AI Agents in a Parking

Mahtab Dahaghin 4850232

mahtab.dahaghin@gmail.com

Abstract

The purpose of this project is to create AI agents (cars), which learn to park in a 3D parking lot. The implementation is by Unity's ML-Agents framework. The agents are implemented by deep neural networks which are trained with Proximal Policy Optimization (PPO) algorithm. The implementation starts with a simpler model in which there is only one agent, and in the end by improving the code, it becomes possible to increase the number of agents.

1: Introduction

1.1: Reinforcement Learning

Reinforcement learning is related to the dynamic scenarios where we have autonomous agents that are embedded in an environment and learn to accomplish some tasks. Their learning progress is by interacting with their environment, without any human supervision and any provided datasets.
In reinforcement learning, they are going to be provided with the data in the form of what is called *"state-action"* pairs. The *states* are the observations of the agent or the inputs to the system. The *actions* are the actions that the agent wants to take in its environment. The goal of the agent is to take actions that maximize its reward. At each step, the agent executes an action, observes a new state, and receives the reward according to its action. [Figure 1]

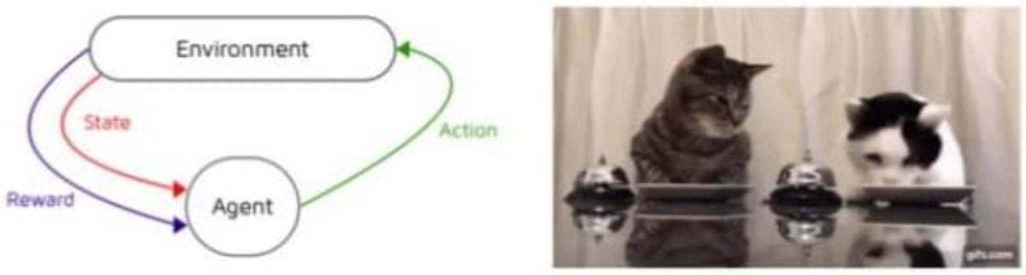

Figure 1 – Reinforcement Learning Framework [1]

1.1.1: The PPO Algorithm

One of the problems in reinforcement learning is that the training data which is generated by the agent by interacting with the environment is itself dependent on the current policy. Data distributions of agent’s observations and rewards are constantly changing as the agent learns, which causes instability in the whole training process. Another problem in reinforcement learning is that it is very highly sensitive to hyperparameter tuning and initialization.
To address these kinds of problems, OpenAI designed a new reinforcement algorithm that is called Proximal Policy Optimization (PPO) [2]. The main purpose of PPO is to strike a balance between ease of implementation, sample efficiency, and ease of tuning.
Unlike some popular approaches that can learn from stored offline data, PPO learns online. This means that it does not use a buffer to store past experiences. Instead, it learns directly from what the agent encounters in the environment. Once a batch of experience has been used to do a gradient update, then the experience is discarded and the policy moves on.

1.2: The Unity Platform

Unity is a real-time 3D development platform that consists of a rendering and physics engine as well as a graphical user interface called the Unity Editor. A Unity Project consists of a collection of *Assets*. *Scenes* are a special type of *Asset* that defines the environment or level of a Project. *Scenes* contain a definition of a hierarchical composition of *GameObjects*, which correspond to the actual objects (either physical or purely logical) within the environment. The behaviours and functions of each *GameObjects* are determined by the components attached to it. It is also possible to define custom components using C# scripts or external plugins. [3]

1.2.1: The Unity ML-Agents Toolkit

The Unity Machine Learning Agents Toolkit (ML-Agents) is an open-source project that enables games and simulations to serve as environments for training intelligent agents. It enables researchers and developers to create simulated environments using the Unity Editor and interact with them via a Python API. Researchers can use the provided Python API to train Agents using reinforcement learning, imitation learning, neuroevolution, or any other methods. The ML-Agents Toolkit is mutually beneficial for both game developers and AI researchers as it provides a central platform where advances in AI can be evaluated on Unity’s rich environments and then made accessible to the wider research and game developer communities.

2: Related Work

The basic idea for this project came from a GitHub repository that implemented a simple parking environment [Figure 2]. This project is built by Unity, using ML-Agents. The purpose of this project is to simulate an intelligent car to park in a simple parking environment, which has only one available parking lot. The start position of the car is the same at the beginning of each episode and also parking lots are always on one side of the parking.

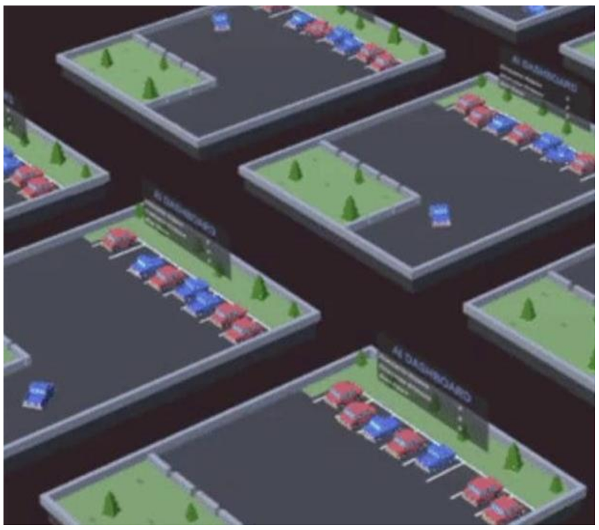
Figure 2 – The Base of the Project

In this implementation, the observations are the local position of the agent, its rotation, its velocity, and also the position and the rotation of the goal. The available actions for the agent are moving forward, backward, turning right, turning left, or stay idle.

3: Implementation

3.1: Problem Statement

In this project, are implemented two scenarios for the parking problem. The implementations are flexible and they can be modeled for more agents too.
In the first scenario, only one car is situated in the environment. The car is trained to park in an available parking lot. The first position of the car is changing randomly at the beginning of each episode. Also, the number of available parking lots and their positions are changing in every episode randomly in a range between a minimum and a maximum number. Therefore, in every episode, the car learns to solve a different problem by facing a new model of the environment. [Figure 3]
In the second scenario, two agents are situated in the environment. They are trained to park in a parking environment with more than two free parking lots. The first positions of both cars are changing randomly at the beginning of each episode. Also, the number of available parking lots and their positions are changing every episode. [Figure 4]

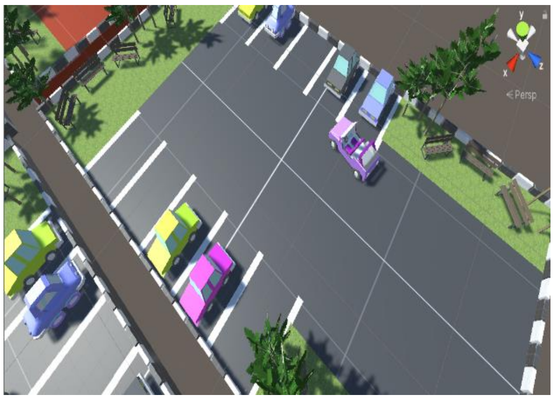
Figure 3 – First Scenario with One Car

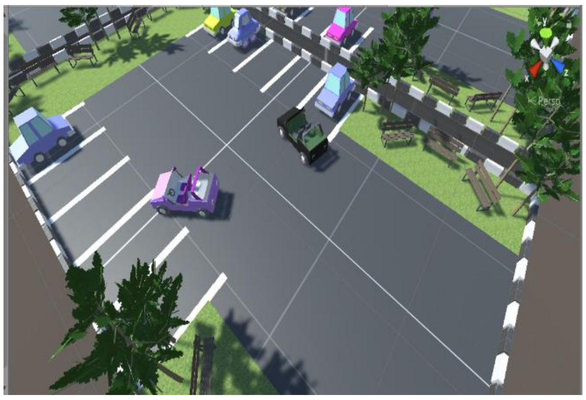
Figure 4 – Second Scenario with Two Cars

3.2: Unity prefabs

For creating the environment, three free assets are used from the Unity asset store. The first one is the Simple City Plain package. This package is used for implementing some of the prefabs of the parking, like the grass, benches, and trees. The second package is the *Gridbox Prototype Materials*. This package is used for its materials for success, failure, and default. The third package is *Cartoon Car-Vehicle* Pack. This pack is used for the parked cars in the parking, and also for the agents.

The environment consists of the following prefabs:

- **Road:** This prefab shows the ground material (grey color for the ground of the parking). Whenever the agents park successfully, its color changes to green, and whenever the agents have too many mistakes its color changes to red.
- **Barriers:** This prefab surrounds the parking and contains all the barrier blocks. Each block has its *box collider* and "*CarObstacle*" script. Whenever the agent hits one of these blocks, the "*CarObstacle*" script calls the "*TakeAwayPoints*" function of the "*CarAgent*" script to give a negative reward to the agent.
- **AllGrass:** This is a prefab that shows the grass parts in the parking. Whenever the agent goes on the grass, the "*CarObstacle*" script which is added to them, calls the "*TakeAwayPoints*" function of the "*CarAgent*" script to give a negative reward to the agent.
- **GoalFinal:** This prefab is an empty object, which has a *box collider* and the "*CarGoal*" script. *CarGoal* script has an *OnTriggerEnter* function, that whenever an agent collides the prefab, it gives the agent a positive reward. If the rotation of the agent around the y-axis is about 90 degrees (by considering an error), the parking action of the agent becomes accepted and it will get the final reward. In the end, the *CarGoal* script destroys its gameobject.
- **AgentsController:** This prefab is an empty object, which contains the "*AgentsController*" script. This script has a function that is called at the beginning of each episode. This function randomizes the position of each agent at the beginning of each episode. Also, it destroys the previous goals, and then it calls the "*Setup*" function of the "*CarSpot*" script, to reset the parking spots.

- **CarSpots:** This prefab contains all the parked cars as its children. Whenever the "*Setup*" function of its script (*CarSpot*) is called, it chooses a random number for free parking lots. Then it chooses random places for the goals, makes the parked car of that place disable, and creates "*GoalFinal*" prefab instead of them.
- **CarAgent:** This prefab has a box collider for realizing whenever it collides with other objects in the environment. Also, it has a *rigidbody* component, to seem more realistic. Whenever two agents collide with each other, they will take a negative reward by *OnTriggerEnter* function of their *CarObstacle* script. This prefab also has another script named *CarController*. In this script, after that the agent decides the action, it applied those actions and changes the position or rotation of the agent.

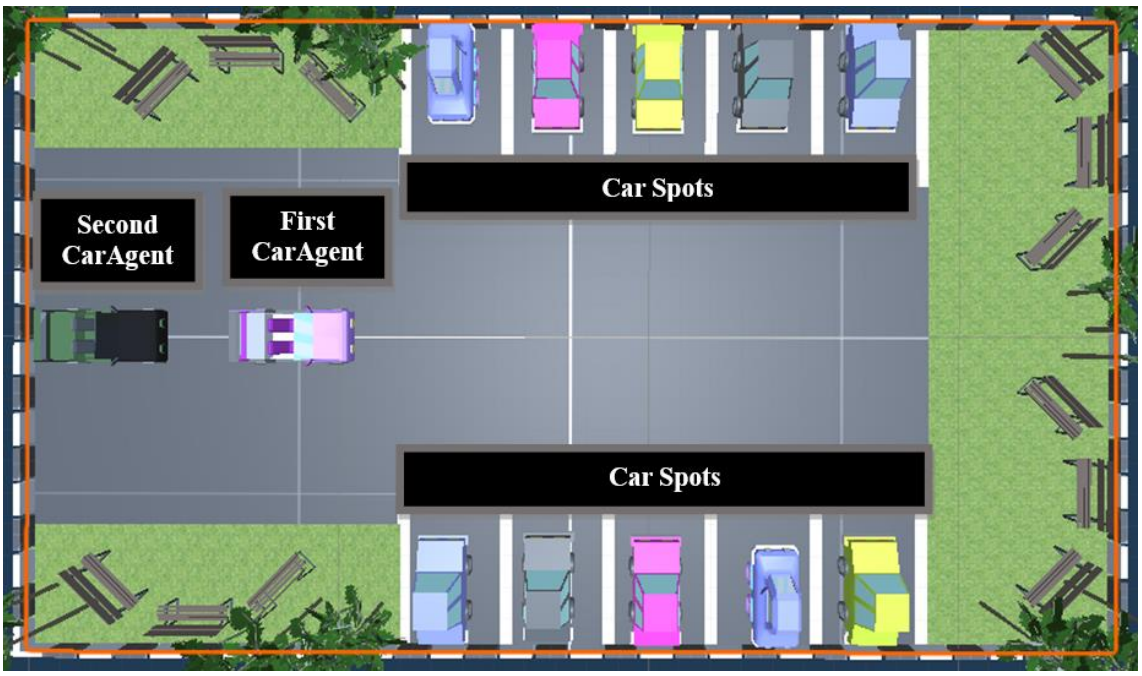

Figure 5 - Prefabs

3.3: Agent Configuration

3.3.1: Decisions

Each time the agent requests a decision, the observation-decision-action-reward cycle repeats. Agents will request a decision by calling the *Agent.RequestDecision*() function. The car agent has the *Decision Requester* component, which able the agent to request decisions on its own at regular step intervals. In this implementation, the *Decision Period* parameter for all the agents is set to 5.

3.3.2: Observations and Sensors

The observations should include all the information the agents need to accomplish their tasks. Without sufficient and relevant information, probably the agents may not learn at all. In this project, the provided ways for the agents to make observations are as follows:

- **Generating Observations**: The aspects of the environment which are numerical and non-visual are observed by overriding the *Agent.CollectObservations()* method and passing the observations to the provided *VectorSensor*. In this project, each of the agents has seven observations, which are the local position of the agent (3 observations), the rotation of the agent around the y-axis (1 observation), and the relative position of the nearest goal to the agent (3 observations).

- **Raycast Observations:** *Raycasts* are another possible method for providing observations to an agent. In this project, a *RayPerceptionSensorComponent3D* is added to each of the Agents. During observations, several are cast into the environment, and the objects that are hit determine the observation vector that is produced. The chosen degree for the *raycasts* is 180 and the ray length is 20. So the agents can observe all around them up to 20 meters far from them. [Figure 6]

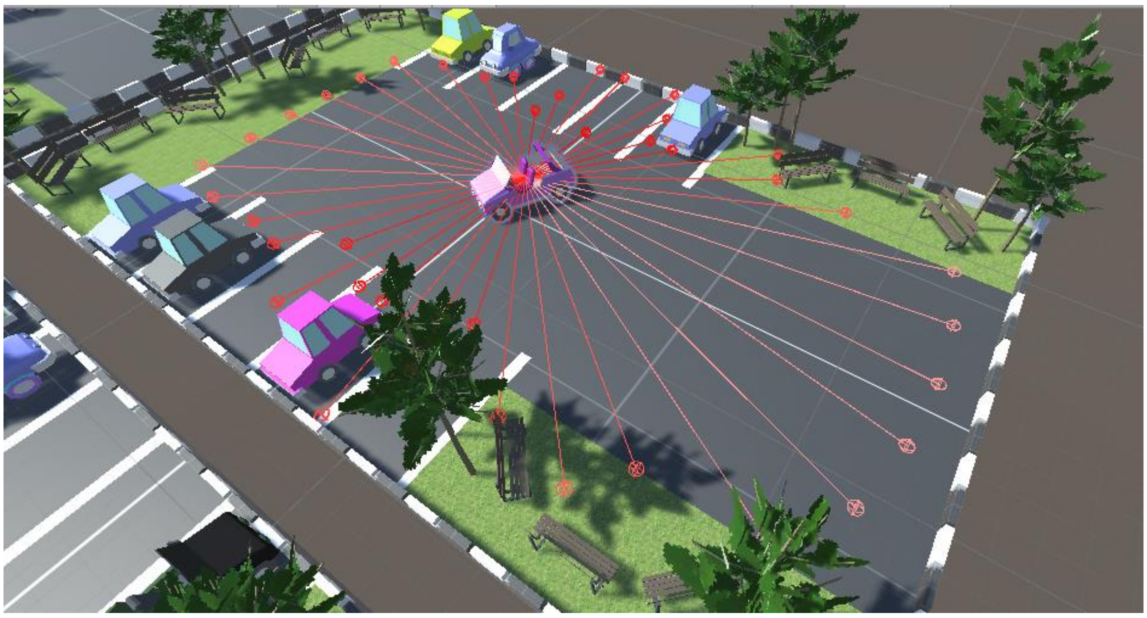

Figure 6 – RayCast

3.3.3: Actions and Actuators

The training algorithm simply tries different values for the action list and observes the effect on the accumulated rewards over time and many training episodes. In the *OnActionReceived()* function the actions for the Agent are defined. In this implementation the action type is discrete. The allowed actions which are defined in the *OnActionReceived()* function, are moving forward, moving backward, turning right, and turning left.

3.3.4: Rewards

The reward is a signal that the agent has done something right. The PPO reinforcement learning algorithm works by optimizing the choices an agent makes such that the agent earns the highest cumulative reward over time. In this project, there are two kinds of rewards. The positive reward, for when the agent makes a good action, and the negative reward, force the agent to a sooner result. The positive rewards are for when the agent collides the goal. If it collides a middle goal, the reward is calculating according to the rotation of the agent around the y-axis. As the rotation value becomes nearer to the 90 or -90 degree, the reward becomes more. The second positive reward is when the agent collides the final goal with an acceptable rotation angle around the y-axis. The negative reward is calculating by the distance of the agent to the nearest goal.

3.4: HyperParameters

The below table shows selected hyperparameters and network settings for the two models. The selected parameters are the same for both pieces of training. The only difference is the batch size in the second training model.

Parameters	Description	One Car	Two Cars
Batch Size	The number of experiences used for one iteration of a gradient descent update.	32	64
Buffer Size	Corresponds to how many experiences should be collected before we do any learning or updating of the model.	12000	12000
Learning Rate	Corresponds to the strength of each gradient descent update step.	0.0003	0.0003
Beta	Corresponds to the strength of the entropy regularization, which makes the policy "more random."	0.001	0.001
Epsilon	Corresponds to the acceptable threshold of divergence between the old and new policies during gradient descent updating.	0.2	0.2
Lambda	Corresponds to the lambda parameter used when calculating the Generalized Advantage Estimate (GAE).	0.93	0.93
Number of Epochs	The number of passes through the experience buffer during gradient descent.	5	5
Hidden Units	Correspond to how many units are in each fully connected layer of the neural network.	128	128
Number of Layers	Corresponds to how many hidden layers are present after the observation input, or after the CNN encoding of the visual observation.	3	3

4: Results

The results of the two models are shown by two kinds of statistics.

- **Cumulative Reward:** The mean cumulative episode reward over all the agents with the same behaviour. The general trend in reward should consistently increase over time. Small ups and downs are to be expected.
- **Episode Length:** The mean length of each episode in the environment for all the agents with the same behavior.

All the training processes are done by a computer with an i7 8th generation and a GTX 1050 Ti.

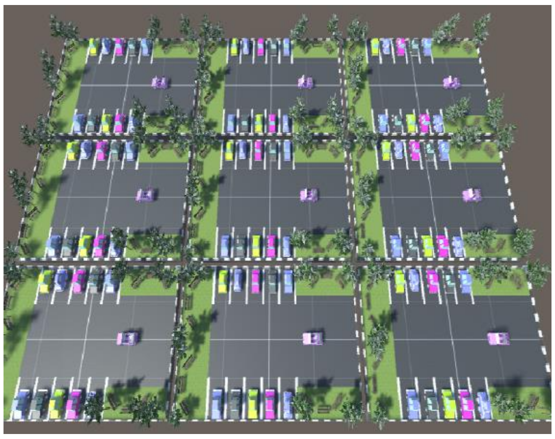
Figure 7 –Training Scene of the Model with One Car

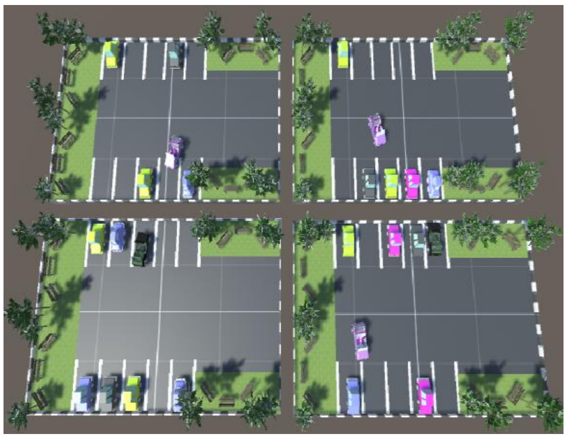
Figure 8 –Training Scene of the Model with Two Cars

4.1: Single Car

The first step for training the agent is implementing and designing the training scene. In this project, the training process is done simultaneously by nine parking environments. [Figure 7] The training took about two hours for about 1.4M steps. As we can see in Figure 9, the cumulative reward is increasing by increasing the steps. Also, from Figure 10, we can realize that the length of each episode is decreasing over time. This can be the cause of the negative reward, which the agents take for each action, and they try to reach the goal with minimum possible actions.

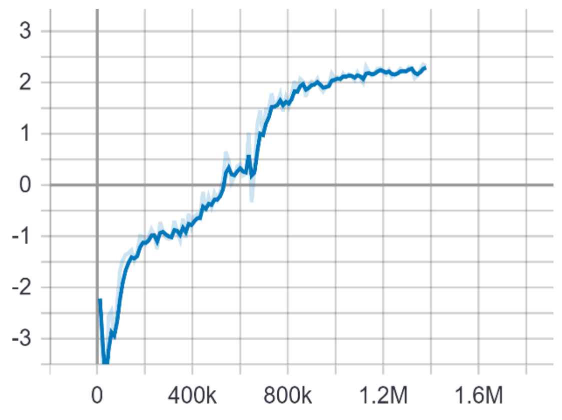

Figure 9 - Cumulative Reward

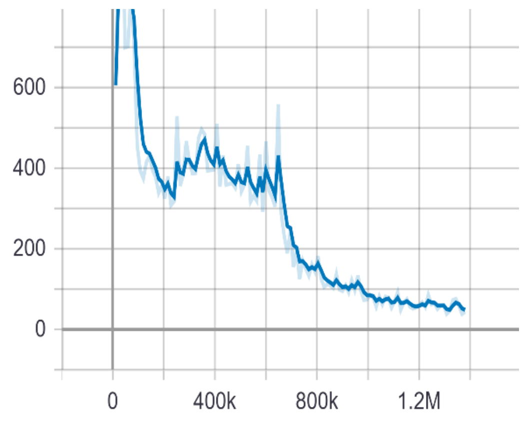

Figure 10 - Episode Length

4.2: Two Cars

The training scene of this model is completely different from the previous model. For training, the agents in this scenario are implemented in four training environments. [figure 8] In two of these environments, is trained only the first agent. In one of these environments, is trained the second agent, and in the last environment, is trained both of these agents simultaneously. The training took about eight hours. For the first car, the training is done in about 1.8M steps, and for the second car, the training is done in about 1.2M steps. As we can see in Figure 11, the cumulative reward for both agents is increasing by increasing the steps. Also, from Figure 12, we can see that the length of each episode is decreasing over time.

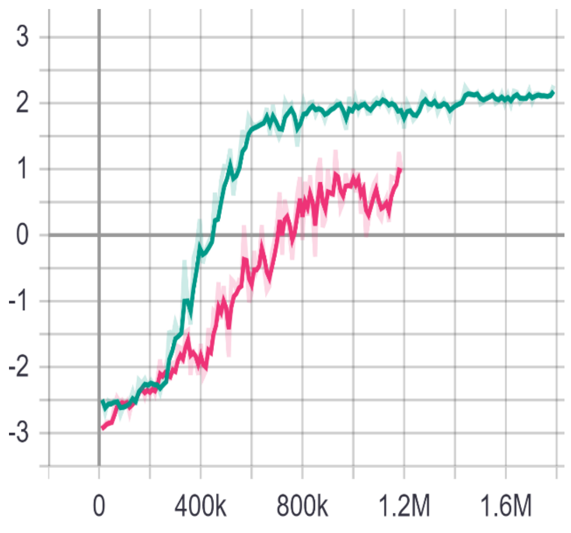

Figure 11 - Cumulative Reward

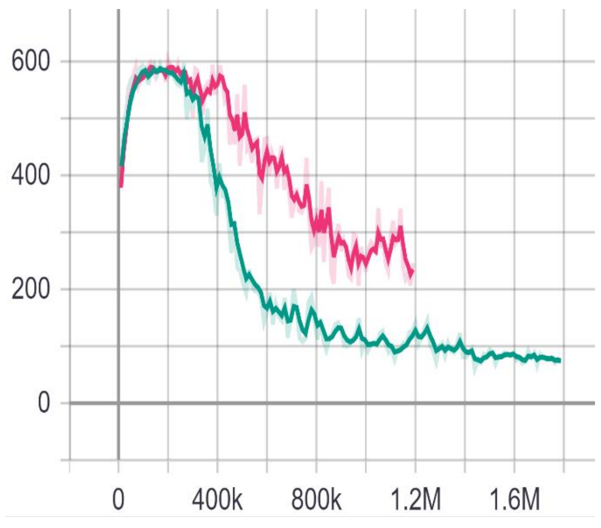

Figure 12 - Episode Length

5: Discussion and Future Work

Many different tests and experiments have been left for the future due to lack of time (i.e. the experiments are very time consuming, requiring many hours to finish a single run). The following ideas could be tested:

- It could be interesting to consider the parked cars in the parking, as agents too. So their purpose can be driving out of the parking, without crashing obstacles and other agents.
- It will be interesting to increase the number of agents and see how they learn to search for an available spot without crashing each other.
- It is also possible to change the parking environment during the training and see how agents learn to park in new parking environments.

References

[1] Introdution to Deep Learning Couse of the MIT university

[2] Schulman, J., Wolski, F., Dhariwal, P., Radford, A. and Klimov, O., 2017. Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347.

[3] Juliani, A., Berges, V.P., Teng, E., Cohen, A., Harper, J., Elion, C., Goy, C., Gao, Y., Henry, H., Mattar, M. and Lange, D., 2018. Unity: A general platform for intelligent agents. arXiv preprint arXiv:1809.02627.