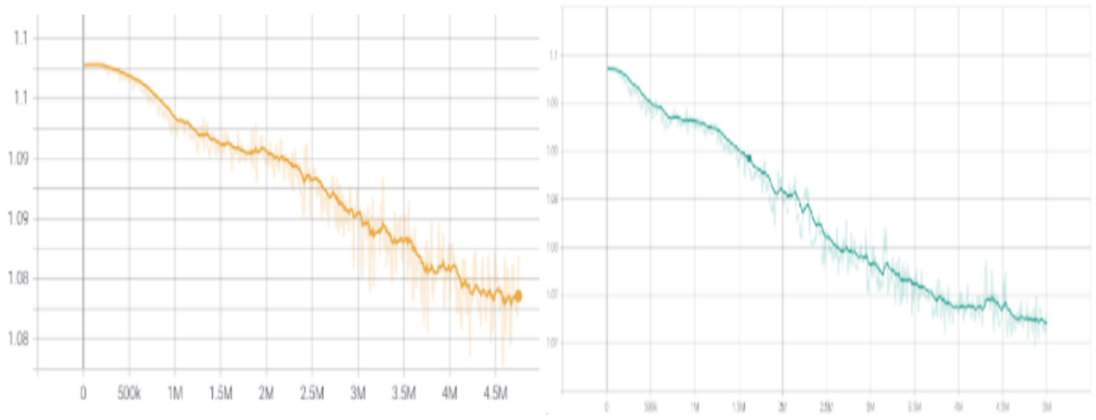


Figure 18: Entropy for Level 1

9.4 PPO with GAIL Value Loss

The mean loss of the value function update correlates to how well the model is able to predict the value of each state. At first, it increases since the agent is trying to learn. Once the reward stabilizes, it decreases [10]. Fig. 20 and Fig. 21 depicts the same for level1 and level2 respectively.

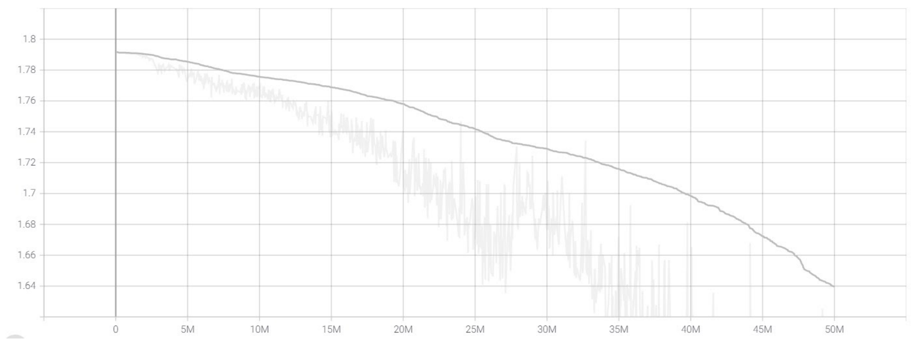


Figure 19: Entropy for Level 2

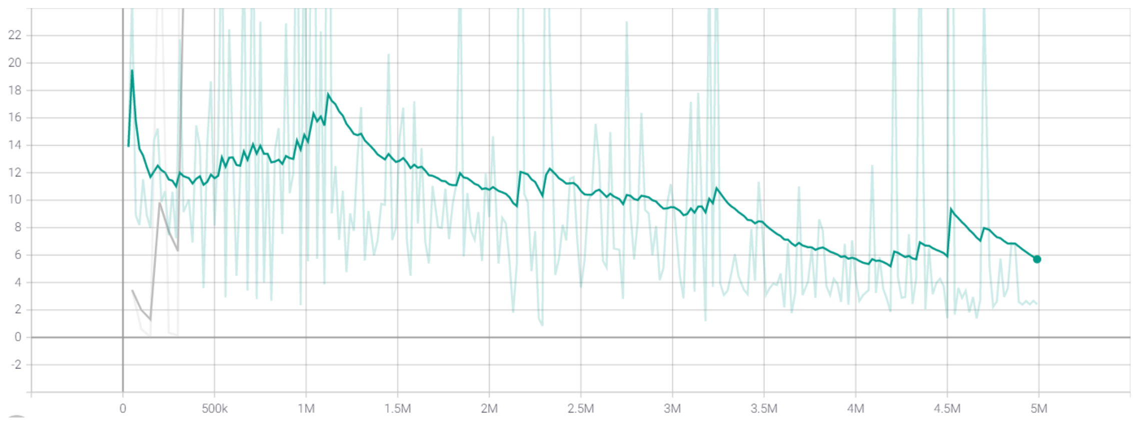


Figure 20: PPO with GAIL Value Loss for Level 1

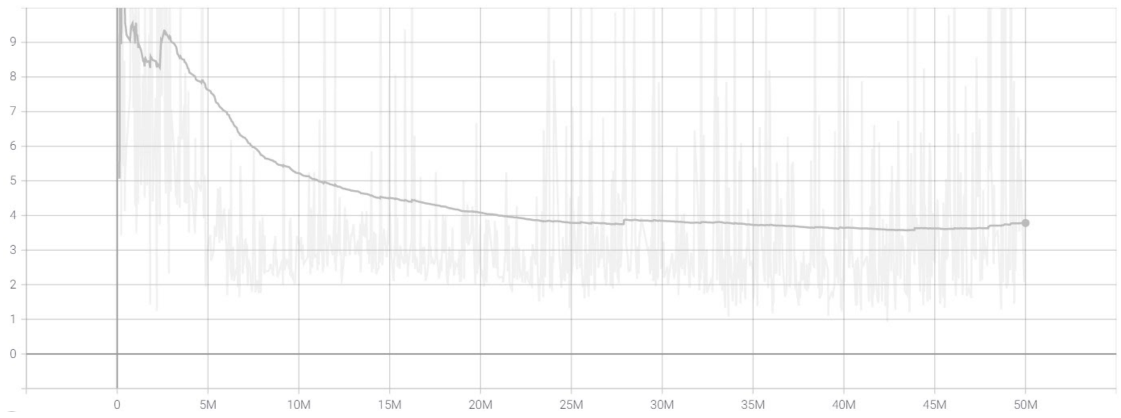


Figure 21: PPO with GAIL Value Loss for Level 2

9.5 GAIL Loss

The mean magnitude of the GAIL discriminator loss corresponds to how well the model imitates the demonstration data [13]. GAIL Loss for level1 and level2 are shown in the Fig. 22 and Fig. 23 respectively.

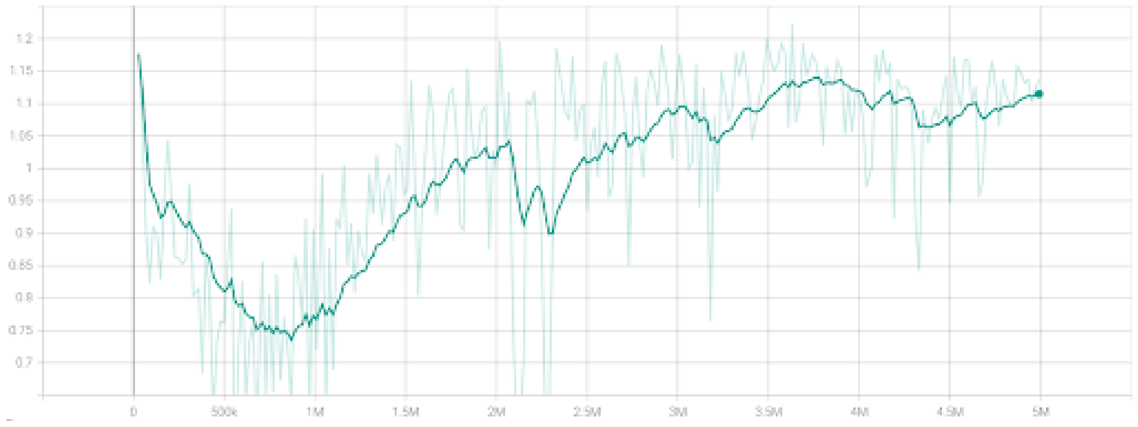


Figure 22: GAIL Loss for Level 1

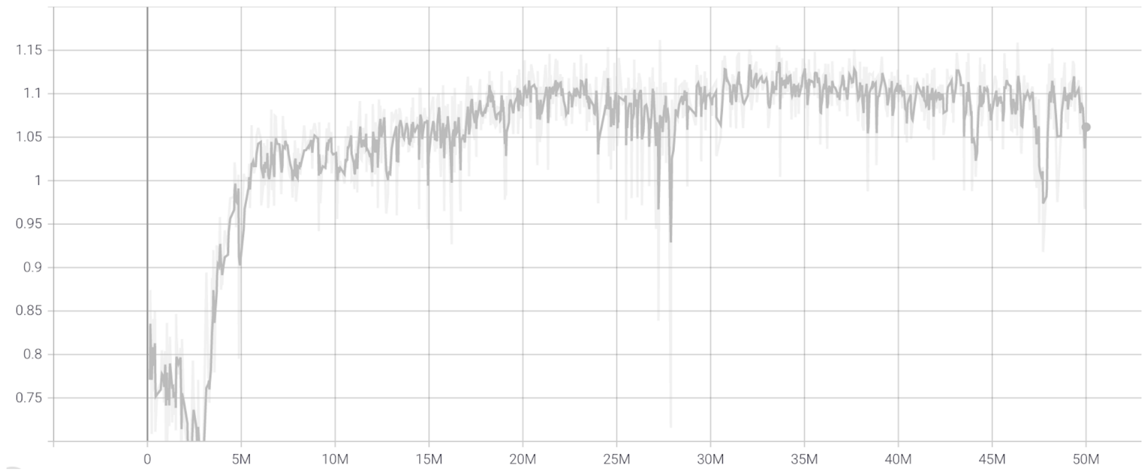


Figure 23: GAIL Loss for Level 2

9.6 Evaluation Statistics

Due to significant differences in the environment complexity, reward systems, training time and hyperparameters, it is not possible to compare Level 1 and Level 2 statistics together. However, we compare Level 1 models together and present Level 2 statistics. The agent trained in Level 2 is far superior to the agent trained on Level 1. So, we present the statistics differently.

9.6.1 Comparison of Level 1 Models

For testing the accuracy of the trained model, we calculated different statistics for the model we trained and logged the results shown in Fig. 24. Train spots are the exact spots for which the agent is trained. Test spots represent a different set of spots unlike the training spots and are used to infer how well the agent has learned the optimal policy for those spots for which the agent is not trained.

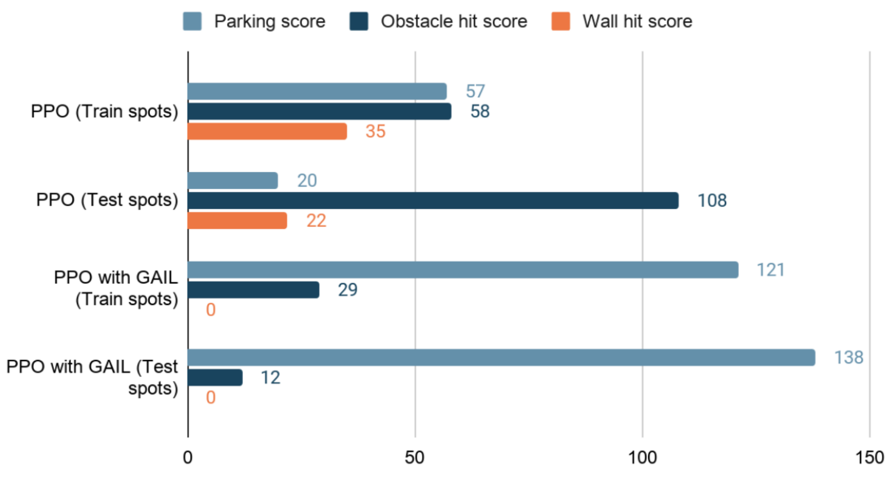


Figure 24: Inference statistics for Level 1

9.6.2 Level 2 Statistics

For evaluating the accuracy of the trained model, the model was allowed to make inferences on the same spots as training spots as well as new locations (known as test spots). Two different types of test spots were defined, placed either on storey 1 or storey 2. The results are presented in Fig. 25.

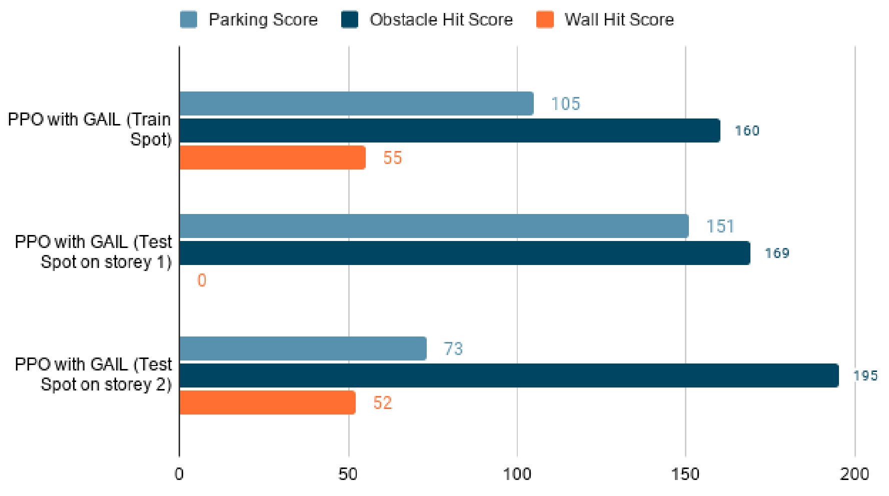


Figure 25: Inference statistics for Level 2

10 Trials and Errors

10.1 Curiosity Learning

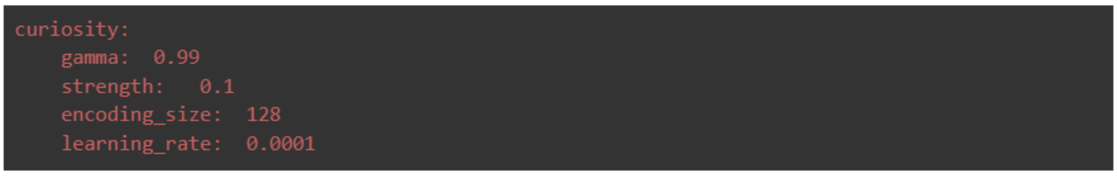
Reinforcement Learning is based on the reward hypothesis, which claims that each goal is represented by the maximization of rewards. The current issue with extrinsic reward systems is that this feature is hard coded by a human, which is inefficient. Curiosity-Driven learning creates an intrinsic reward function for the agent called Curiosity; i.e the reward system is generated by the agent itself. This type of reward system bolsters the agent to predict the consequence of its own actions.

Intrinsic Curiosity module [14] is used to calculate curiosity which constitutes two models-

- Inverse model which helps learn feature representation of a state and its next state.
- Forward Dynamics model that generated the predicted feature representation of next state.

So, a Curiosity learning agent will explore the environment better and favours areas where the agent has spent less time thereby boosting prediction error [15].

Our use case adopts a dense reward system instead of a sparse reward system, which is a requirement in Curiosity learning. So, it did not work. We used the following parameters for Level 1 and ran for 2M steps:



```
curiosity:
    gamma:  0.99
    strength:   0.1
    encoding_size:  128
    learning_rate:  0.0001
```

Figure 26: Curiosity Learning parameters

We used Curiosity Learning in Level 2 as well with a different set of parameters and ran for 5M steps. The reward system was also modified to a sparse reward system so as to accommodate Curiosity-driven learning. But the training results were not satisfactory. The agent displayed procrastination-like behaviour. So, we decided not to go forward with it.

In order for the agent to explore the arena even better, a large-scale Curiosity-driven learning without extrinsic rewards [16] has also been implemented. Due to the lack of any extrinsic rewards, the agent is expected to explore the arena and create a much useful reward system for itself. But, the results of the training were basic. Even though the agent explored the arena, it was not able to learn how to avoid obstacles. So, it was not considered any further.

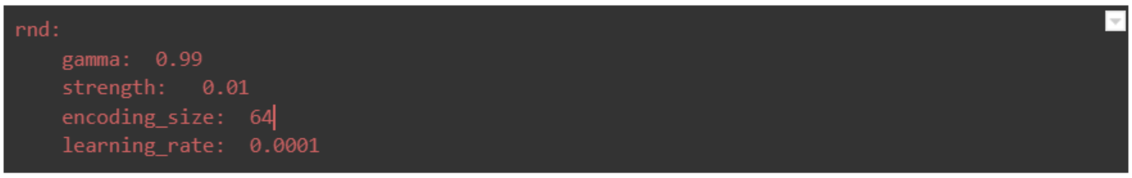
10.2 PPO with RND

Curiosity Learning suffers from the drawback of procrastination in stochastic and noisy environments. So, we decided to experiment with Random Network Distillation, which consists of two networks-

- A target network with fixed, randomized weights which is never trained.
- A prediction network that tries to predict the target network's output.

So, an RND agent tries to predict a state's feature based on a random network, thus removing dependence on previous state, which helps it perform better in case of random stochastic environments [17].

For our use case, since we had a dense reward system and our environment was quite deterministic, so it didn't work. We used following RND parameters and ran for 2M steps:



```
rnd:
    gamma:  0.99
    strength:   0.01
    encoding_size:  64
    learning_rate:  0.0001
```

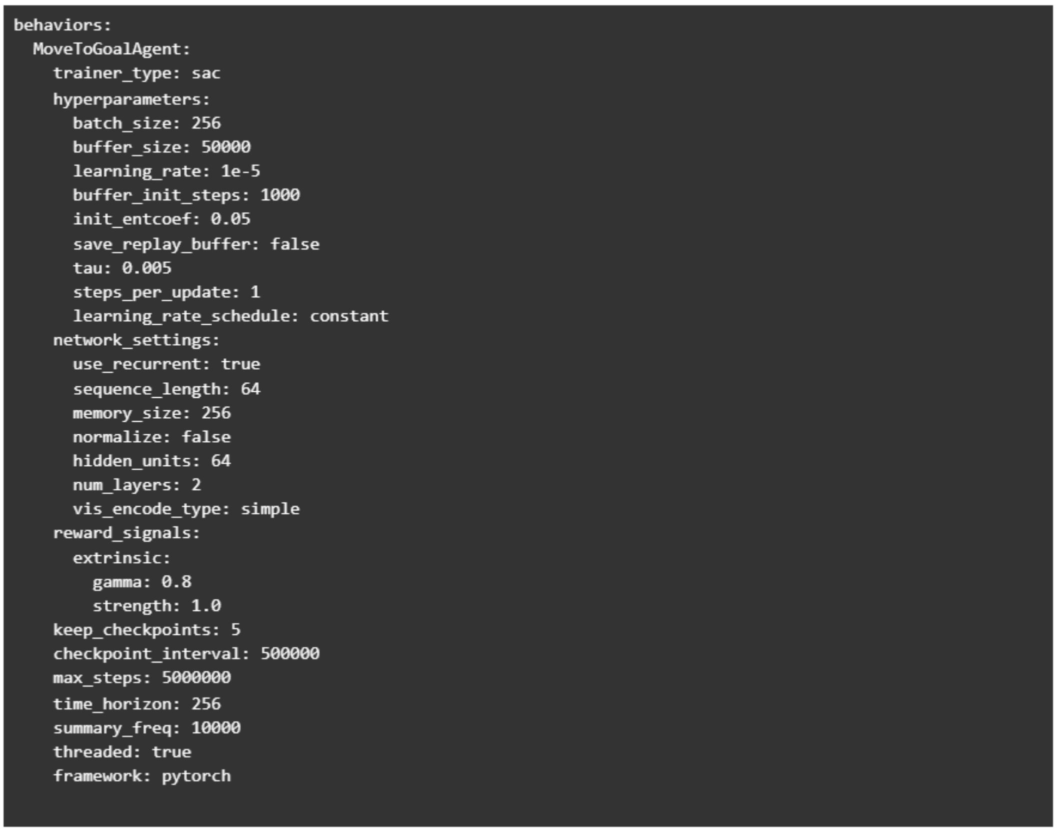
Figure 27: RND parameters

For Level 2 a Sparse reward system was designed to better accommodate the Random Network Distillation with Curiosity Learning. With a new set of parameters the agent was trained for 7M steps. The agent was able to overcome the deficits of pure Curiosity learning. The results obtained were also promising, but there was no increase in the Cumulative reward graph. The graph was sporadic.

10.3 Soft Actor Critic

Traditional model-free reinforcement learning algorithms suffer from challenges like high sample complexity and brittle convergence. Soft actor-critic is an off-policy deep reinforcement learning algorithm i.e, the algorithm improves a policy that is different from the behaviour policy, where the actor aims to maximize expected reward while also maximizing entropy [18].

In our use case, the algorithm took significantly high training time of more than 20hrs for 2 million steps which became almost computationally infeasible on a local machine. It has been observed that the cumulative reward wasn't stable since the algorithm tried to explore too much. Hence, it was decided to not go ahead with this model. The hyperparameters used for training the agent with SAC are as follows.



```
behaviors:
  MoveToGoalAgent:
    trainer_type: sac
    hyperparameters:
      batch_size: 256
      buffer_size: 50000
      learning_rate: 1e-5
      buffer_init_steps: 1000
      init_entcoef: 0.05
      save_replay_buffer: false
      tau: 0.005
      steps_per_update: 1
      learning_rate_schedule: constant
    network_settings:
      use_recurrent: true
      sequence_length: 64
      memory_size: 256
      normalize: false
      hidden_units: 64
      num_layers: 2
      vis_encode_type: simple
    reward_signals:
      extrinsic:
        gamma: 0.8
        strength: 1.0
    keep_checkpoints: 5
    checkpoint_interval: 500000
    max_steps: 5000000
    time_horizon: 256
    summary_freq: 10000
    threaded: true
    framework: pytorch
```

Figure 28: SAC parameters

10.4 Other Hyperparameters

Hyperparameter tuning is one of the most important steps in training an agent. There are so many paramters that need to be trained and it is important to identify the correct combination of parameters in order for the agent to learn a good policy. We tried numerous combinations of hyperparameters for both level 1 and level 2 and these are outlined in Table 4 and Table 5 respectively. Few of the important parameters that we tuned include batch size, buffer size, beta, epsilon, layers and hidden units.

S.No	**Algorithm**	**Parameters**	**Number of Steps**	**Results**

1	PPO	batch size = 256
		buffer size = 10240
		learning rate = 0.0001
		beta = 0.01
		epsilon = 0.3
		lambd = 0.92
		normalize = False
		layers = 2
		hidden units = 128
		time horizon = 256
			5M	Decreasing reward function, performance didn't improve over time

2	PPO	batch size = 32
		buffer size = 2048
		learning rate = 1e-05
		beta = 0.01
		epsilon = 0.3
		lambd = 0.92
		normalize = False
		layers = 2
		hidden units = 64
		time horizon = 128
			1M	Rewards didn't increase over time, no drop in entropy

3	PPO	batch size = 32
		buffer size = 3028
		learning rate = 1e-05
		beta = 0.03
		epsilon = 0.1
		lambd = 0.92
		normalize = False
		layers = 2
		hidden units = 64
		time horizon = 256
			1M	Cumulative Rewards kept decreasing over time

4	PPO	batch size = 256
		buffer size = 20480
		learning rate = 1e-05
		beta = 0.03
		epsilon = 0.1
		lambd = 0.92
		normalize = False
		layers = 2
		hidden units = 64
		time horizon = 256
			1M	Cumulative Rewards was constant over time

5	PPO	batch size = 256
		buffer size = 20480
		learning rate = 1e-05
		beta = 0.03
		epsilon = 0.1
		lambd = 0.92
		normalize = False
		layers = 3
		hidden units = 128
		time horizon = 256
			5M	Decreasing reward function, performance didn't improve over time

Table 4: Hyperparameters for Level 1

S.No	Algorithm	Parameters	Number of Steps	Results
1	PPO with GAIL	batch size= 64 buffer size= 409600 learning rate= 1e-5 beta= 0.03 epsilon= 0.3 gail strength=0.7 gail gamma=0.9 lambd= 0.92 normalize= false hidden units= 32 num layers= 2	8M	Increasing reward till 3M then decreasing till 5M then increasing again, Entropy increasing, Parked 50 times while training, Not good around obstacles
2	PPO with GAIL	batch size= 256 buffer size= 20480 learning rate= 1e-5 beta= 0.03 epsilon= 0.1 gail strength=0.7 lambd= 0.92 normalize= false hidden units= 64 num layers= 2	7M	Unstable Cumulative Reward graph, Entropy increased after1M steps but decreased after 4M, Not good around obstacles
3	PPO with GAIL	batch size= 64 buffer size= 409600 learning rate= 1e-5 beta= 0.03 epsilon= 0.3 gail strength=0.7 gail gamma=0.9 lambd= 0.92 normalize= false hidden units= 32 num layers= 2	8M	Cumulative reward never increased in 5M steps, Entropy was not a constantly decreasing curve, Not good around obstacles

Table 5: Hyperparameters for Level 2

11 Conclusion

In conclusion, we have created three agents, one using PPO with LSTM for level 1 and the other using PPO with imitation learning (GAIL) for level 1 and level 2 to navigate an arena, avoid obstacles and park a vehicle in the highlighted parking spot.

For level 1, we noticed that the agent that uses the PPO with GAIL algorithm is trained particularly well to park in multiple parking spaces. PPO alone also does well for a single parking spot but needs significantly more training time to be able to park in multiple parking spots. PPO with GAIL has the model accuracy of 92% while PPO alone has an accuracy of 13% for new parking spots. GAIL also decreases the training time for an agent since it has reference examples for ideal behavior.

For level 2, we used the combination of PPO with GAIL as it significantly reduces the training time for moving obstacles. We noticed that even though the agent has learned a good policy, the accuracy for the training spots is at 32.81% , for test spots at storey 1 the accuracy is 47.18% and for test spots at level 2 the accuracy is at 22.81%. These statistics indicate that with more training time, the agent can significantly improve the accuracy of the model. Since we were training on our local machine with limited memory and processing power and considering the timely constraints of the final exam, we restricted our training to only 50M steps for level 2 which lasted for 6 days. We strongly believe that with more training, the agent statistics will improve significantly.

12 Future Work

Firstly, we have developed a very promising agent for level 1 of the game with static obstacles and the inference statistics show a very high accuracy for the model. For level 2, the accuracy drops to roughly 50% and in future we plan to improve the accuracy of the model for level 2. This will require more training and sophisticated decision making skills by the agent.

Second, the agent should be smart enough to start at any position and park at any other random location in the arena. Currently, the agent only starts at one start point and parks at randomly located spots on the arena. A superior agent could be developed in the future which is agnostic to the environment. Real world scenarios are mercurial and have numerous unaccounted factors, so an agent should be developed that can account for the dynamicity and perform a seamless navigation to the parking spot.

References

[1] senevsemih, “Unity–carparking.” [Online]. Available: https://unitylist.com/p/134h/Unity-Car-Parking

[2] F. M. company, “A decade after darpa: Our view on the state of the art is self-driving cars.” [Online]. Available: https://medium.com/self-driven/

[3] D. Zhang and C. P. Bailey, “Obstacle avoidance and navigation utilizing reinforcement learning with reward shaping,” 2020. [Online]. Available: https://arxiv.org/abs/2003.12863

[4] D. Acharya, W. Yan, and K. Khoshelham, “Real-time image-based parking occupancy detection using deep learning,” *CEUR-WS*, vol. 2087, no. 5, pp. 33–40, 2018.

[5] D. Isele, A. Cosgun, K. Subramanian, and K. Fujimura, “Navigating intersections with autonomous vehicles using deep reinforcement learning,” *CoRR*, vol. abs/1705.01196, 2017. [Online]. Available: http://arxiv.org/abs/1705.01196

[6] A. Folkers, M. Rick, and C. Büskens, “Controlling an autonomous vehicle with deep reinforcement learning,” *CoRR*, vol. abs/1909.12153, 2019. [Online]. Available: http://arxiv.org/abs/1909.12153

[7] S. Aradi, T. Bécsi, and P. Gáspár, “Policy gradient based reinforcement learning approach for autonomous highway driving,” 08 2018, pp. 670–675.

[8] S. OwaisAli Chishti, S. Riaz, M. BilalZaib, and M. Nauman, “Self-driving cars using cnn and q-learning,” in *2018 IEEE 21st International Multi-Topic Conference (INMIC)*, 2018, pp. 1–7.

[9] B. Thunyapoo, C. Ratchadakorntham, P. Siricharoen, and W. Susutti, “Self-parking car simulation using reinforcement learning approach for moderate complexity parking scenario,” in *2020 17th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON)*, 2020, pp. 576–579.

[10] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal policy optimization algorithms,” *CoRR*, vol. abs/1707.06347, 2017. [Online]. Available: http://arxiv.org/abs/1707.06347

[11] J. Ho and S. Ermon, “Generative adversarial imitation learning,” *CoRR*, vol. abs/1606.03476, 2016. [Online]. Available: http://arxiv.org/abs/1606.03476

[12] D. Thumar, “Training ml-agents with google colab.” [Online]. Available: https://dhyeythumar.medium.com/training-ml-agents-with-google-colab-cb166c3dca46

[13] "Using tensorboard to observe training." [Online]. Available: https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Using-Tensorboard.md

[14] D. Pathak, P. Agrawal, A. A. Efros, and T. Darrell, "Curiosity-driven exploration by self-supervised prediction," 2017.

[15] T. Simonini, "Curiosity-driven learning made easy part i." [Online]. Available: https://towardsdatascience.com/curiosity-driven-learning-made-easy-part-i-d3e5a2263359

[16] Y. Burda, H. Edwards, D. Pathak, A. Storkey, T. Darrell, and A. A. Efros, "Large-scale study of curiosity-driven learning," 2018.

[17] Y. Burda, H. Edwards, A. Storkey, and O. Klimov, "Exploration by random network distillation," 2018.

[18] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," 2018.