TEAM: PARK IT RIGHT!

CSCI 527 Applied Machine Learning for Games

Engineering Design Document

A car parking autonomous agent to park at the designated spot navigating through obstacles

Keerthana Nandanavanam nandanav@usc.edu
Krishna Manoj Maddipatla km69564@usc.edu
Nidhi Chaudhary nidhicha@usc.edu
Sumanth Mothkuri mothkuri@usc.edu

TABLE OF CONTENTS

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 the two levels of the game in Unity using different machine learning algorithms by tuning different hyperparameters and evaluating the results on the tensorboard.

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. Self driving/autonomous cars are a big sensation in recent times. 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[2] 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)[3]. 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[4]. The autonomous exploration of a parking lot is simulated and the controls of the vehicles are learned via deep reinforcement learning[5]. 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[6].

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 can be described as follows:

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 that highlighted spot and then navigate towards it through three obstacles that are placed in the center of the arena. Fig1 demonstrates the first level of the car parking game.

Fig1: Level 1 of the Car Parking Game

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 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 storey. The car has to identify the respective highlighted parking spot or storey entrance and navigate through moving obstacles in the arena to reach the parking spot. Fig2 demonstrates the second level of the car parking game. The objects in the yellow color are obstacles and they move parallelly to the walls of the arena.

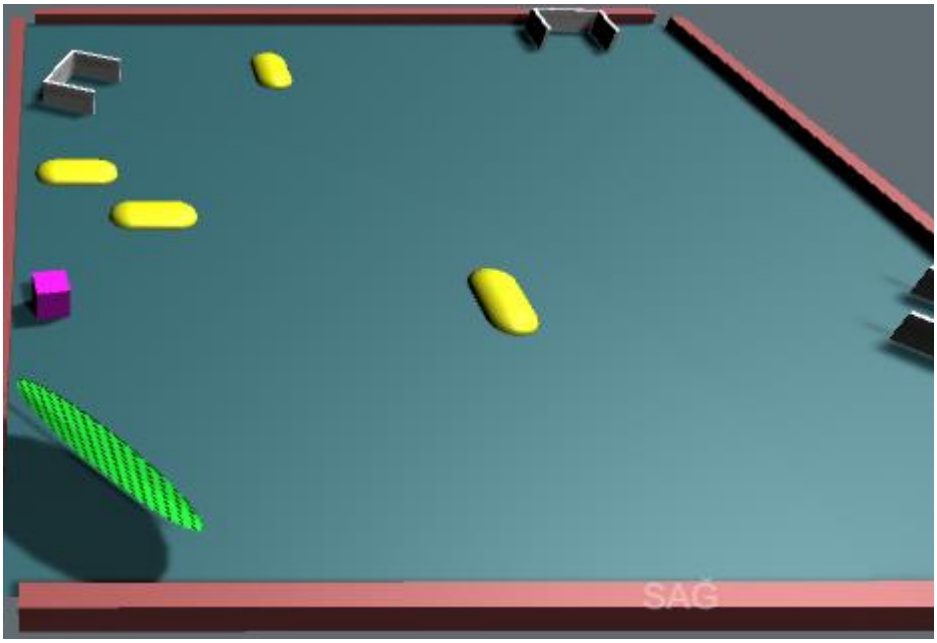


Fig2: Level 2 of the Car Parking Game

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.

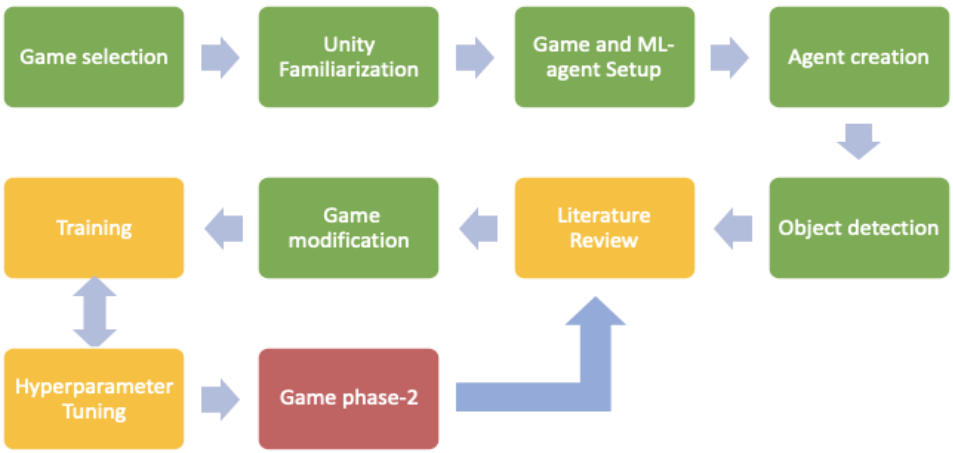


Fig3: Diagram explaining the different stages of the project

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.[7]

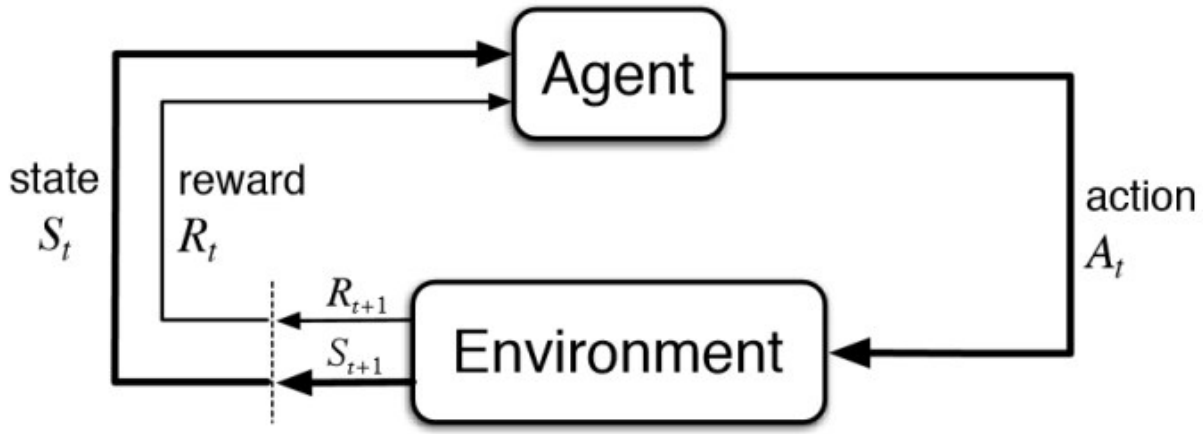


Fig4: Reinforcement Learning

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.[8]

PPO Objective function can be defined as follows:

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

- 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

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

1. 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.

2. 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.[9]

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, we migrated the game to the latest version of the Unity on Mac machine.

- 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

Mlagents 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.

7.1.3 Google Colabatory

- 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.[10]
- Google colab setup has following steps:
    a. Clone ml-agents repository onto google colab environment
    b. Install and activate ml-agents environments
    c. 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)
    d. Deploy headless server build and provide respective permissions to all the executables.
    e. Adding training configuration details to a file
    f. Enable tensorboard
    g. Start the training process using the command:

    !mlagents-learn ./config.yaml --run-id=$run_id --env=$env_name --no-graphics

h. 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. In our 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 that the agent performs. It can either turn left or right or perform no action which means that the car does not change direction.

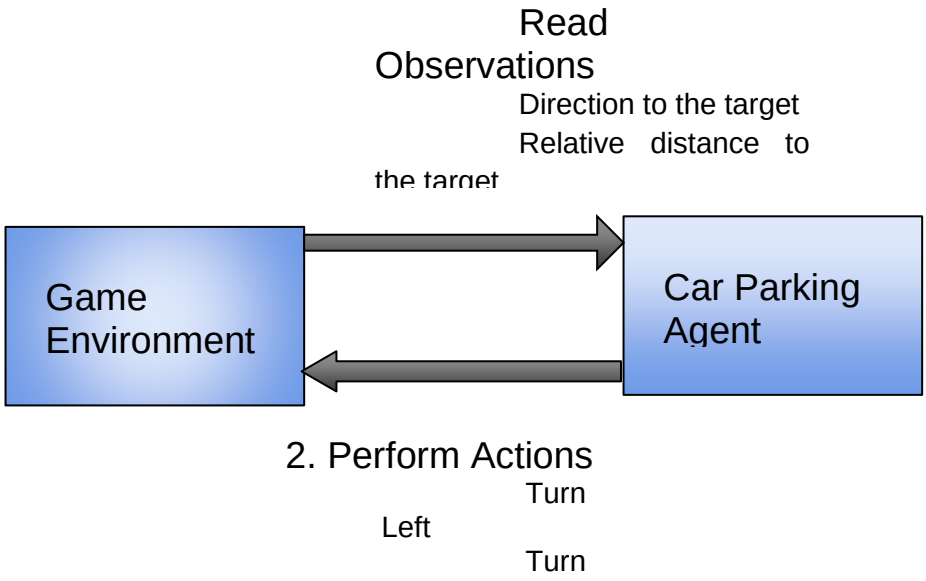


Fig5: High Level Design

8.2 Game Modifications

Some features have been to the game and made some modifications for incorporating machine learning capabilities in the game:

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:

a. Number of times agent parked

b. Number of times agent hit walls

c. Number of times agent hit obstacles

d. Cumulative reward in an episode: 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 us in debugging different scenarios and also identify if the agent is making correct inferences.

Fig6: Scoreboard

8.2.2 Navigation

The open source game from Unity had touch screen controls, to change the direction of the vehicle being parked. The touch screen controls have been changed 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. 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 mlagents 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 in order to reduce the training time.

8.2.4 Fixed but Multiple Goal Points

In our modified game, we have 3 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 and assigned for the agent to park. The higher the number of goal locations, the more is the training time for training the agent. We noticed that with 3 goal locations, it takes around 12 hours with imitation learning (GAIL) to train the model and hence used 3 goal locations for our project.

8.2.5 Collision Objects

In the open source game, only the obstacles in the arena are treated as collision objects. We modified the game to treat both the walls and also the obstacles as collision objects. Whenever the agents collides with either the wall or an obstacle, the episode ends and the agent is starting again at the start location to park the vehicle.

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. This script overrides 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 our 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. 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

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

9.	Car Parked [Episode Ends]	+5	+5

8.3.4 Agent observation space

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

a. *Direction to the target* : Relative and normalized Vector3 value

b. *Relative distance to the target*: Normalized Vector3 value

c. *Relative distance to obstacle 1*: Normalized Vector3 value

d. *Relative distance to obstacle 2*: Normalized Vector3 value

e. *Relative distance to obstacle 3*: Normalized Vector3 value

f. *Relative distance to wall 1*: Normalized Vector3 value

g. *Relative distance to wall 2*: Normalized Vector3 value

h. *Relative distance to wall 3*: Normalized Vector3 value

i. *Relative distance to wall 4*: Normalized Vector3 value

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. We set the DecisionPeriod to 3 which means that the Agent will request a decision every 5 Academy steps.

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 Observation

- "Space Size" represents the observation space size. In our case this value is 27
- "Stacked Vectors" represents the number of previous vector observations that will be stacked and used collectively for decision making. In our case, the value is 1.

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.

0	No change in the direction
1	Take a right turn
2	Take a left turn

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 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
```

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.

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

8.4.2.1 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.

8.4.2.2 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
```

8.4.3 Trained Neural Network Model

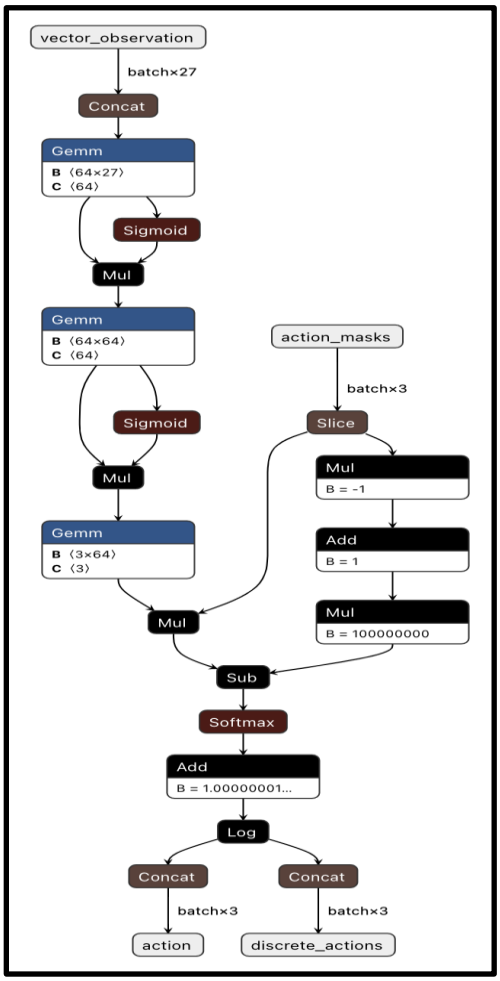


Fig7: Trained Neural Network Model

The model has following layers:

1. **Input Layer**: A vector observation consisting of 27 neurons (refer to section 6.4.4)
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.
3. **Output Layer**: The output layer consists of 3 neurons, one for each action (refer to section 6.4.6.2)

9 RESULTS AND EVALUATION

9.1 Cumulative Rewards

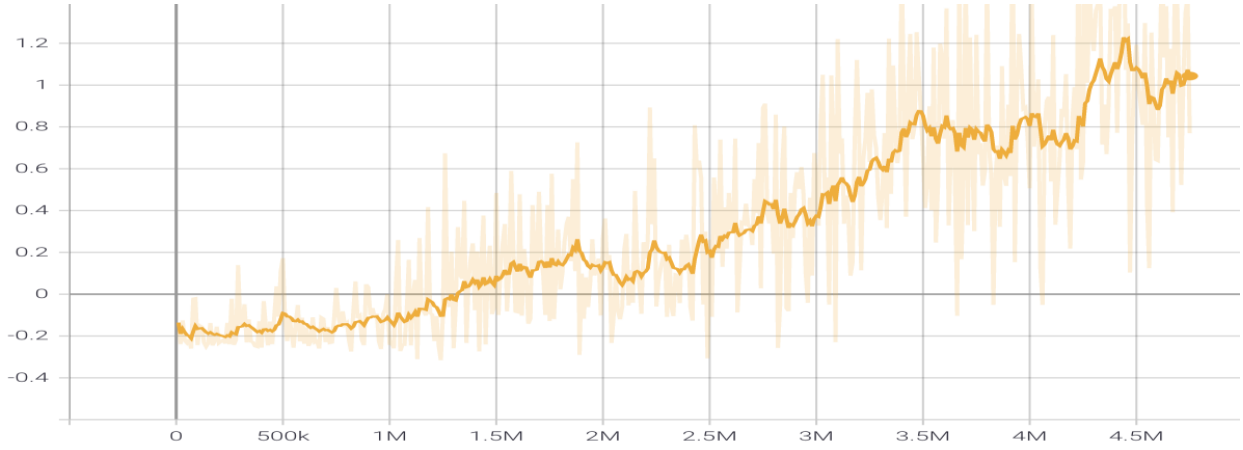


Fig8: PPO Cumulative Rewards

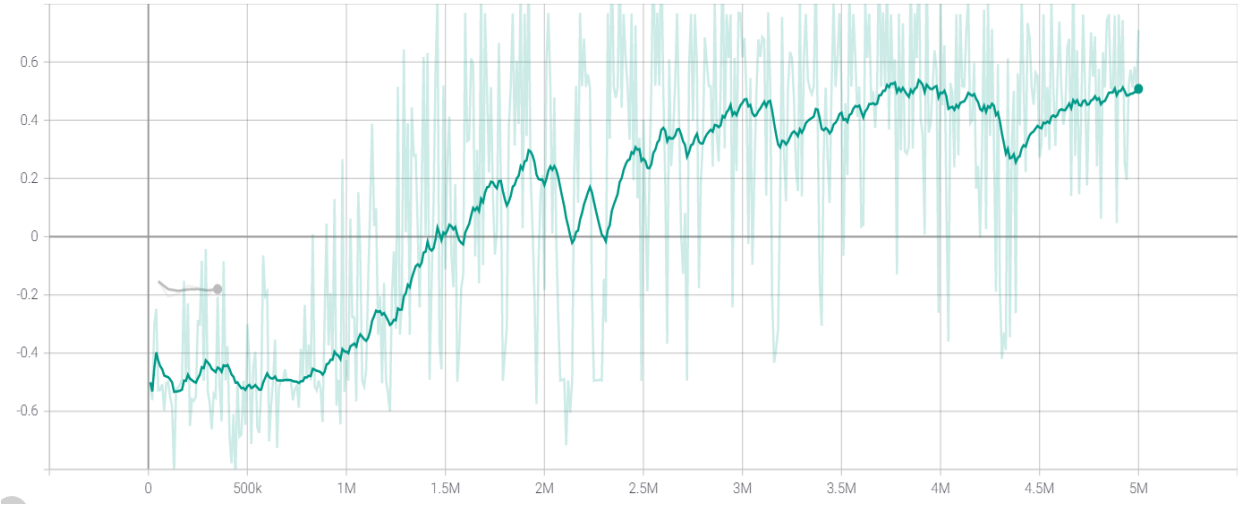


Fig9: PPO with GAIL Cumulative Rewards

The cumulative rewards keep 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.

9.2 Policy Loss

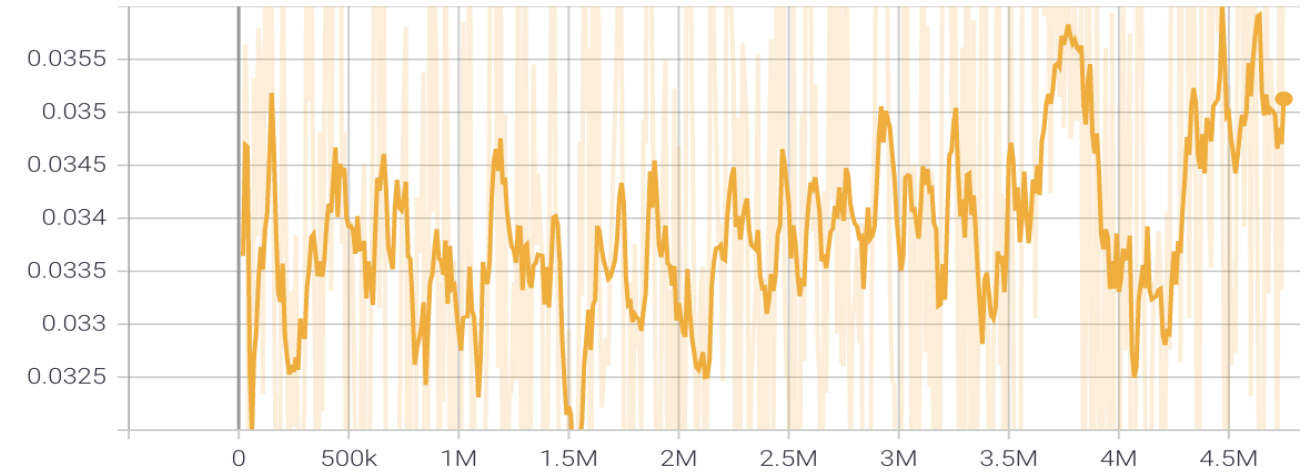
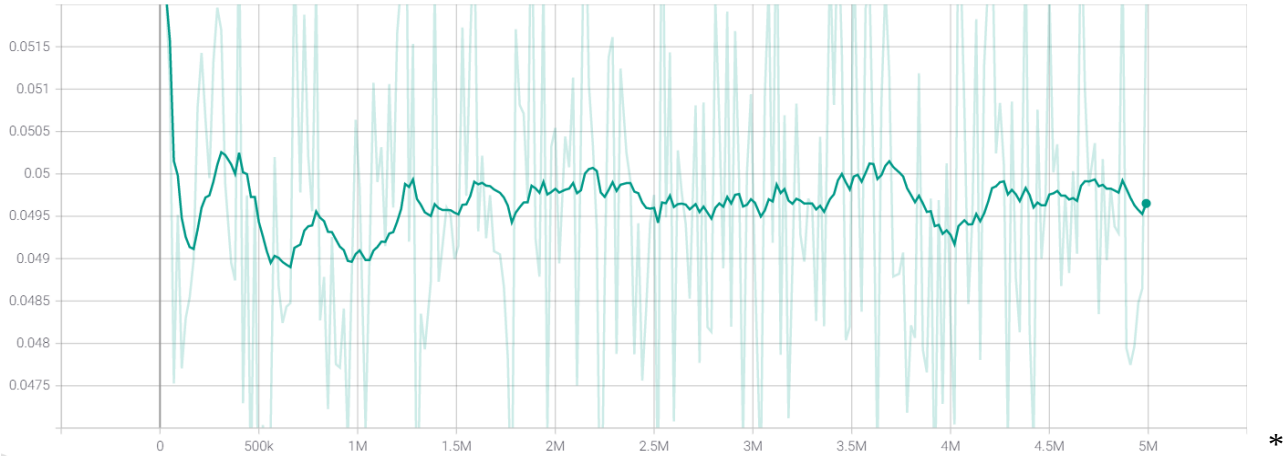


Fig10: PPO Policy Loss



*

Fig11: PPO with GAIL 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.

9.3 Entropy

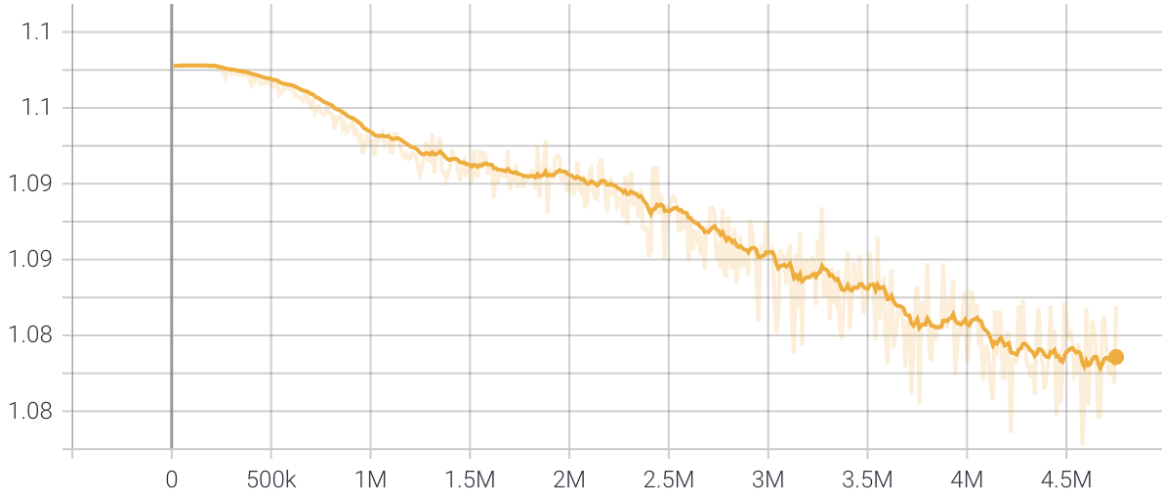


Fig12: PPO Entropy

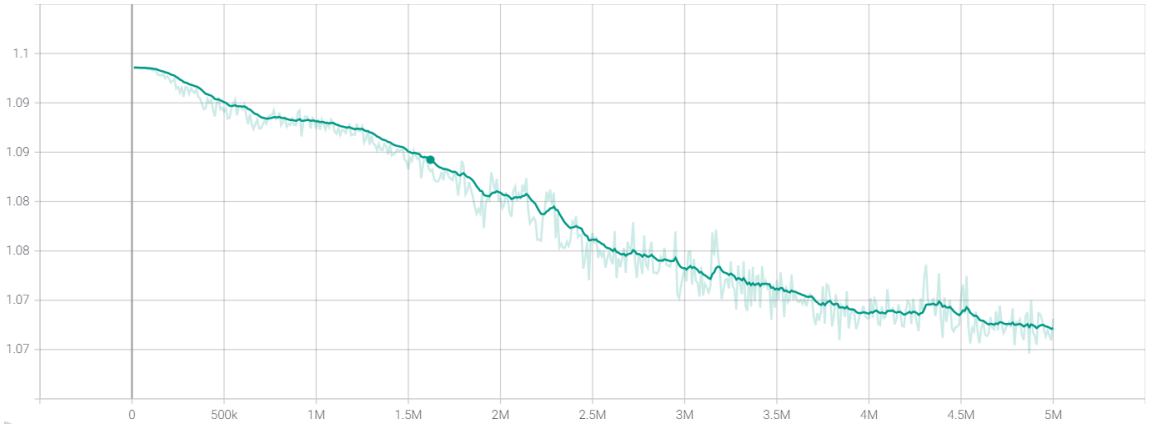


Fig13: PPO with GAIL Entropy

The entropy of the system is decreasing continuously over steps for both the algorithms. Initially the decisions of the agent are random, but as it learns optimal policy, the decisions become more informed.

9.4 PPO with GAIL Value Loss

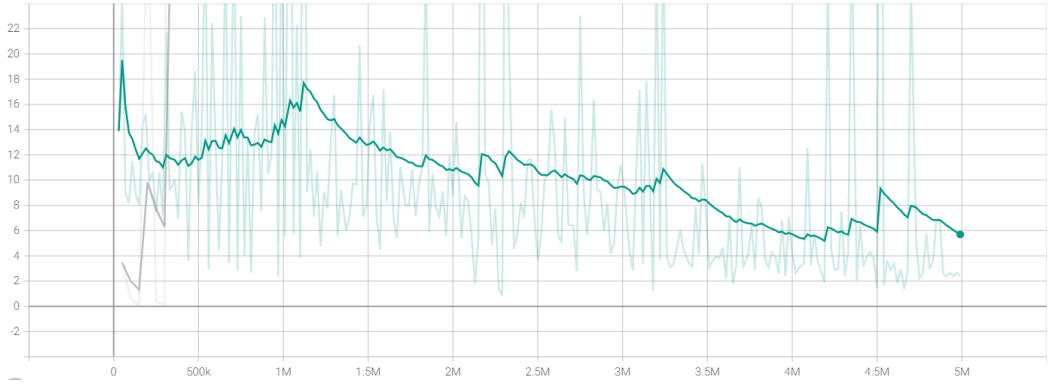


Fig14: PPO with GAIL Value Loss

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. [8]

9.5 GAIL Loss

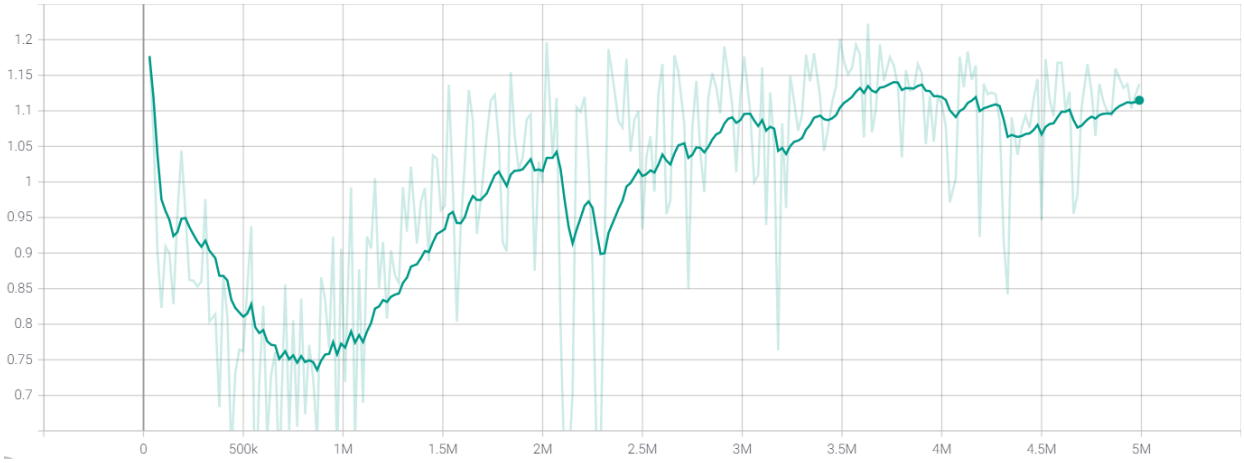


Fig14: GAIL Loss

The mean magnitude of the GAIL discriminator loss. Corresponds to how well the model imitates the demonstration data.[11]

9.6 Comparison

For testing the accuracy of the trained model, we added different parking spot locations from the ones we trained our model on and logged the results:

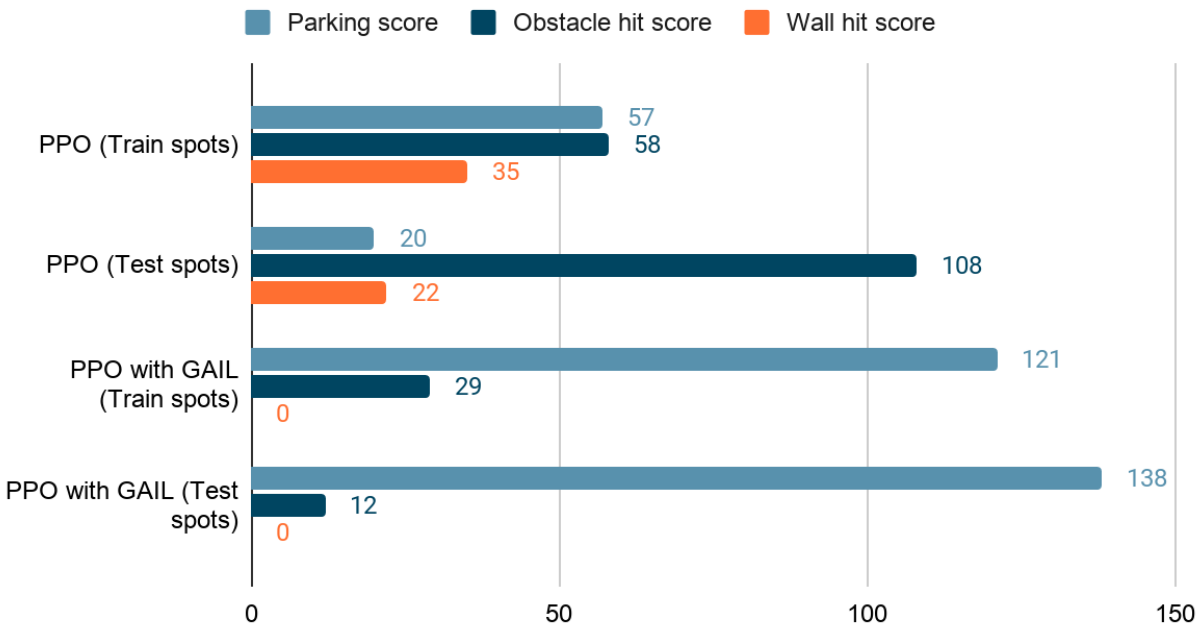


Fig15: Parking accuracy Comparison

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[12] 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.[13]

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 and ran for 2M steps:

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

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.[14]

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
```

10.3 Other Hyperparameters

We tried numerous other hyperparameters, but they didn't give us good reward gains:

SL.No.	Parameters	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	Decreasing reward function, performance didn't improve over time [ran for 5M steps]
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	Rewards didn't increase over time, no drop in entropy [ran for 1M steps]
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	Cumulative Rewards kept decreasing over time [ran for 1M steps]
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	Cumulative Rewards was constant over time [ran for 1M steps]
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	Decreasing reward function, performance didn't improve over time [ran for 5M steps]

11 CONCLUSION

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

12 FUTURE WORK

In future, we plan to train our agent compatible being able to park at any random location in the arena. This will require more training and sophisticated decision making skills. We have only used PPO and GAIL to train our model. However, we can also use SAC as another algorithm to train the agent and compare the performances of each of the models and determine the best algorithm for our project.

13 REFERENCES

[1] senevsemih. Unity–carparking. URL https://unitylist.com/p/134h/Unity-Car-Parking.

[2] Obstacle Avoidance and Navigation Utilizing Reinforcement Learning with Reward Shaping by Daniel Zhang, Colleen P. Bailey, 2020, https://arxiv.org/abs/2003.12863

[3] http://ceur-ws.org/Vol-2087/paper5.pdf

[4] D. Isele, R. Rahimi, A. Cosgun, K. Subramanian, and K. Fujimura, "Navigating occluded intersections with autonomous vehicles using deep reinforcement learning," 2017

[5] https://arxiv.org/pdf/1909.12153.pdf

[6] Aradi, S. et al. "Policy Gradient Based Reinforcement Learning Approach for Autonomous Highway Driving." *2018 IEEE Conference on Control Technology and Applications (CCTA)* (2018): 670-675.

[7] B. Thunyapoo, C. Ratchadakorntham, P. Siricharoen and W. Susutti, "Self-Parking Car Simulation using Reinforcement Learning Approach for Moderate Complexity Parking Scenario," 2020 17th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON), Phuket, Thailand, 2020, pp. 576-579, doi: 10.1109/ECTI-CON49241.2020.9158298.

[8] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, Oleg Klimov , Proximal Policy Optimization Algorithms, 2017, https://arxiv.org/abs/1707.06347

[9] Jonathan Ho, Stefano Ermon, "Generative Adversarial Imitation Learning", 2016, https://arxiv.org/abs/1606.03476

[10] https://dhyeythumar.medium.com/training-ml-agents-with-google-colab-cb166c3dca46

[11] https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Using-Tensorboard.md

[12] Deepak Pathak, Pulkit Agrawal, Alexei A. Efros, Trevor Darrell, "Curiosity-driven Exploration by Self-supervised Prediction", 2017, https://arxiv.org/abs/1705.05363

[13] https://towardsdatascience.com/curiosity-driven-learning-made-easy-part-i-d3e5a2263359

[14] Yuri Burda, Harrison Edwards, Amos Storkey, Oleg Klimov, "Exploration by Random Network Distillation", 2018, https://arxiv.org/abs/1810.12894