Project Report

INFO 6105
Data Science Engineering
Methods and Tools

Reinforcement Learning Using Unity ML-Agents

By
Xinyue Cui
Parthiv Shah
Pranav Mujumdar

Contents

Introduction

Brief overview

With groundbreaking innovations in robotics and artificial intelligence every day, there will be a huge amount of work that will need to be put to make them learn the tasks and make them smarter, of course one can program certain aspects to make them behave in certain way, but this may not be a feasible way when dealing with complex situations. Sometimes, just by defining what is good and what is bad we can effectively make the computer learn what actions to take. With examples like DeepMind's AlphaGo (DeepMind, 2015), we can certainly say that there is incredible research going on in this branch of artificial intelligence like never before.

Starting with IBM's Deep Blue (IBM, n.d.), a computer that defeated Gary Kasparov in 1996 and 1997, artificial intelligence is rapidly trying to catch up to human intelligence. With advances in engineering and manufacturing the computers have gotten faster and faster every iteration, and this made the terms like Machine Learning, Artificial intelligence, Deep learning very common nowadays. These are now the things that a fairly good programmer and someone who has knowledge of statistics and probabilities, can toy around on his own personal computer.

About Unity

Unity is a development tool that is widely used by game developers, designers, artists, to make 2D/3D games and AR, VR visualizations. (Unity, n.d.)
There is an online aspect of gaming where there are multiple players playing the game either together in a team or against each other, called "Network Gaming". However, in order to achieve that we have to make sure there are multiple players connected to the game, for example the popular game PUBG(Player Unknown Battle Ground), where 100 players have to be connected to play the game, but sometimes there not enough players to start the game, and we don't want human players to wait around for hours to get started with the game, for avoiding which the developers found a clever way, adding AI that can play the game almost as good as humans, so suppose there are 70 human players ready to play the game,

the game will add remaining 30 AIs to get the play started. These AI players are fondly referred to as “Bots” in the gaming circle. This encouraged fast development in the field of AI developing tools.

Why use reinforcement learning?

There are certain human activities that are so complex that they cannot be mimicked by writing the usual for or while loops or if-else conditions, even if we successfully break everything down to the most minute possibilities and chances and make a program that behaves like a human it would be only good at the specific thing that it was made for, not to mention the gigantic amount of work it will take to program something so complex. For achieving such complex things, we can tell the computer what actions it can take, give it appropriate rewards for taking good decisions and actions, and punishing it when it does something wrong. Just like training a dog, we reward the dog when it sits on our command, reinforcing in his brain that when the word “sit” is said, it has to sit, so that it can get its favorite treat. Similar thing happens with the adolescent child, it does random things and either gets rewarded or gets punished by his parent, reinforcing good and bad things in his brain.

Artificial Neural Networks

Human brain is filled with neurons connected to each other, that send electrical signals to each other to make a “decision” or take any action. Our decisions and actions are naturally the sum and average of all our past experiences. This complex arrangement of neurons is being used by Deep learning algorithms by mathematically translating the neurons into nodes interconnected to each other.

Background

Problem Statement

Self-driven cars are something we have been hearing for the past few years and a lot of research is going on to make an autonomous car. Tesla has successfully implemented the self-driven car that makes real time decisions based on image recognition by capturing images using cameras mounted on all sides of the car.

This project aims to create and simulate an AI driven car, that will be able to find the parking spot in a parking lot. In order to understand the programming aspects of using the unity engine and simulating the scenario, we will be first trying to make an AI that can play the infamous game called flappy bird, as there is abundance of reference material for it, and AI will have to make a simple decision as to when to make the bird jump, to avoid the obstacle.

Objectives

- Understand various algorithms in reinforcement learning, like Q-learning, Proximal Policy Optimization, Soft-Actor-Critic.
- As a precursor to the main project build an AI that can play the game "Flappy Bird"
- Finally, after understanding implementation process in Unity ML agents, try to build an AI that can find the empty parking spot in the parking lot

Methodology

Most of the reinforcement learning algorithms follow a similar approach, as explained below

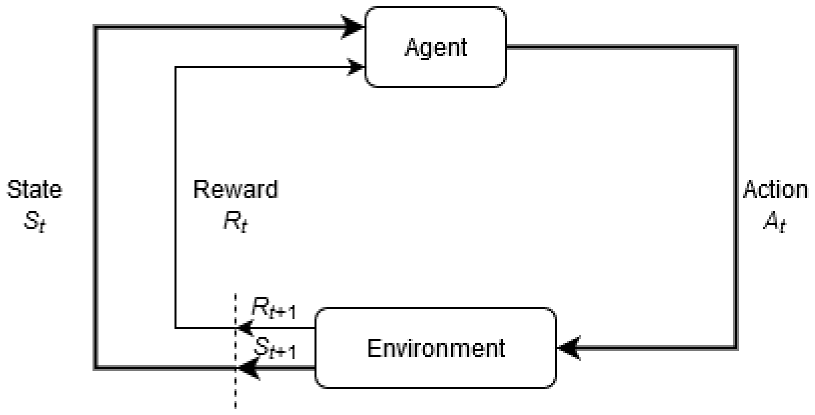


Figure 1 Agent-Environment Interaction cycle

(Bartow, 2018)

We have an agent, which takes certain actions, in an environment, and its actions are rewarded with either a punishment or a positive reward, and then it changes to the next state, where it again has to take an action. This loop continues and the agent finds out the most favorable actions for him (i.e. Actions that got him the most positive reward), and avoids actions that caused him to get a punishment. The terminal state of this loop is at the end of the episode, which can either be defined when the agent takes a certain number of steps and still doesn’t reach its expected goal or if it does reach its goal.

In our project, of finding a parking spot, we have a goal defined, as soon as the Agent(car) reaches the parking spot we will reward it positively, and if it hits another car that is already parked, we will punish it with a negative reward. The episode ends when the agent finds the parking spot or after 5000 steps, as this seems to be an ideal time to be able to find if there are any parking spots available.

For better understanding of the further technical aspects we need to clarify a few definitions

Definitions

1. *Agent:* The actor or the actual AI that makes the decisions, represents the RL algorithm
2. *Environment:* The object on which the agent takes the actions, (e.g. In flappy bird the game is the environment, and in parking AI the parking lot area is the environment)
3. *Action(A):* List of all possible actions that an AI can take
4. *State(S):* Current status or state of the situation returned by the environment
5. *Reward(R):* The reward that the environment assigns for each action that the agent takes
6. *Model Free Algorithms:* most of the reinforcement learning algorithms are model free, which means that the agent takes action based on trial and error, and updates its decisions depending upon the reward it gets
7. *Neural Networks:* A neural network computing systems vaguely inspired by the biological neural networks that constitute animal brains. Such systems "learn" to perform tasks by considering examples, generally without being programmed with task-specific rules (Collaborative, 2020)
8. *Hyperparameters:* These are the parameters that allow us to fine tune our learning policy, and optimize the training of the agent (Unity, n.d.)
9. *Policy:* A *policy* is a rule used by an agent to decide what actions to take. It can be deterministic; or it may be stochastic.

   Because the policy is essentially the agent's brain, it's not uncommon to substitute the word "policy" for "agent", e.g. saying "The policy is trying to maximize reward." (OpenAi, 2018)

Algorithms

We have mainly used two general purpose RL algorithms that are favorable to be used in the scope of our project and are supported by unity’s ML-agents toolkit.

Proximal Policy Optimization

An algorithm released by OpenAI, that performs comparably or better than state-of-the-art approaches while being much simpler to implement and tune. PPO has become the default reinforcement learning algorithm at OpenAI because of its ease of use and good performance.

With supervised learning, we can easily implement the cost function, run gradient descent on it, and be very confident that we’ll get excellent results with relatively little hyperparameter tuning. The route to success in reinforcement learning isn’t as obvious — the algorithms have many moving parts that are hard to debug, and they require substantial effort in tuning in order to get good results. 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. (Schulman, Klimov, Wolski, Dhariwal, & Radford, 2017)

Soft Actor Critic

An algorithm released by Berkeley Artificial Intelligence Research, Soft actor-critic (SAC), is an off-policy model-free deep RL algorithm. In particular, we show that it is sample efficient enough to solve real-world robot tasks in only a handful of hours, robust to hyperparameters and works on a variety of simulated environments with a single set of hyperparameters. (Haarnoja, et al., 2018)

Generative Adversarial Imitation Learning

GAIL itself is not an algorithm however it can be paired with another algorithm, in order to achieve faster training. The idea behind this is that if we collect data from few episodes where a human expert has taken actions and provide these episodes to the algorithm while training. This, however, has drawbacks, because we want the

AI to perform better than humans, do tasks more efficiently. Hence it is preferable to not use GAIL, if the objective is to build an AI better than humans.

Implementation of AI for Flappy Bird

Overview of the environment

Before we implement the agent that can play the game, we need to understand how the game is to be played. As depicted in the picture below, the user taps the screen to make the bird fly, which in actuality is similar to man jumping a hurdle. If the bird falls to the ground the game is over and if it crashes onto the oncoming obstacles, the game is over. The trick of the game is to make sure the bird flies from between the obstacles, not to fly too high or too low.

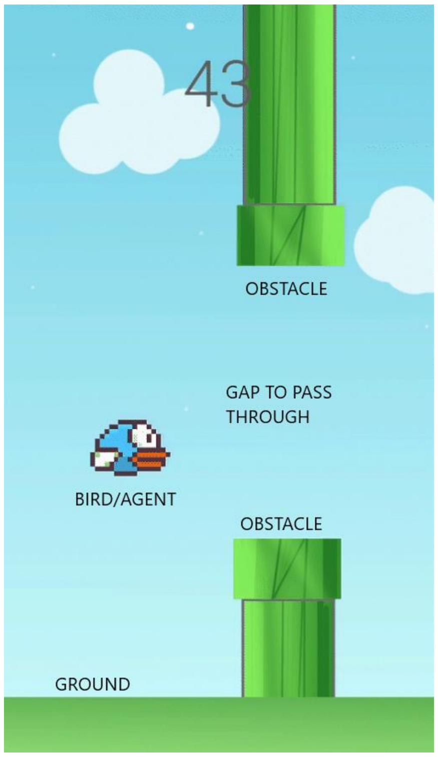


Figure 2 Flappy Bird Environment

Specification and building the environment using unity:

- Firstly, we created the required assets like the pipes, the background, using GIMP, an open-source image creator. Also find the image of a suitable looking bird online, as creating bird is irrelevant to our project and time-consuming process and requires artistic mastery.
- To make these assets into a game we drag and drop them onto the scene view, and then add logic to them using C# scripts.

- Bird Script (Assets/Scripts/BirdAgent.cs) Our bird is affected by gravity, and has to be detectable in the environment, hence we add a 2DRigidBody and a circle collider onto it. The Push() method inside the script applies upward force onto the bird.
- Pipes Script (Assets/Scripts/PipeSet.cs and Pipes.cs and BottomPipe.cs) The pipes have to be moving from right to left continuously with varying distance between them and at varying height. With trial and error, we managed to get the good amount of variation between the pipes. The pipes also need to be assigned a box collider

Configuring the agent

Before writing the agent script we need to add some elements in the editor for the agent to perform properly.

1. In order to get the Agent to make decisions whether to jump or not, we need to add sensors that will detect the oncoming obstacle, which can be done by adding something called as “Ray Perception Sensors” on the agent

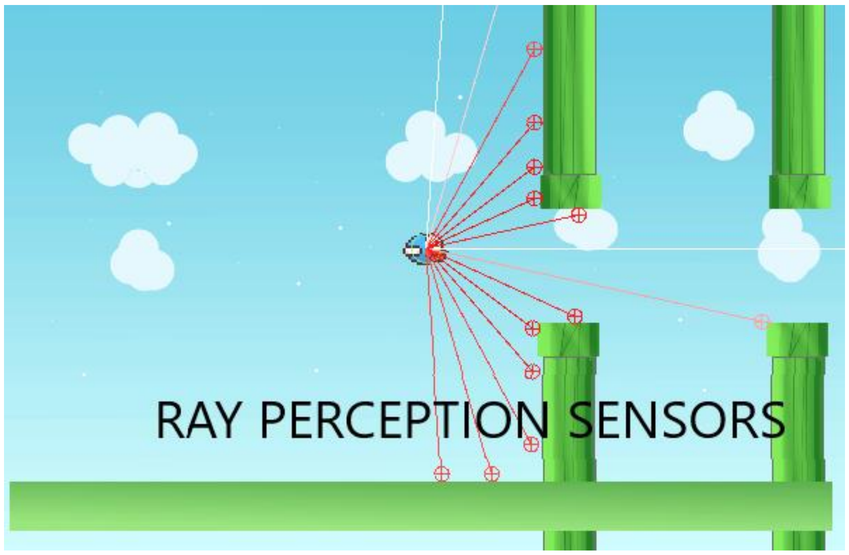


Figure 3 Depicting the Ray Perception Sensors 2D

   The agent will keep collecting information from these sensors and take appropriate actions.

2. Behavior Parameters
   a. Vector Actions: these are the actions that the agent can take, we need to make sure we add proper nature of the actions whether continuous

    or discrete, continuous actions are in a range and gradually increasing whereas discrete actions are in discrete values, e.g. on or off is a discrete action

    b. Vector Action Branches: Here we have only two actions, either jump or do not jump, both in one branch, so we give 2 actions in the branch 0.
    c. Use child sensors: Set it to true, as the sensors that are attached to the agent are to its child object and we need to make sure the agent considers the inputs from the sensors.

3. Decision Requester
   In order to get the decisions from the agent, we need to add the decision requester script, and set the decision period to 5, as we don’t want the AI to take decisions every moment, we can let the AI decide some actions and then follow its decided actions for a while

Scripts:

1. Agent Script
   Path to script: Flappy Bird\FlappyBird\Assets\Scripts\BirdAgent.cs
    1. We have to make sure the algorithm know that this is an Agent, for which we need to derive it from the Agent class
    2. There are some necessary methods that need to be overridden from the Agent base class so that the configuration is logical, as below
        - OnEpisodeBegin()
          This method makes sure every episode maintains the standard set of rules in the environment
        - OnActionReceived()
          When the agent decides to take an action based on the vector actions we defined in the editor, we have to configure what actual changes happen to the agent and the environment
        - Push()
          This method applies the actual forces that makes the bird jump
        - Died()
          This Method allow us to punish the agent when it hits the obstacles or the ground, we give punishment of 1, i.e. AddReward(-1f);

Please refer to the documentation inside the actual script for the other non-essential methods.

2. Collision Detection Script

   Path: Flappy Bird\FlappyBird\Assets\Scripts\CollisionDetect.cs

   - This script handles the logic of the agent colliding with the obstacles which are tagged as “DeadZone”, upon collision with which they Died() method inside the Agent is called and the episode resets.

Additional scripts like the ScoreUpdater were added for tracking how much time the AI stays alive were added so that we can get a quick understanding of how well it is performing during the training and we are also able to view the rewards, using TensorBoard charts and tracking how much reward the AI was able to collect throughout the episode. Here the length of the episode is a supposed to be increasing as the AI tries to survive as long as possible

Once everything is set up, we go ahead and start training the AI, firstly using the Proximal Policy Optimization

Training

For training the AI, we need to make sure all the installations are properly set up, please refer to the installations chapter in Appendix and User Guide section of this report.

For the AI to train quickly, we create multiple instances of the environment and train the AI simultaneously on multiple instances and all the experiences from the different environments collect and report to the same neural network manipulating the weights of the nodes.

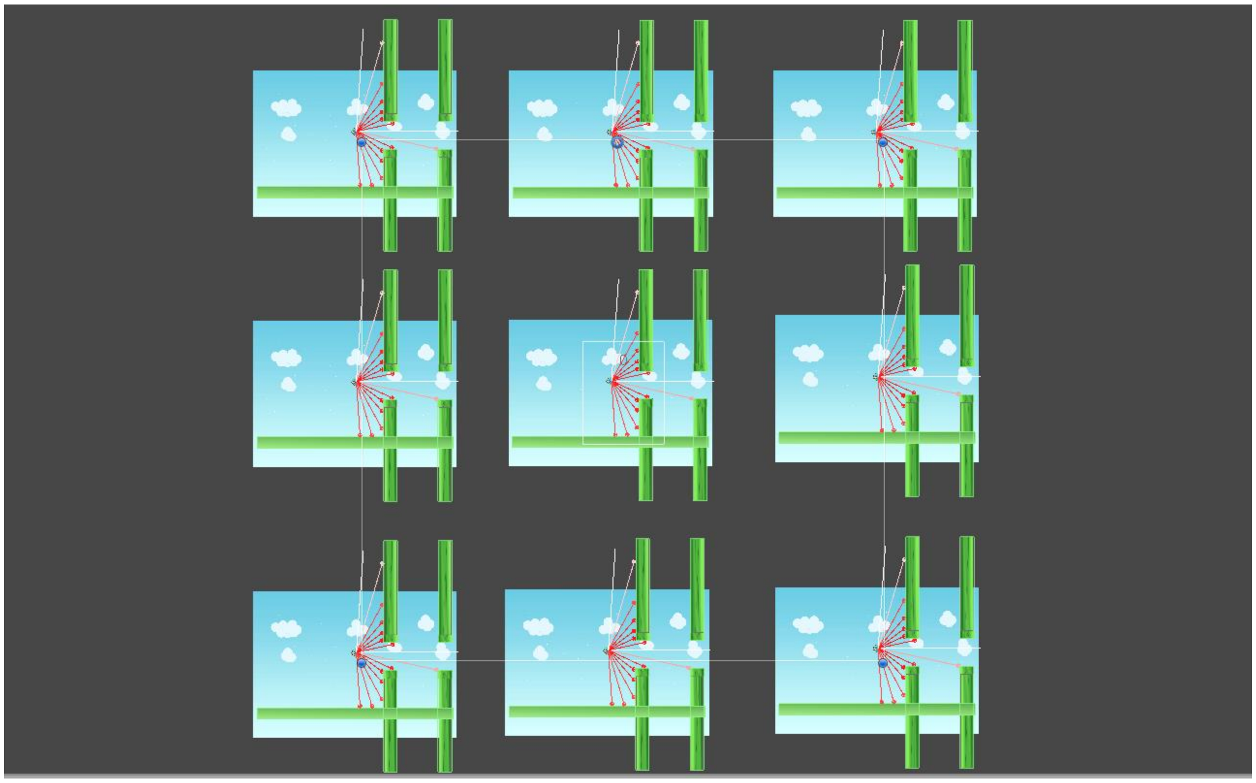


Figure 4 Multiple instances of the Environment

Finalized Hyperparameters

Please refer the Appendix for detailed description of the Hyperparameters

For PPO algorithm Following hyperparameters were used:

```
default:
    trainer: ppo
    batch_size: 1024
    beta: 5.0e-3
    buffer_size: 10240
    epsilon: 0.2
    hidden_units: 128
    lambd: 0.95
    learning_rate: 3.0e-4
    learning_rate_schedule: linear
    max_steps: 5.0e5
    memory_size: 128
    normalize: false
    num_epoch: 3
    num_layers: 2
    time_horizon: 64
    sequence_length: 64
    summary_freq: 10000
    use_recurrent: false
    vis_encode_type: simple
    reward_signals:
        extrinsic:
            strength: 1.0
            gamma: 0.99
FlappyBird:
    max_steps: 1.0e7
    num_epoch: 3
    batch_size: 256
    buffer_size: 4096
    beta: 1.0e-2
    hidden_units: 512
    summary_freq: 60000
    time_horizon: 64
    num_layers: 3
```

Training Time: 8:29:11HR, as the max_steps are 1e07

Results

Following are the graphs generated using TensorBoard

Training 1: denoted by the RED line

Training 2: denoted by the GRAY line

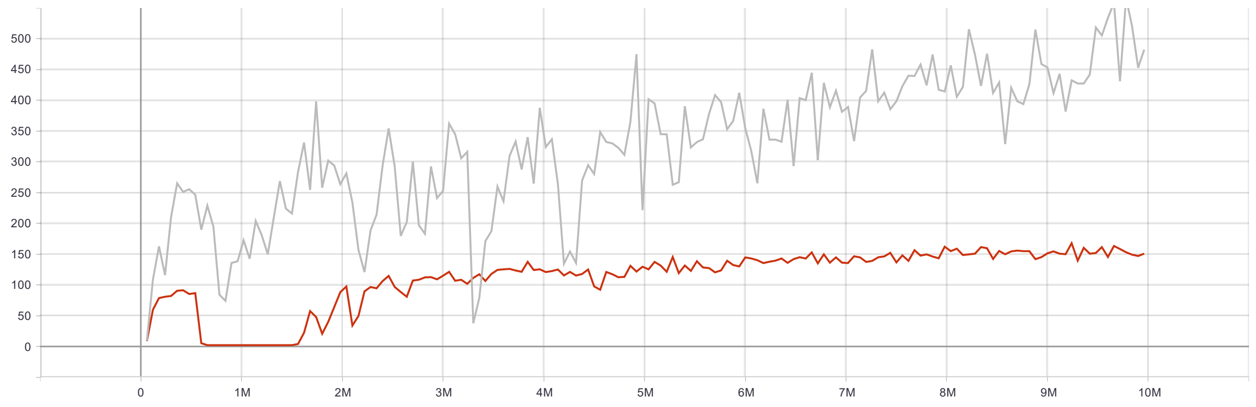


Figure 5 Flappy Bird Environment Cumulative Reward

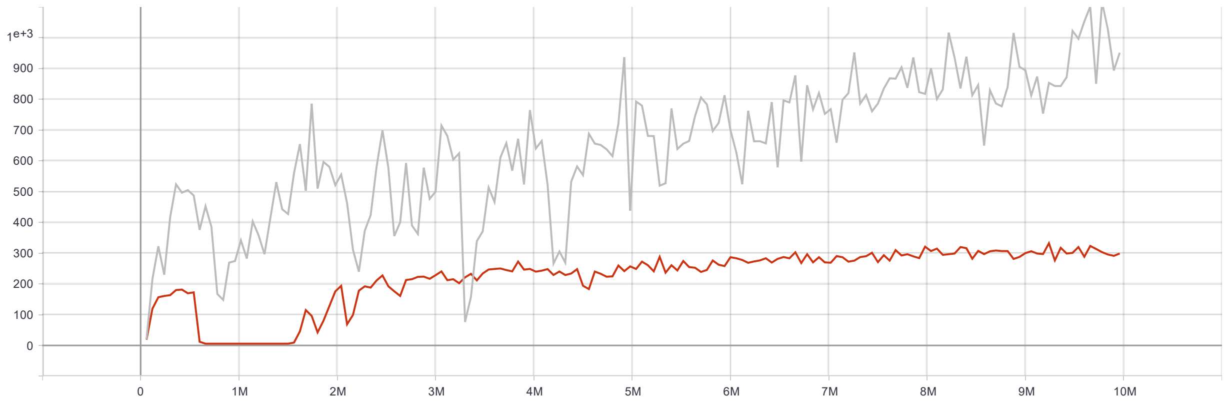


Figure 6 Flappy Bird Environment Episode Length

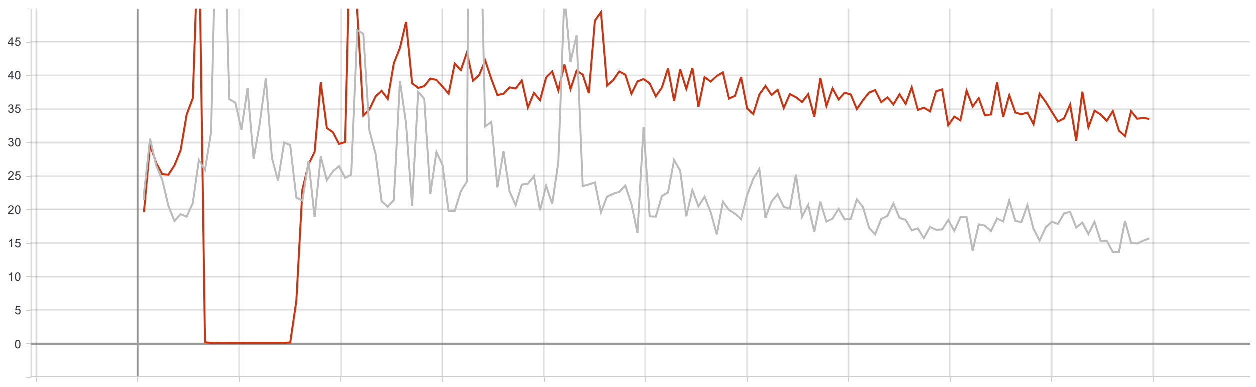


Figure 7 Flappy Bird Losses Value Loss

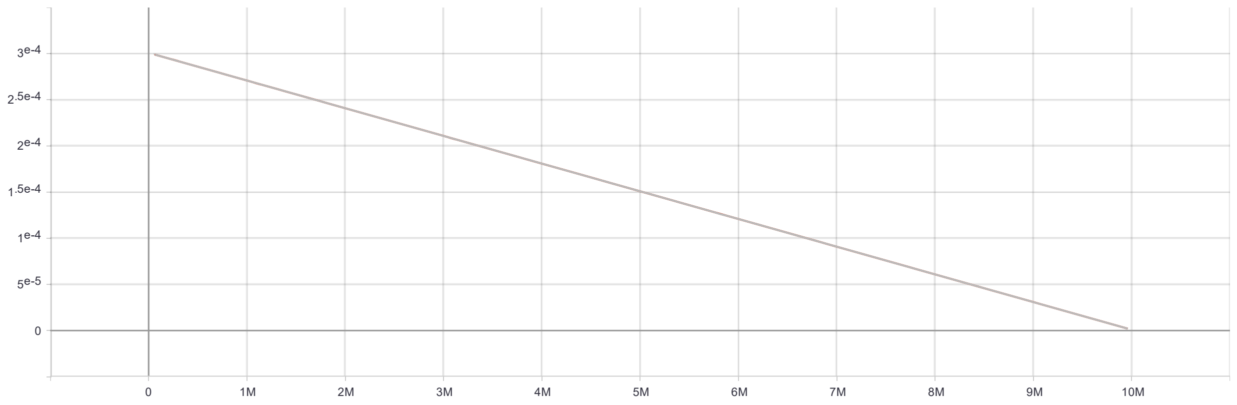


Figure 8 Flappy Bird Policy Learning Rate

Analysis of the Results:

1. Cumulative Reward:
   - After 60k Steps
     i. Training 1: 8.824
     ii. Training 2: 9.618
   - After 9.96M Steps
     i. Training 1: 150.9
     ii. Training 2: 482.3
2. Episode Length:
   - After 60k Steps
     i. Training 1: 18.9
     ii. Training 2: 20.46
   - After 9.96M Steps
     i. Training 1: 298.8
     ii. Training 2: 951.5
   - The Episode length must increase as we train, as we want the agent bird to survive as long as possible
3. Value Loss: Correlates to how well the model is able to predict the value of each state. This should increase while the agent is learning, and then decrease once the reward stabilizes.
4. Learning rate
   - How large a step the training algorithm takes as it searches for the optimal policy. Should decrease over time.
   - Coincides for both the trainings

Implementation of an Autonomous Parking AI

After receiving satisfactory outputs from the Flappy Bird implementation, we move on to the main objective of our project, i.e. creating an AI that will be able to find a parking spot, in a parking lot.

Overview of the environment

The environment consists of a parking lot with certain cars, already parked, and the agent car that is looking for a parking spot. To ensure there is not one specific parking spot that is available or vacant, we have randomized the spot and other parked cars, as well as the agent car that requires to be spawned in different positions of the parking lot and in different orientations as well.

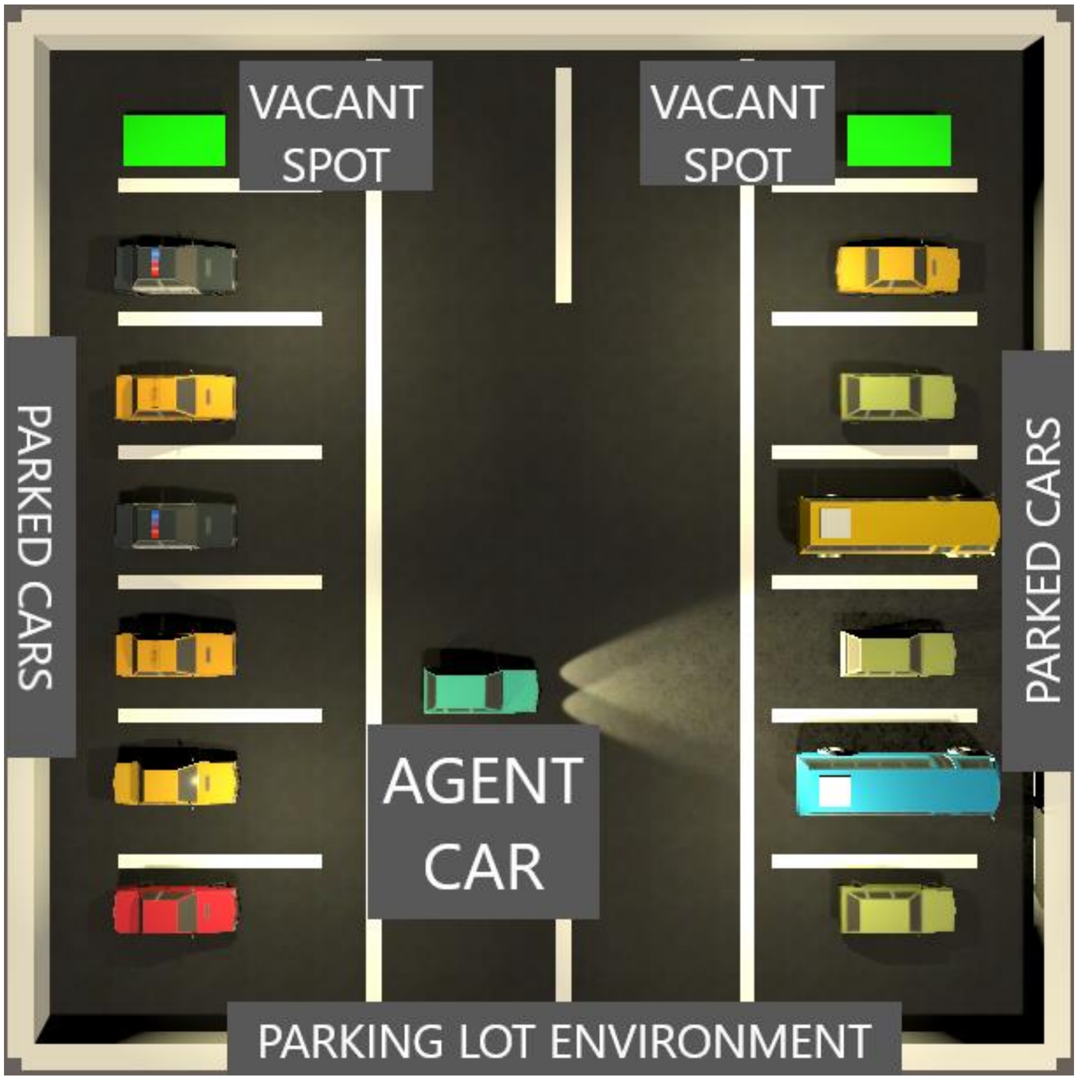


Figure 9 Car Parking AI Environment

Note: All the cars and the vacant spots are generated randomly with a chance of 85% that the spot will be occupied. So, there might be some episodes where there are no vacant spots in the lot, which encourages the agent to look through the whole parking lot for the spot.

Specification and building the environment

1. Assets
   - Cars: These cars models were downloaded from as a free package from the unity asset store, every car has a body and four wheels which allows us to apply torque to the wheels

Figure 10 Low Poly Car Models from Asset Store

2. Building the environment
   - Parking lot

     We created the parking lot which is a square shaped plane with walls on the sides, the area has multiple spawn positions, for randomly spawning the parked cars and the agent car, using script, ParkingArea.cs

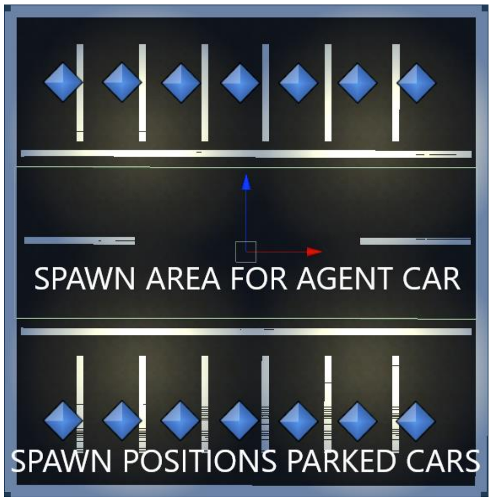


Figure 11 Layout of the environment

- Script Description:
  The script contains following methods
  - randomSpawnCarsAndParking()
    This method helps randomly spawning either a car or a parking on the positions marked in blue diamonds in the above image
  - clearParking()
    This method allows the area to be cleared for next random spawning once the episode ends
  - resetArea()
    This method is created to call the above two functions one after the other from the agent script
- Car setting Script
  This script contains the variable for margin multiplication where the agent car would spawn, the larger the margin the more difficult it is to train, as the agent car would spawn randomly at different locations
- Collision Detection Scripts
  - Obstacle Detection (obstacleDetect.cs)
    This script will detect when the agent car collides with any of the spawned parked cars, and call the hitACar() method inside the agent script
  - Wall Detection (wallDetect.cs)
    This script will detect when the agent car collides with the side walls of the parking lot and call the hitAWall() method inside the agent script
  - Parking Detection (parkingDetect.cs)
    This script will detect when the agent car properly finds the empty parking spot and drives to it. It will call the parked() method inside the agent script

Configuring the agent

This is our agent car, the car that needs to find its parking spot. For making sure the car behaves the way it should, we apply following build

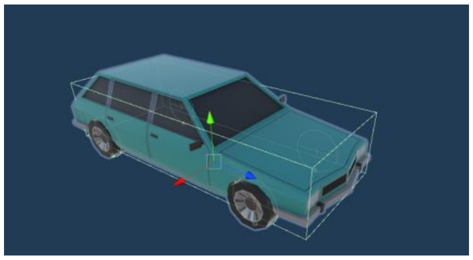


Figure 12 Agent Car

We add wheel colliders onto the front wheels of the car to enable real like driving ability to the car

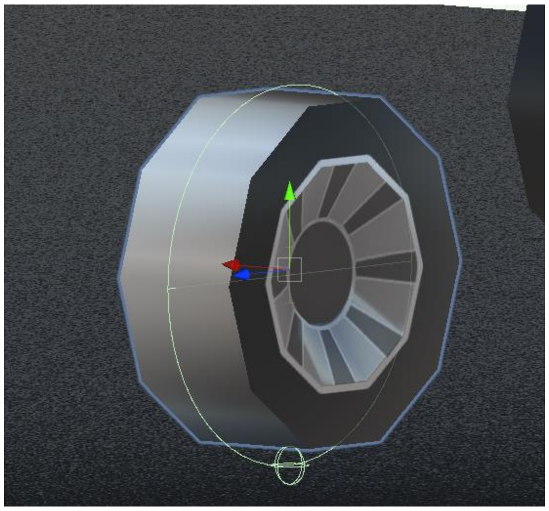


Figure 13 Shows the Wheel Collider (a thin green line) added to the wheel

- Steer Wheels: The two wheels in front will be able to have the steering ability
- Driving Wheels: Considering the car is front wheel drive car we add the driving ability to the front wheels as well Adding a box collider on the car, with a rigid body that weighs around 1500kg, just to make the car behave life-like.
- Adding Ray Perception Sensors
  Just like we did for our flappy bird agent, we add 3-dimensional ray perception sensors onto the car, in the front and in the back, which would be able to sense objects in 10m range.

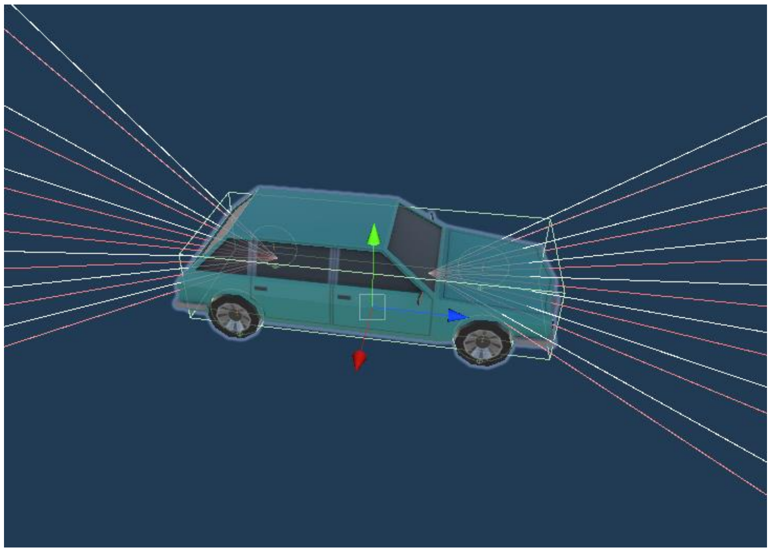


Figure 14 Shows the 3D Ray Perception Sensors added to the agent car

Agent Behavior Parameters

- Vector Action Space Type: Since this is a case where the agent should be able to partially steer the car, we will be keeping the vector actions continuous as opposed to discrete actions in the flappy bird agent.
- Vector action Space Size: 2
  The car can either go forward or reverse which constitutes as one action space. Also, the car can turn left or right, which constitutes as one action space. Hence, we use two as the vector action space size.
- Decision requester: We keep the decision period to 5 as we don't want the AI to make a decision before every action. And we also allow the AI to take actions between decisions.

Agent Script:

Now we can go ahead and write the agent script

Following are the important functions we defined in the script

- GetRandomSpawnPos(): Get a random Spawn position for the agent car in the center area of the parking lot depending upon the spawn area margin multiplier
- hitACar(): Punishing the agent for hitting another parked car, we add a reward of -0.1f
- hitAWall(): Punishing the agent for hitting the wall of the parking lot, we add a reward of -0.1f
- parked(): Positive reward for finding the empty parking spot

- OnActionReceived(): When we receive an action from the agent, we call the moveAgent method, also we add penalty to make sure the agent improves and tries to perform in least number of actions
- moveAgent(): This function actually drives the car, depending upon the action received, this method will steer and accelerate the car.
- OnEpisodeBegin(): Every episode needs to maintain standard values for making sure the AI learns properly, like the car should be stationary at the beginning, so we remove all the forces from the previous episodes. Also, we need to reset the area and randomly spawn the parked cars, and the empty parking spots, which we can do using this function.

Note: Please refer to the script, CarAgent.cs, (path: \CarParking\Assets\Scripts\CarAgent.cs)

Training

Now that we have configured the agent and the environment, we can move onto training phase, just like the flappy bird example we replicate the environment multiple times to train the AI faster. We create 12 instances of the environment.

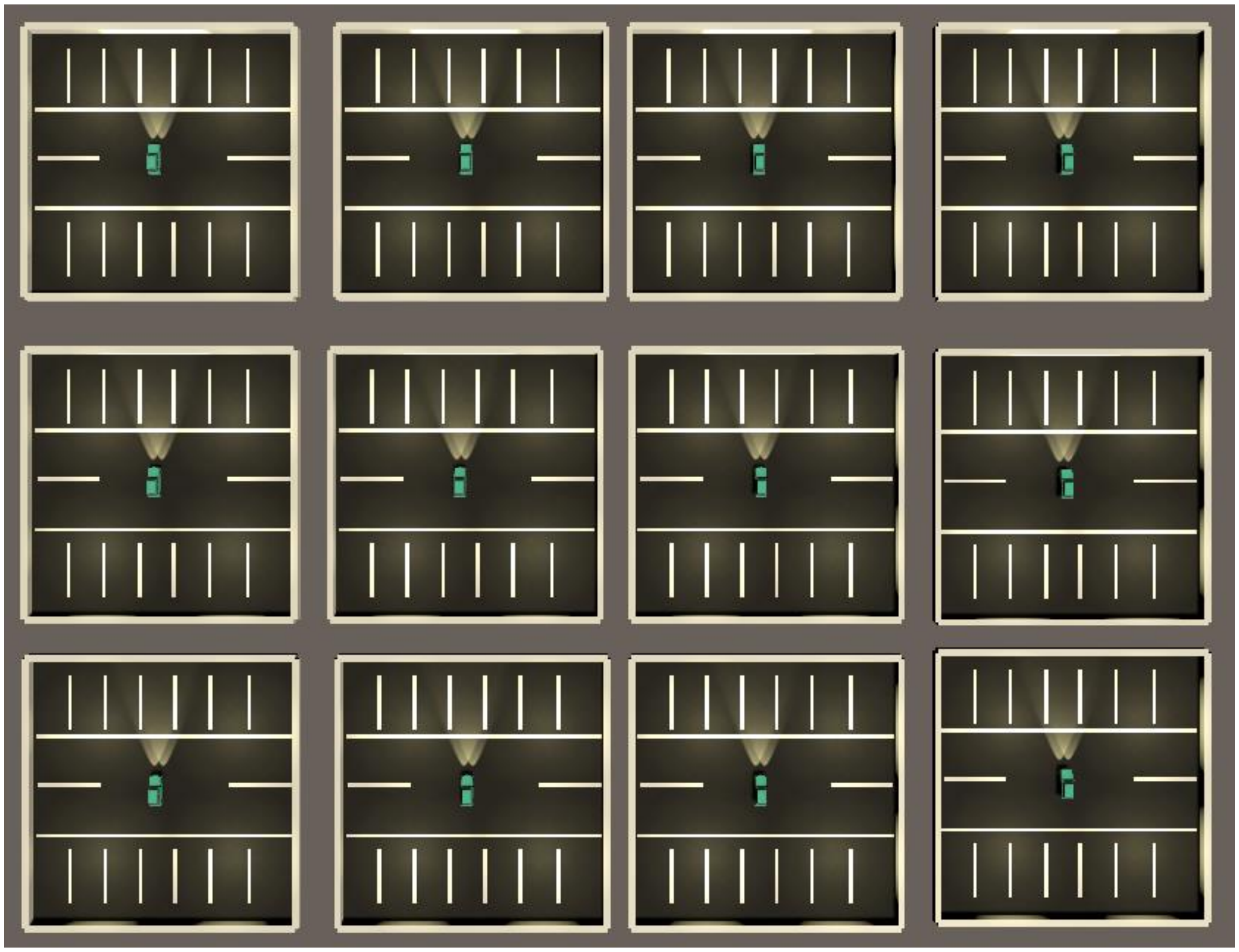


Figure 15 Shows the 12 instances of the Parking environment

And we train the AI first using PPO

Finalizing the Hyperparameters

After trying various values for the hyperparameters we ended up finalizing the following

```
 default:
    trainer: ppo
    batch_size: 1024
    beta: 5.0e-3
    buffer_size: 10240
    epsilon: 0.2
    hidden_units: 128
    lambd: 0.95
    learning_rate: 3.0e-4
    learning_rate_schedule: linear
    max_steps: 5.0e5
```

```
        memory_size: 128
        normalize: false
        num_epoch: 3
        num_layers: 2
        time_horizon: 64
        sequence_length: 64
        summary_freq: 10000
        use_recurrent: false
        vis_encode_type: simple
        reward_signals:
            extrinsic:
                strength: 1.0
                gamma: 0.99
     summary_freq: 30000
        time_horizon: 512
        batch_size: 512
        buffer_size: 2048
        hidden_units: 256
        num_layers: 3
        beta: 1.0e-2
        max_steps: 1.0e7
        num_epoch: 3
        reward_signals:
            extrinsic:
                strength: 1.0
                gamma: 0.99
            curiosity:
                strength: 0.02
                gamma: 0.99
                encoding_size: 256
```

We added extrinsic reward and curiosity to the agent, so that it will explore a bit more in the parking lot. With gamma as 0.99 the curiosity and the reward diminish every time.
Training Using Generative Adversarial Imitation Learning(GAIL):
For the AI to understand the environment early on, we can provide some demos, so we decided to record 100 demo instances to provide our AI. We played a 100 episode and provided the recorded model for learning by adding its path to the PPO hyperparameters while training.

```
gail:
          strength: 0.02
          gamma: 0.99
          encoding_size: 128
          use_actions: true
          demo_path: ../demos/ExpertParker.demo
```

Results

We have uploaded the inference of our best trained AI on YouTube.
Training 1. PPO without GAIL, Denoted by the Green Line
Training 2. PPO With GAIL, Denoted by the Blue Line

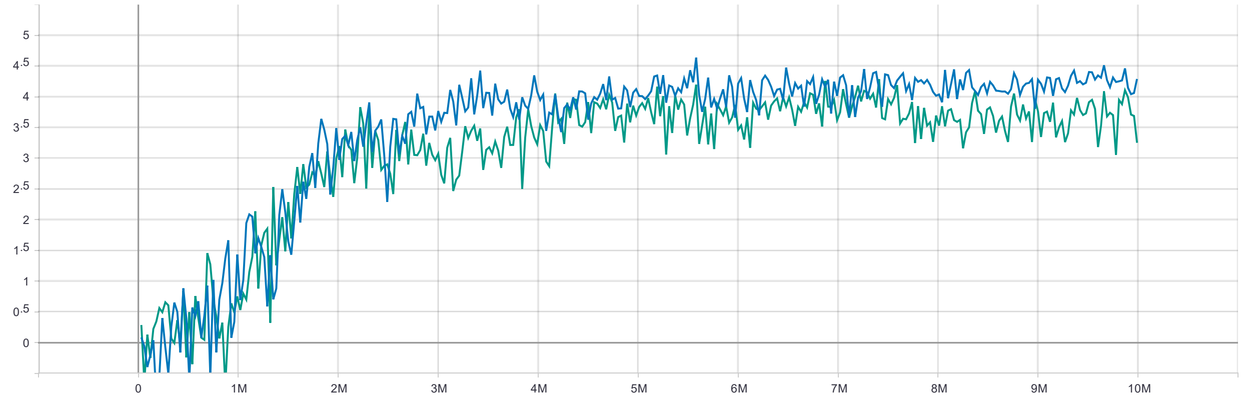


Figure 16 Parking Environment Cumulative Reward

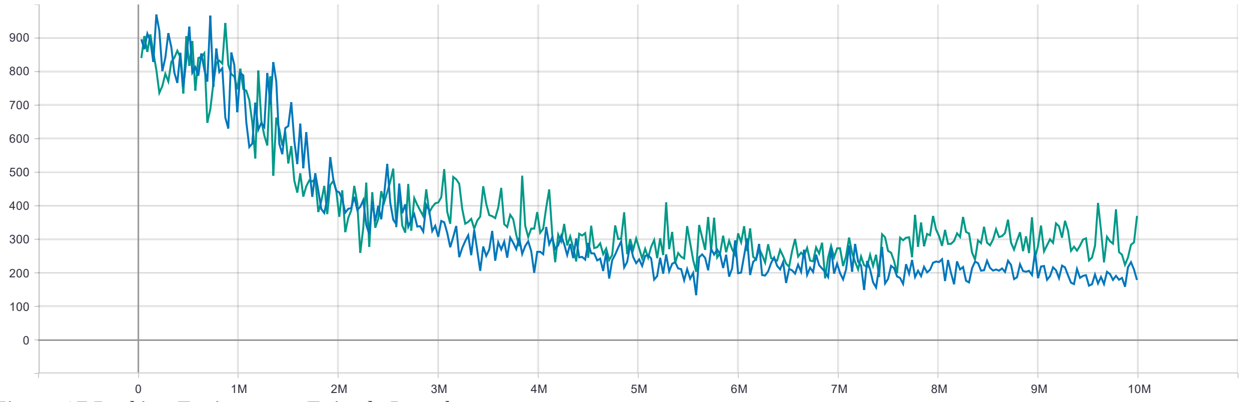


Figure 17 Parking Environment Episode Length

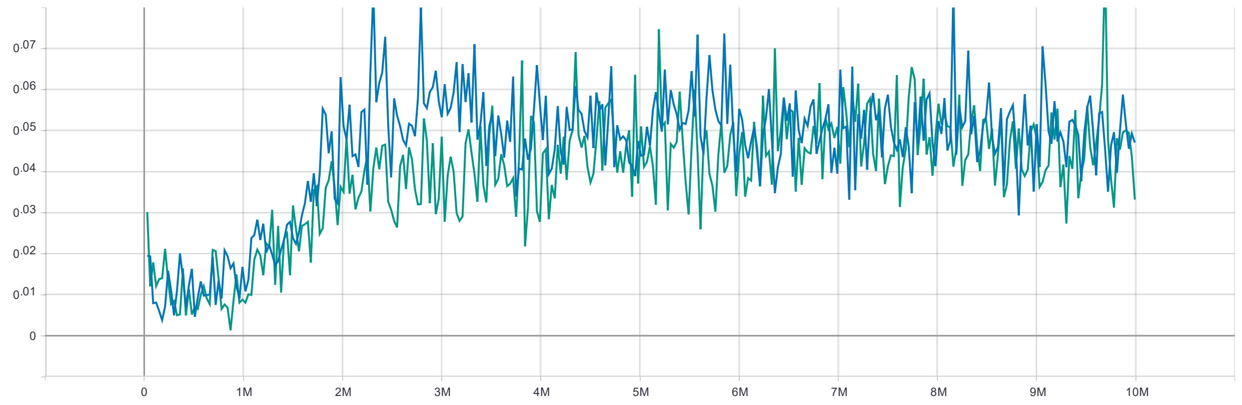


Figure 18 Policy Value Loss

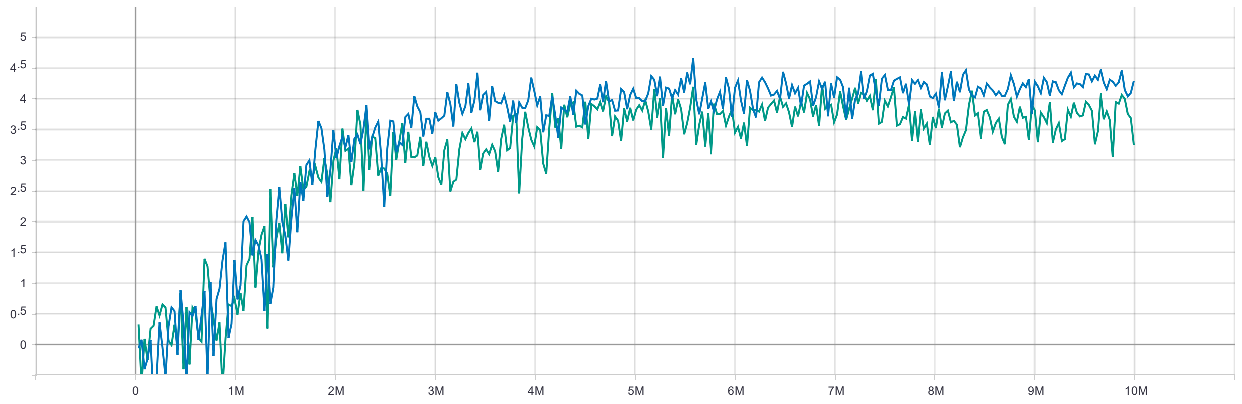


Figure 19 Policy Extrinsic Reward

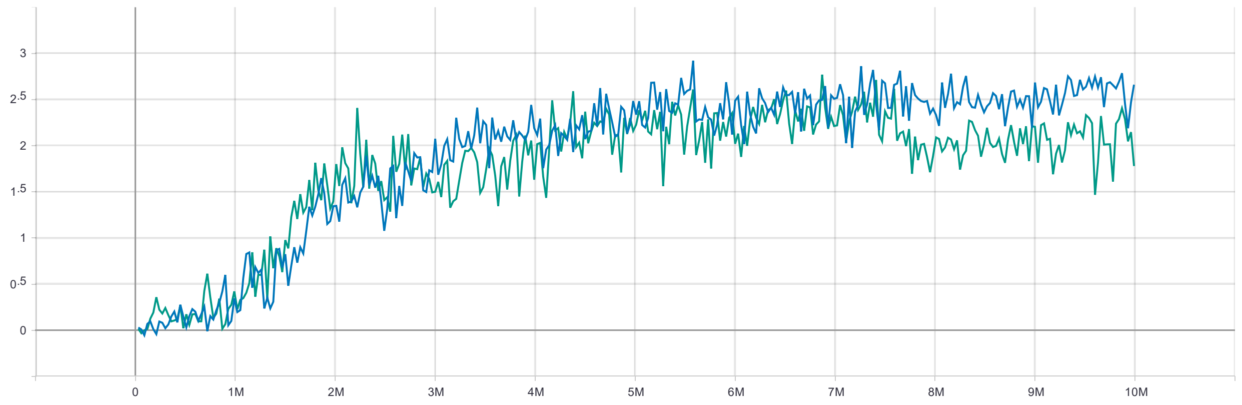


Figure 20 Policy Extrinsic Value Estimate

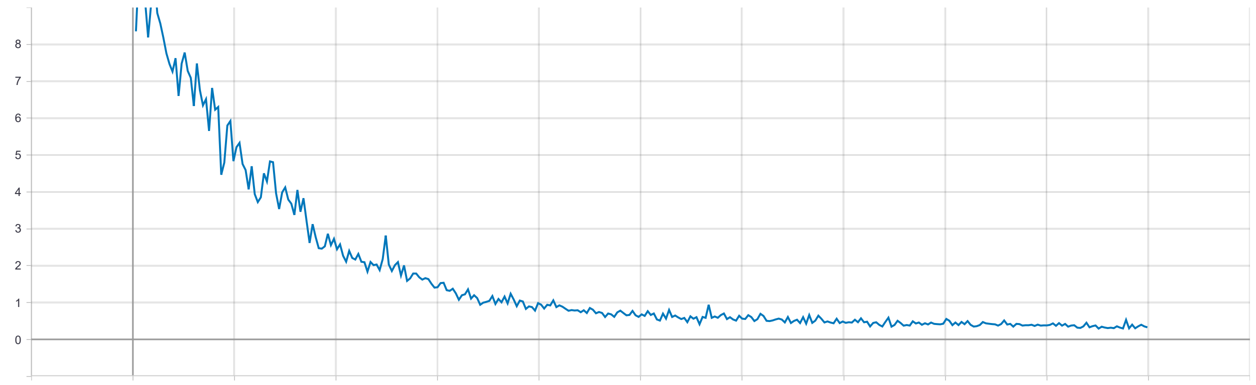


Figure 21 Policy GAIL Reward

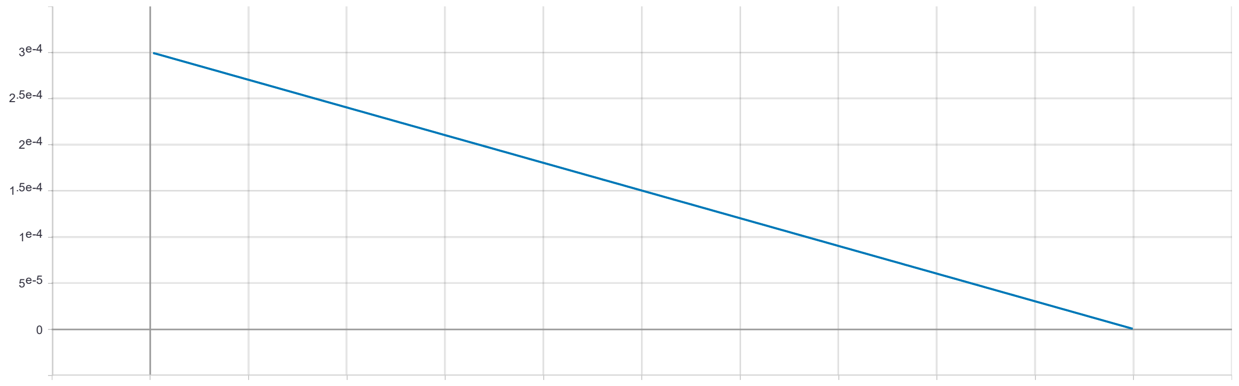


Figure 22 Policy Learning Rate

Analysis of the results

1. Cumulative Reward:
   - After 30k Steps
     1. Training 1: 0.2871
     2. Training 2: 0.08488
   - After 9.99M Steps
     1. Training 1: 3.252
     2. Training 2: 4.29
2. Episode Length: Should be decreasing, which would mean that the AI is finding the parking spot faster
   - After 30k Steps
     1. Training 1: 840.2
     2. Training 2: 896.2
   - After 9.99M Steps
     1. Training 1: 370

ii. Training 2: 179.1

- The Episode length must increase as we train, as we want the agent bird to survive as long as possible

3. Value Loss: Correlates to how well the model is able to predict the value of each state. This should increase while the agent is learning, and then decrease once the reward stabilizes. We can train this agent for even longer time, as the value loss hasn't decreased significantly
4. Policy Extrinsic Value Estimate
    - The mean value estimate for all states visited by the agent. Should increase during a successful training session.
5. Policy Extrinsic Reward
    - Should increase during a successful training session
6. GAIL Reward
    - Should decrease, as the AI stopped taking inference from our demonstrations and learned on his own
7. Learning rate
    - How large a step the training algorithm takes as it searches for the optimal policy. Should decrease over time.
    - Coincides for both the trainings

Conclusion

During this project, we tried training the AI with multiple configurations and hyperparameters for getting better performance out of the Proximal Policy Optimization algorithm with and without Generative adversarial imitation learning and Soft actor critic as well.
Where SAC is better when it comes to lesser training time, we found out that PPO albeit time consuming to train gives out better performance.
Unity engine provides a unique way of simulating the real-life environment virtually. We can simulate the problems for training and with the help of robotics we can implement the trained model.
With improvements in sensor technology, image recognition and computing there will be a day when machines will be autonomous and will have the ability to learn their actions based on reinforcement learning.

Works Cited

Bartow, R. S. (2018). *Reinforcement Learning, An Introduction* (2nd ed.). Cambridge, Massachusetts: The MIT Press.

Collaborative. (2020). *Artificial Neural Network*. From Wikipedia: https://en.wikipedia.org/wiki/Artificial_neural_network

DeepMind. (2015, October). *AlphaGo | DeepMind*. From deepmind.com: https://deepmind.com/research/case-studies/alphago-the-story-so-far

Haarnoja, T., Pong, V., Hartikainen, K., Zhou, A., Dalal, M., & Levine, S. (2018, December 14). *Soft Actor Critic—Deep Reinforcement Learning with Real-World Robots*. From BAIR, Berkeley: https://bair.berkeley.edu/blog/2018/12/14/sac/

IBM. (n.d.). *IBM 100 - Deep Blue*. From IBM: https://www.ibm.com/ibm/history/ibm100/us/en/icons/deepblue/

OpenAi. (2018). *Part 1 Key Concepts in RL | Spinning Up Documentation*. From OpenAi: https://spinningup.openai.com/en/latest/spinningup/rl_intro.html

Schulman, J., Klimov, O., Wolski, F., Dhariwal, P., & Radford, A. (2017, July 20). *Proximal Policy Optimization*. From OpenAi: https://openai.com/blog/openai-baselines-ppo/

Unity. (n.d.). *PPO Hyperparameters Documentation*. From Github: iv. https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Training-PPO.md#hyperparameters

Unity. (n.d.). *Unity Real-time Developement Platform | 2D, 3D & AR Visualizations*. From Unity.com: https://unity.com/

Additional References

i. The Free Car Asset Pack we used,
https://assetstore.unity.com/packages/3d/vehicles/land/simple-cars-pack-97669

ii. Link to the ML-Agents Documentation
https://github.com/Unity-Technologies/ml-agents/

iii. Link to the YouTube video of the inference we ran on the Parking Environment
https://youtu.be/0G5L-2T0a5s

iv. Hyperparameter Description from GitHub
https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Training-PPO.md

v. PowerPoint Presentation for this project
Link to the google drive file

vi. Complete package of this project
Link to the google drive folder

Appendix

This section of the report contains the instructions for installation, environment and agent setup screenshots, and training commands

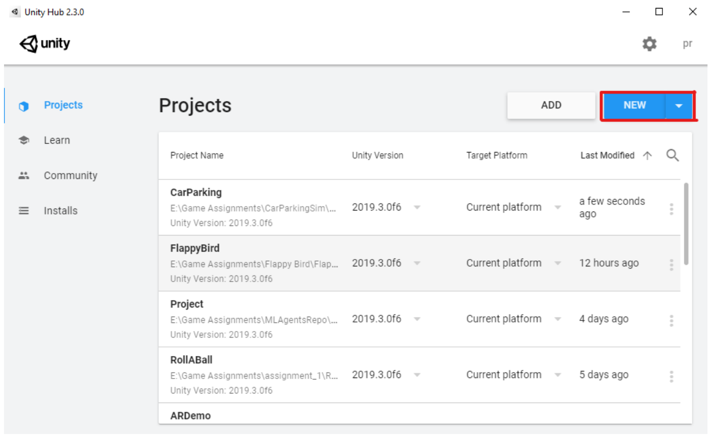
Installations

There are multiple number of installations for the tools necessary in the project

1. Unity
   Go to https://unity3d.com/get-unity/download,
   Click choose your unity + download, once inside go to the individual tab, and select personal use, please select a version of unity 2019.3 or afterwards
2. Python 3.7
   Go to https://www.python.org/downloads/release/python-370/
   Select the appropriate download according to your computer's operating system, Once installed make sure to install pip,
   If you're using Windows OS, you might want to add pip and python to the environment variables
3. Unity ML-Agents
   Run `pip install mlagents` in the terminal or command prompt window,
   Download the GitHub repository for ML-Agents to your computer, https://github.com/Unity-Technologies/ml-agents

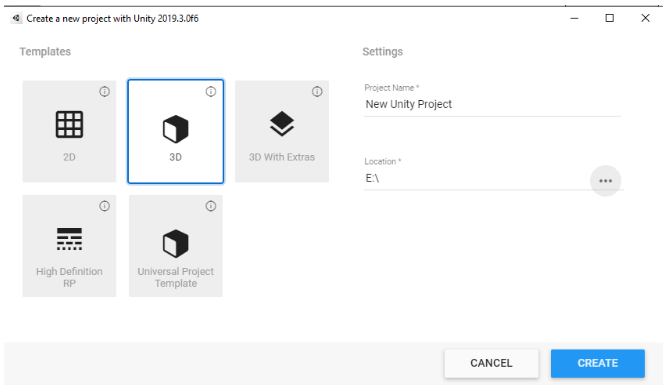
Setup

Once everything is installed properly, open Unity Hub and start a new project, on the next screen select 3D and give name and the directory for the project



Project Name	Unity Version	Target Platform	Last Modified
CarParking E:\Game Assignments\CarParkingSim\... Unity Version: 2019.3.0f6	2019.3.0f6	Current platform	a few seconds ago
FlappyBird E:\Game Assignments\Flappy Bird\Flap... Unity Version: 2019.3.0f6	2019.3.0f6	Current platform	12 hours ago
Project E:\Game Assignments\MLAgentsRepo\... Unity Version: 2019.3.0f6	2019.3.0f6	Current platform	4 days ago
RollABall E:\Game Assignments\assignment_1\R... Unity Version: 2019.3.0f6	2019.3.0f6	Current platform	5 days ago
ARDemo			

Figure 23 Open a new Project


Create a new project with Unity 2019.3.0f6
Templates
2D
3D
3D With Extras
High Definition RP
Universal Project Template
Settings
CANCEL
CREATE

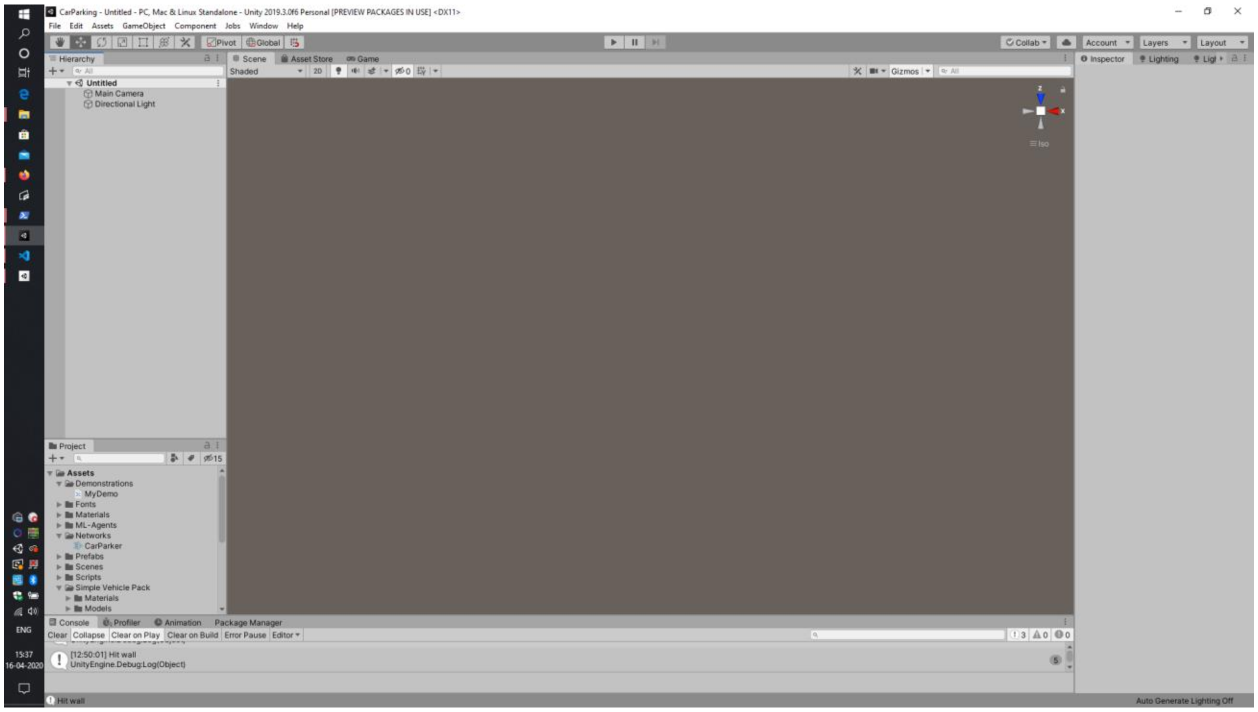
Project Name *
New Unity Project

Location *
E:\

Figure 24 Select 3D, then enter the name and location

Click “Create” and let it load.

Once opened you will see a screen like *figure 25*.



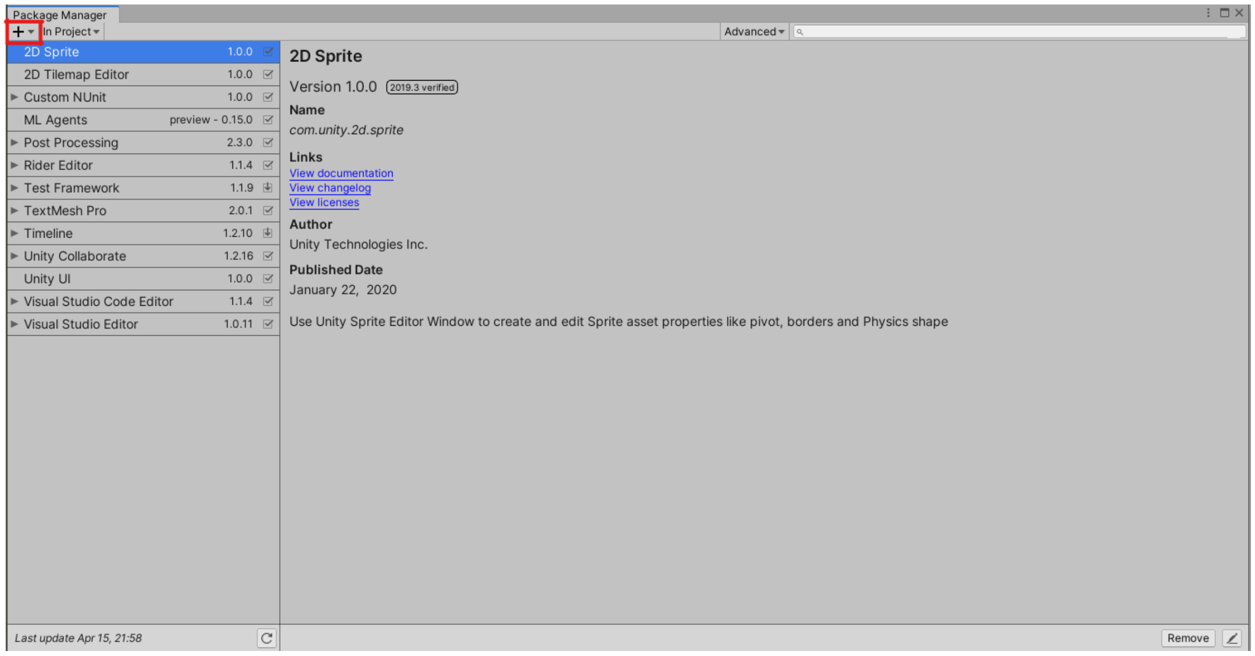
CarParking - Untitled - PC, Mac & Linux Standalone - Unity 2019.3.0f6 Personal [PREVIEW PACKAGES IN USE] <DX11>

[12:50:01] Hit wall
UnityEngine.Debug:Log(Object)

Hit wall

Figure 25 Screen after the project opens

Go to Window tab > Package Manager
It will open up a new pop window



Package Manager

In Project

Advanced

2D Sprite	1.0.0	✓
2D Tilemap Editor	1.0.0	✓
Custom NUnit	1.0.0	✓
ML Agents	preview - 0.15.0	✓
Post Processing	2.3.0	✓
Rider Editor	1.1.4	✓
Test Framework	1.1.9	↓
TextMesh Pro	2.0.1	✓
Timeline	1.2.10	↓
Unity Collaborate	1.2.16	✓
Unity UI	1.0.0	✓
Visual Studio Code Editor	1.1.4	✓
Visual Studio Editor	1.0.11	✓

Last update Apr 15, 21:58

2D Sprite

Version 1.0.0 2019.3 verified

Name
com.unity.2d.sprite

Links
View documentation
View changelog
View licenses

Author
Unity Technologies Inc.

Published Date
January 22, 2020

Use Unity Sprite Editor Window to create and edit Sprite asset properties like pivot, borders and Physics shape

Remove

Figure 26 Package Manager Screen

Click on the plus sign on the left top corner, and select "Add package from disk...", Navigate to the ml-agents repository downloaded from the GitHub, inside the directory called “com.unity.ml-agents” select the package.json file inside it and click Open.

This will let unity know that the ml-agents package and import necessary assembly information for the Agent and other classes.
Delete

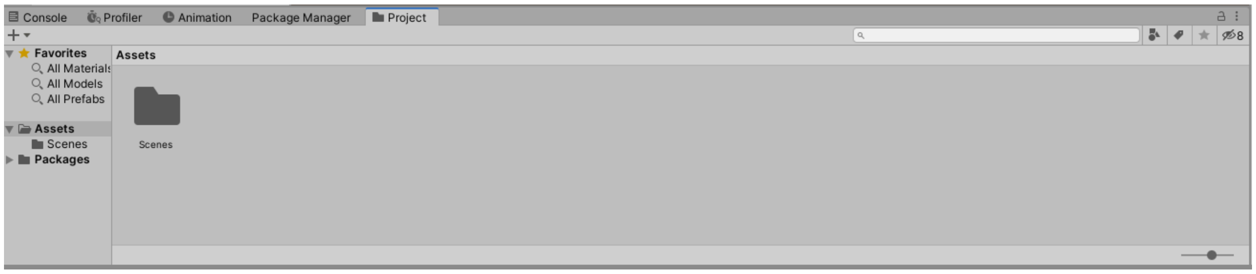


Figure 27 Assets in the project located at the bottom of the screen

The scenes directory from the project assets
Now open the project files for this project in file explorer, and drag and drop the following folders inside on the assets space

- Materials, Networks, Prefabs, Scenes, Scripts, Simple Vehicle Pack and Textures

Making the assets look something like this

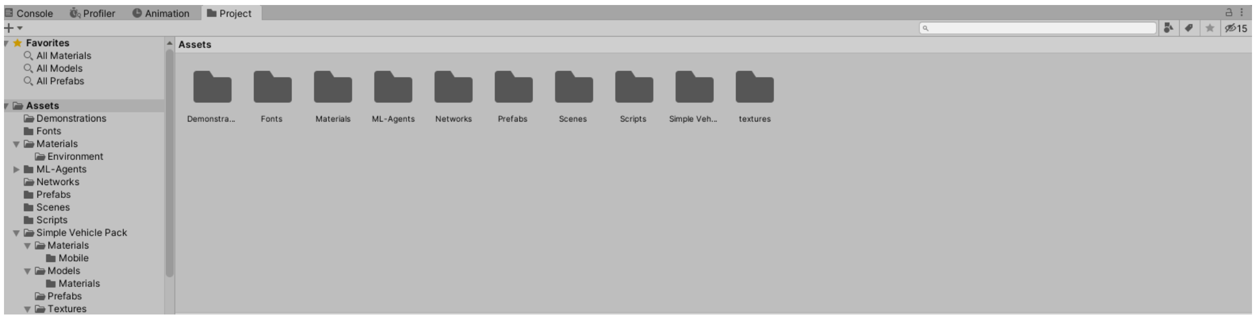


Figure 28 Project Assets after copying the project files

Directory Structure:
Demonstrations:
Contains the demonstration recorded for GAIL
Materials:
The Tramac material used for the parking lot floor, also the texture and color for the material
Networks: Trained Neural Networks by us

Prefabs: These are the environments that we created in order to test out which environment performs better, Area is kind of a huge parking lot with 4 rows of parked vehicles, whereas AreaOne is the one we used for actual training the agent.
Scenes: The scene that we are using for this project is called NewParking, referring to the prefab called AreaOne. The other scene which is called SampleScene is the scene with the bigger parking lot.
Scripts: Contains all the scripts used for configuration of the project
Simple Vehicle Pack: contains all the low poly car models and textures that we used for cars in the project

Running inference

Now that everything is imported and set up, double click on the NewParking scene inside the scenes directory to open it in the editor. And you will be able to see the following screen like *figure 29*

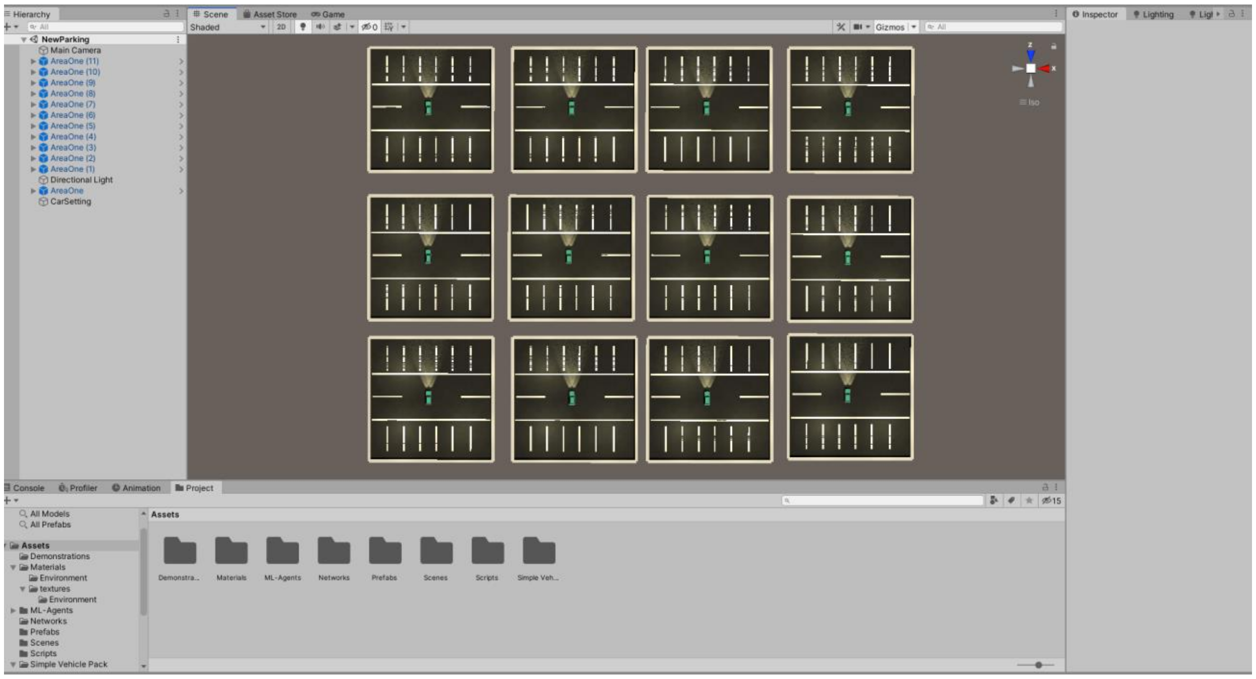


Figure 29 Scene View of the NewParking scene

Just click the play button in the middle top part of the editor screen and watch the AI park the car.
Note: similar steps are to be followed for the flappy bird AI and running the SampleScene.

Training

If you wish to train the AI on your own machine, follow these steps

1. Using terminal or CMD, navigate to the “config” directory inside the ml-agents repository you downloaded,
2. Run the following command for training with PPO
```
mlagents-learn trainer_config.yaml --run-
id=<enterTheNameOfTraining> -- train
```
3. Press the play button on the unity editor, and watch it train

Note: Make sure you add the hyperparameters in the trainer_config.yaml file, the configuration files are included with the submitted project package, inside the configurations folder

4. Similarly, you can train using the SAC as well as the GAIL configuration by adding the hyperparameters
5. For Realtime visualization during training using TensorBoard
   In the same directory, run the following command, summary frequency will be the hyperparameter that will generate summary after the given number of steps
```
tensorboard --logdir=summaries --port=6006
```