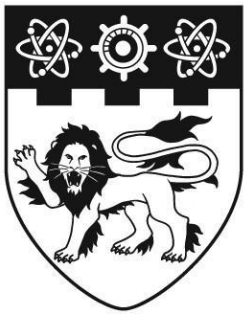

NANYANG TECHNOLOGICAL UNIVERSITY

Deep Reinforcement Learning for Autonomous Car Application

Submitted by: Faza Muhammad Allam

Matriculation Number: U1720631B

Supervisor: Dr Tan Chee Wah, Wesley

RI Co-Supervisor: Dr Jing Wei. Institute of Infocomm Research (I2R)

School of Electrical & Electronic Engineering

A final year project report presented to the Nanyang Technological University in partial fulfilment of the requirements of the degree of Bachelor of Engineering

2020/2021

Table of Content

Abstract

Autonomous car development has been increasing in recent years. Everyone tried to improve the level of automation of autonomous cars, and also to increase the convenience and safety of the user. With the emergence of faster communication networks and Internet of Things (IoT) devices, it this can be used to develop autonomous cars that can communicate with another autonomous car. Research has been focused on connecting autonomous cars with each other which allows cars to share their data such as velocity and position, and they are known as Connected Autonomous Vehicle. In this project, we will try to develop the concept of autonomous cars using several simulation strategies based on the concept of Connected Autonomous Vehicle, by solving simulations of an autonomous car driving through a highway, and parking situation with information from another car and its surrounding. We proposed our solution using reinforcement learning approach with the addition of an attention layer. The result shows a successful attempt in the simulation, with improved performance compared to previous experiment.

In this report, the theoretical basis on the design and formula will be discussed. Along with that, we will examine the effectiveness of each algorithm on previous reinforcement learning problems, and test the performance of each algorithm before conducting our main experiment. Each experiment will be compared with previous experiments to show the effectiveness of the reinforcement learning algorithm.

Acknowledgment

I would like to express my utmost gratitude to the following people for guiding me throughout this project.

Firstly, I would like to express my deepest gratitude to Dr Tan Chee Wah, not just for guiding me and supervising me in this project, but also as a mentor who gives life lessons and supports me throughout this project. Dr Tan shows me the importance of research which motivates me in doing this project, and He always shows his enthusiasm while guiding his students. There are many times when Dr Tan helps me check the preparation details of presentation and video recording.

Next, I would like to thank Dr Jing Wei who guided me from the beginning of the project, and helped me to understand the fundamental aspects in this project. With his expertise in this field, his suggestion helped me in doing this project.

I would like to thank Mr. Edouard Leurent, as the writer of the paper that I cite in this project. From our discussion, his direct advice really helped me in this project and in making significant advancement, not just in this project, but also in understanding the fundamental concepts.

Last but not least, I would like to express my gratitude to my parents, all professors, and friends for their support, encouragement, and valuable advice. With their support, I am able to complete this project and any other stuff during my university life.

Faza Muhammad Allam
May 2021

Acronyms

RL	Reinforcement Learning
DRL	Deep Reinforcement Learning
PPO	Proximal Policy Optimization
DDPG	Deep Deterministic Policy Gradient
HER	Hindsight Experience Replay

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Autonomous vehicle is one of the hottest fields now. Everyone is trying to upgrade the level of automation to increase the level of automation from level 3 to level 4 such as automatic steering. To increase the autonomous level, one way to do so is by using Connected Autonomous Vehicles [1]. It is a concept where every vehicle can communicate with each other and even with the road itself using IOT devices. Every vehicle can share and retrieve information from other vehicles, such as their positions, velocities, steering directions, and etc. This will help a vehicle to plan its trajectory, and improve the ability of the car to control the steer and trajectory.

Most research and development in this area has been focused on supervised learning, which learns from pre-collected data to determine the output. It uses images data from cameras that are attached on the vehicle, and some sensors such as LIDAR sensor which gives the distance from the car to the next object. However, connected autonomous vehicle areas are relatively new, which makes the research and data in this area limited. This will limit the ability of supervised learning that it needs a lot of data to make the vehicle achieve a high successful rate of automation.

Reinforcement Learning (RL) [2] has emerged as one of the top research areas in Machine Learning alongside Supervised Learning and Unsupervised Learning. Instead of learning from pre-collected data, RL agents learn from experience data that was collected by the agent itself. RL agents learn while gathering the data so there is no need to pre-collect data and then learn from the data like supervised learning. The experience itself can be directly from doing the experiment, or even using simulation that can be transferred to real word problems. The recent success of Openai 5 to beat professional Dota 2 players [3] has improved the popularity of Reinforcement Learning and increased the amount of research in this area.

RL consists of an agent that tries to complete tasks by reinforcing learning from rewards given by the environment after taking some actions using information given by the environment. Agents need to pick an action that tries to maximize the rewards given by the environment. By reinforcing learning from previous rewards and actions, an agent will improve its ability to generate better actions.

Current research in RL has been focused on Deep Reinforcement Learning which implements deep learning architecture to reinforcement learning agents. Many complex problems have been solved using various DRL algorithms, especially human control tasks like controlling robots, playing Atari games, and training autonomous driving cars.

1.2 Motivation

Aradi et al. [4] has conducted research to implement DRL algorithms into self-driving cars on highways using popular Policy Gradient methods. Assuming the car uses a high level sensor that can detect its position relative to its surrounding, it managed to solve the problem for the car to go through traffic on highways and avoid collision with another car. Following its success on autonomous cars, this study advances beyond previous study by implementing reinforcement learning to the concept of connected autonomous cars. It gives agents more knowledge of their surroundings with more accurate information and faster retrieving time than using sensors. This will improve an agent's reliability to choose action prior to a more complete information given by the environment. Moreover, user safety can be more assured with the avoidance of sudden action since agents retrieve information faster and are able to process it earlier. However, the research in the Connected Autonomous Vehicle area and reinforcement learning are relatively new and very limited research has been conducted in this area.

1.3 Objective

The aim of this study is to implement Reinforcement Learning algorithm into connected autonomous driving vehicles. Autonomous vehicle simulator is used to simulate an autonomous car in several situations such as going through highways, parking situations, and etc. It represents connected autonomous vehicle simulation where our agents receive information regarding themselves and their surroundings such as positions and velocities of themselves and from other cars.

1.4 Organisation

This report comprises of 5 chapters, which are:

- Chapter 1 provides the overall background of the project, including the motivations, objectives, and the scope of the project.
- Chapter 2 reviews the relevant studies and concepts used in this project, on top of past studies that are relevant to this project, which mainly focus on mathematical formulations and concepts.
- Chapter 3 introduces our proposed method, including the setup, graphical models of our network and how the experiment was conducted.
- Chapter 4 presents the results from the experiments conducted in this project. Discussions were also given regarding several aspects of the results such as the effectiveness of novel network modules and comparison to previous study.
- Chapter 5 concludes and summarizes the project. Evaluations and recommendations for potential future work are also given.

Chapter 2

Literature Review

2.1 Reinforcement Learning

Reinforcement Learning consists of an agent that tries to complete tasks by learning from experience. A reinforcement learning agent will be given rewards *r* from the environment after taking action *a* at some state *s*, which give information given by the environment. In order to solve a problem, agents will try to maximize total rewards in solving one problem or episode, by taking action in each step that leads to a maximized total reward. This total reward is called returns, denoted by *R*. Overall, RL problems can be denoted as Markov Decision Processes (MDP) [5]. The illustration of a RL process is shown in Figure 2.1.

There are two kinds of methods in reinforcement learning, the first one is policy based algorithm and the second is value based algorithm. Regarding policy based algorithms, agents choose actions by following policy $\boldsymbol{\pi}$ which is an algorithm or formula to choose an action. The agent will try to improve its policy to generate better actions that will give better returns for the agent. There are two kinds of policy, the first one is deterministic policy, where agents choose action directly from policy. This means agents will always choose similar actions from similar input states. The other policy is stochastic policy, where agents can pick different actions, given similar input states. The Policy will generate action probability distribution and agents will choose action based on randomly sampling from this probability distribution.

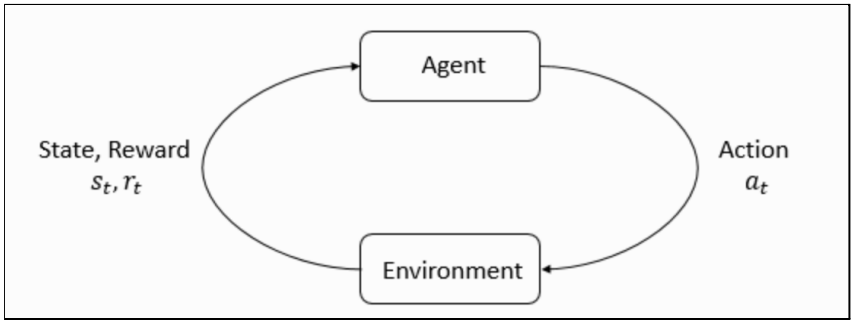

Figure 2.1. Overall Illustration of Reinforcement Learning

In contrast, for value based algorithms, agents use a function called value function to give a score on each action at every state. This value function is the estimated return for the agent on every action it takes. The agent will then take an action with the largest estimated return from the value function. The state-value (V) function gives the estimated return for a particular state achieved by the agent, where state-action value function (Q) denotes estimated returns for taking action *a* at a particular state *s*. They are given in the bellman equations [6]:

$$Q(s,a) = r(s,a) + \gamma Q(s',a') \quad (1)$$

$$V(s) = r(s) + \gamma V(s') \quad (2)$$

where *s* and *a* denote state and action taken at the current step, while *s'* and *a'* are state and action taken at the next step. The bellman equation can be expanded from a recursive formula that explains the behaviour of the agent that needs to take into account the future return to compute current estimated return. The terms γ is discount factor, varied from 0 to 1, which denote the important of future return with respect to the current return. It denotes how much we want to account future return into consideration of current return. By setting the discount factor to be high, that means the agent valued future reward more and will pick actions that maximize future returns. On the other hand, if the discount factor is small, the agent will assume that future return is not significant, and it will prefer to pick actions that lead to higher rewards at current state rather than actions that lead to higher future returns.

2.1.1 Bias and Variance

In reinforcement learning, Bias and Variance are not only explaining overall agent performance, but also how well reinforcement signal such as value function reflect the true reward signal. A naive approach to a RL learning algorithm would be to encourage actions which were associated with positive rewards, and discourage actions associated with negative rewards. Instead of updating our agent's policy based on immediate rewards, we would consider more actions (and the states of the environment when those actions were taken) which lead up to higher returns. There are a number of approaches for doing this, all of which involve doing a form of

credit assignment. Credit assignment will assign some credit to the series of actions leading to positive returns, and not just the most recent action.

In reinforcement learning, bias and variance describe both accuracy and stability of an agent. High bias results in stable learning but inaccurate value estimate. Meanwhile, high variance corresponds to noisy result, but with an overall accurate value function result. There are two methods in computing the estimated return in a Markov process [7].

1. **Monte Carlo Estimate**

   Monte Carlo Estimate uses historical data of the results from the previous episode, to estimate the total return of the state in the current episode. By relying on the historical data, the agent will have an accurate knowledge about the estimated returns from one episode, and the agent can compute estimated returns for this episode using the one that was retrieved from the previous episode. The estimated return will be more accurate since it does not use any value estimator. The estimate return of a state-action pair on current episode for Monte Carlo estimate is expressed by:

$$R(s,a) = \sum_{t}^{T} \gamma^{t} r_{t} \qquad (3)$$

   where r_t is the reward from each step *t* of the previous episode. The problem with Monte Carlo estimators happens when the policy is stochastic. With stochastic policy, there will be some noise associated with the estimated returns, since with the same state, an agent can produce different actions. This stochasticity leads to high variance in the rewards as well. This variance is typically mitigated by using a large number of action trajectories, with the hope that the variance introduced in any of the trajectory will be reduced in aggregate, thus providing an estimate of the "true" reward structure of the environment. Another disadvantage from Monte Carlo estimator is it does not allow the agent to do online learning. Online learning supports an agent to learn while collecting experience data and this will hasten the learning process. But with Monte Carlo estimate, the agent need to have complete

information from one episode, and that means learning can only happen after completing one episode.

2. Temporal Difference Estimate

The other way to estimate the returns is by using Temporal Difference (TD) estimator. In TD estimator, a state value function V is used to estimate the return of the current episode.

$$R(s,a) = r_t + \gamma V(s_{t+1}) \quad (4)$$

Future rewards in TD estimates are updated by using state value function. This is called bootstrapping which is shown in Figure 2.2. After predicting the estimated return for the next step, it will compare the estimate return value with the estimated value function from the current state, to compute TD-error (δ) as shown in equation (5). The estimator will use TD-error to improve its ability to generate more accurate state value function on the next episode. The TD error and update of TD estimator are denote by:

$$\text{TD-error } \delta = r + \gamma V(s) - V(s) \quad (5)$$

$$V(s) = V(s) + \alpha\,\delta \quad (6)$$

where$\alpha$is the update step size specified by the user. By relying on value function estimators, the expected returns tend to be stable over time since the estimator generate the expected returns with low variance and the agent will learn in a stable manner for the whole process. Mostly, neural networks [8] are used as value function estimators. They provide high flexibility in predicting the state value function, but most neural networks are very noisy and not able to reflect the real condition of reward, especially in the beginning of the learning. The advantage of using TD is that it allows agents to do online learning, since it uses value function estimates and does not depend on rewards from previous episodes.

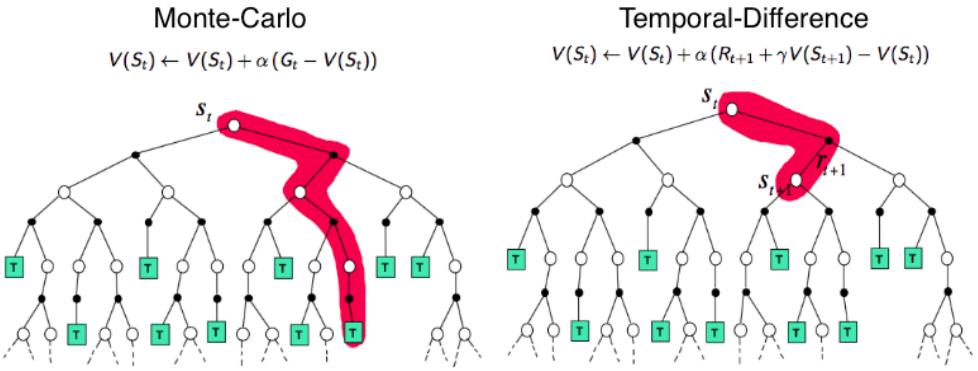

Figure 2.2. Illustration of Monte Carlo Estimate (left) and Temporal Difference (right)

2.1.2 Advantage Function

One way to balance between bias and variance in reinforcement learning is by using the advantage function. It combines the principle of Monte Carlo estimate and TD estimate by using both of them in the formula. The Advantage function is the function to compute the difference between the estimated return and the value function of the current step. Formally, the Advantage function can be denote as follows:

$$A(s_t, a_t) = R(a_t, s_t) - V^{\pi}(s_t) \tag{7}$$

where R can be obtained from equation (3) and (4), and V is value generated from value function. This Advantage function is related directly to how much better the agent actually performed than was expected on average, thus allowing for intuitively interpretable values. From Monte Carlo point of view, it will add more bias by using the value from the estimator. For TD Learning, it stabilizes the value function by not using the estimated returns directly. It will produce smaller value and stabilize the learning by keeping agents from not moving too much from the previous value.

2.1.3 General Advantage Estimate (GAE)

Although Advantage function might address the bias and variance issues, it still cannot control the amount of bias and variance and only minimize them in general. In order to control the bias and variance, Schulman et al proposed a solution by using

the General Advantage Estimate (GAE) [9] function. Unlike normal Advantage functions, by using GAE, it provides flexibility to choose between Monte Carlo estimate and TD estimate, or even a value between them. Besides using γ to discount future rewards, it uses lambda to control the variance and how accurate the function would be. The GAE(γ, λ) function is expressed as below:

$$gae_t = \delta + \gamma\lambda * gae_{t+1} \quad (8)$$

$$R(s_t, a_t) = gae_t + V(s_t) \quad (9)$$

GAE is computed in reversed direction, starting from the last step in an episode. From the expression above, the GAE shows the flexibility to choose between Monte Carlo and TD estimates. If λ is set to be 0, it will be similar to TD and has more bias. If λ is set to 1, all rewards from future steps are considered, and it will become Monte Carlo with high variance. To balance the variance with more stable learning, usually lambda are chosen to be 0.95.

2.1.4 Policy Gradient

In a policy based method, it uses the gradient method to improve from previous policy to the next policy. It is called policy gradient [10]. The policy can be denote as a function of Θ. The goal is to maximize estimated returns for following the policy, and using that value to improve the policy. It is an optimization problem where the objective function is denoted as:

$$J(\theta) = E[\sum_{t}^{T} R(s_t, a_t)|\pi_\theta] \quad (10)$$

This policy based method often uses neural networks as its architecture. By using this objective function, it will take its gradient and use it for back-propagation [8] to update the policy. By substituting the return with Q value for following policy πinto equation (7), and with simplification, the gradient can be expressed as:

$$\nabla_\theta J(\theta) = E[\nabla_\theta log\pi_\theta(s, a)Q^\pi(s, a)] \quad (11)$$

Parameters are updated using gradient ascent, in order to achieve an optimal policy. There are two kinds of policy gradient methods, off-policy learning and on-policy learning.

A. On-Policy Learning

On-Policy learning algorithms are the algorithms that evaluate and improve the same policy which is being used to select actions. That means we will try to evaluate and improve the same policy that the agent uses to select actions. Example of on-policy algorithm is Proximal Policy Optimization (PPO) which will be discussed more in this project in chapter 2.2.3.

B. Off-Policy Learning

Off-Policy learning algorithms evaluate and improve a policy that is different from a Policy that is used for action selection. The easiest way to distinguish between off-policy and on-policy is by looking at the number of policies. In an off-policy algorithm, there are usually two policy networks, a main network and a target network. The Agent will use the main network as behavioural policy to choose an action, while it will use target network to train the policy. Another way to view off-policy is when agents will use a replay buffer to store previous state and action pairs. It will update the policy using state-action pair from the replay buffer that is not generated by the target network, while the update is happening toward the target network. Examples of off-policy learning is Deep Q Network (DQN) [11] and Deep Deterministic Policy Gradient (DDPG) [12] where each learns from the replay buffer that was generated using the previous policy, and not by the current target policy.

2.1.5 Reward Function

The most important part in reinforcement learning is how to design the reward function for a specific problem. In designing an optimum reward function, it needs to account for the possibility of the agent in acquiring a local optimum solution instead of our desired solution. For example, in designing a reward function for a cleaning

machine, a +1 reward is given for turning on the machine, while +100 is given for cleaning the floor. The reward function intended to make the machine turn on one time and then clean the floor. Instead, the agent may turn on and off for over and over again because it will get reward +1 for just doing that and not moving at all. The principle of estimated returns from Monte Carlo estimate and TD learning is very needed to construct an optimum reward function. It is very tricky especially when a high bias value estimator is used, such as a neural network.

In the implementation, environments can either give positive or negative rewards. Negative reward is usually used by goal environment when agents need to achieve the goal as fast as possible. This is to make the agent account for each action as bad action and try to solve the problem sooner to avoid larger penalties. Meanwhile positive rewards are given to agents to quantify the current state of the agent. Positive rewards encourage agents to carry out an action with higher rewards in every step and collect as many rewards as it can. Both rewards can be combined, but the behaviour may not be similar to the both scenarios above if combined.

In some implementations, agents are given similar rewards in each step as long as the agents have not reached their goal state. This reward function is called sparse reward. This kind of reward is given during a game of chess, +1 for winning the game, -1 for losing the game, and 0 for other possibilities. Giving sparse rewards can be effective in the case of small state and action space. It will give objective value because the agent only cares about winning and completing the task. But in a large state and action space, it will take a lot of time because there are a lot of state-action possibilities that an agent needs to explore and it does not give agent rewards and any knowledge until the agent reaches the goal condition.

2.2 Deep Reinforcement Learning Algorithm

Deep Reinforcement Learning (DRL) is a reinforcement learning implementation that uses and maximizes the neural network for its functionality. There are a lot of DRL algorithms, but only 3 algorithms will be discussed in this project. Different

kinds of algorithms can only work on certain conditions. For example DQN [11] only works on discrete state and action space. On the other hand, DDPG which is an extension of DQN only works on continuous action space. Some Algorithms are available in Continuous and discrete action spaces like PPO.

2.2.1 Actor Critic Algorithm

Actor critic [13] methods are reinforcement learning methods that maximize two major structures in RL, policy and state value function. Actor is a Policy network that determines agent action while Critic is a value function estimator that generates estimated return for every action taken in every step. In Deep Reinforcement Learning context, actor and critic networks are implemented by using neural networks.

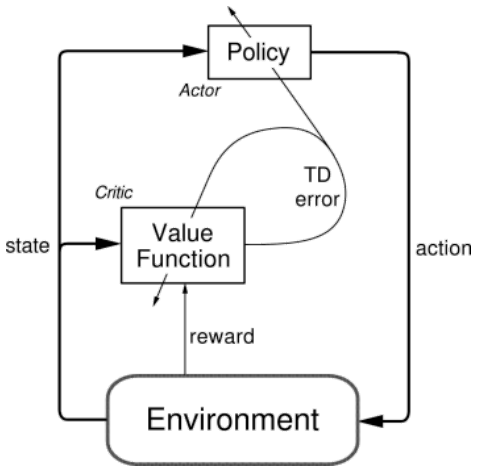

Figure 2.3. Actor Critic Illustration

By using the state information, the actor network will generate action based on the policy. Rewards given from the environment will be used to update the critic network that generates the Q value, by taking the difference between reward and Q value and doing back-propagation with that difference. This will improve the ability of the critic to generate more accurate value functions. Subsequently, the critic will send the value to actor network, and actor will use the TD error from this value to perform policy gradient procedure and improve its policy. The agent will repeat this process for every step until it achieves the optimal policy as shown in Figure 2.3.

2.2.2 Deep Deterministic Policy Gradient (DDPG)

Deep Deterministic Policy Gradient (DDPG) is a reinforcement learning algorithm, based on actor critic methods too. Inspired by the success of implementing neural networks and replay buffers in DQN [11], Lilicrap et al. adopted the same method to implement neural networks into Deterministic Policy Gradient [14]. The overall process is similar to the Actor Critic algorithm, except that the actor-critic learns from randomly picked data from the replay buffer experience, and not just from current state and action. Other than that, there will be two pairs of actor-critic networks. One network acts as the main network, and the other network is the target network for learning purposes. The main network is responsible for the main policy function and Q value function to determine action and Q value for certain states, while the function of the target network is to train and maximize the main network. During training, the main network will freeze and not perform policy gradients. After the training is finished, the target network will update the main network using a soft update that transfers only some percentages of the parameter of the trained network and maintains some percentages from the parameter of the main network. This is to prevent the main network from changing too much and make it more stable during learning. The pseudocode of DDPG is shown in appendix A.

DDPG only works on continuous state-action space. It is an off-policy algorithm since the agent learns using state-action pairs from a replay buffer that contains the state-action pairs that are taken from the previous behavioural policy, different from the current policy. Other than that, to improve exploration during the training process, DDPG uses Ornstein-Uhlenbeck process [15] as the noises, as suggested by Lilicrap et al in the original paper [12].

2.2.3 Proximal Policy Optimization

In policy optimization problems where policy gradient method is used, it succeeds in producing the optimal policy by taking gradients of the step. Using estimated returns as an objective function, it manages to converge to the global optima in most of the problems. However in some problems, gradient free methods such as Covariance Matrix Adaptation (CMA) [16] are chosen as they outperform the policy gradient

method. In order to improve the performance of the policy gradient method, Schulman et al. proposed a method to limit the gradient step, by using the Trust Region Policy Optimization (TRPO) Algorithm [17] where this will have higher probability to converge and outperform gradient free method. However, TRPO uses second derivatives to compute the step region, which takes a lot of time during its calculation. To improve the performance of the algorithm, Schulman et al. proposed another solution to extend his previous method using Proximal Policy Optimization [18]. PPO emerged as a more efficient solution, by changing the loss function from second derivatives term with clip surrogating, in which it becomes computationally efficient and does not include any derivatives. PPO prevents the agent from moving too far from the initial policy. The objective function in PPO is denoted by:

$$L = min(r(\theta)\, A(s,a),\ clip(r(\theta),\ 1-\epsilon, 1+\epsilon)A(s,a)) \qquad (12)$$

$$\text{with } r(\theta) = \frac{\pi_{\theta}(s,a)}{\pi_{\theta_{old}}(s,a)} \qquad (13)$$

where r is the ratio of current policy to the previous policy. By clipping this ratio, it will prevent the current policy from being very different from the previous policy.

As shown in Figure 2.4, if the advantage function is positive, the agent will move with a step size of either the ratio or 1+ϵ whichever is the lower. With this, the agent will increase the probability of given action, but not to make it very probable that other actions cannot be taken. In contrast, if the advantage function is negative, the agent will decrease the probability of action but not to make it impossible to be taken by keeping the step size to be the highest of ration and 1-ϵ. As the step taken is not too big, it allows the agents to undo the learning process in case there are miscalculations during the learning processes.

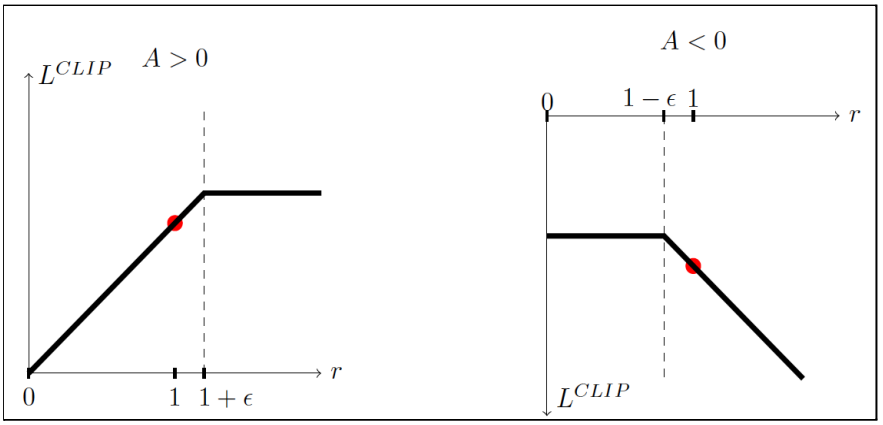

Figure 2.4. Estimated step size by doing clipping. It will be constant after a specific value

PPO is applicable for both discrete and continuous action spaces. PPO uses stochastic policy to generate some randomness in choosing action. For discrete action space, PPO generates the probability of each action and later the agent will pick an action randomly with respect to its probability. For continuous action space, PPO uses Normal distribution to pick the action. After the policy generate the mean and the variance, PPO agents will sample from Normal distribution using this mean and variance value.

2.3 Hindsight Experience Replay

In some goal environments which agents need to achieve the goal in the fastest possible time, usually sparse rewards are used to make the agent consider each reward as a bad reward and it will then try to complete the task as fast as possible to avoid accumulating more negative rewards. However as mentioned in the chapter 2.1.5, this will slow down the agent in learning because the rewards do not have any information regarding the agent state. In large state action space, this will be very costly since the agent will have a lot of things to explore.

Hindsight experience replay (HER) [19] is able to solve this problem. It is a method to extend the replay buffer functionality in off-policy algorithms like DDPG. After performing an action at some state, it will arrive at the next state and by right, DDPG will record all that information in the replay buffer. With HER, other than just storing

this information, it will generate 'fake' data, that assume this new state is the goal for this problem, and give rewards for this action and space. It will give agent information about how to achieve this new state, and improve exploration strategy since agents get more context of the environment and are not required to blindly explore the environment. It cannot be implemented in on-policy algorithms because by using additional data from the buffer, it will change the behavioural policy of the agent. The pseudocode of HER is shown in Appendix B.

2.4 Attention

In a typical sequential problem such as Natural Language processing (NLP), the order of the data affects how the network processes them. It is because to process the data, the network uses the information from its previous data, with the most recent one being the most significant that affects the network. Recurrent Neural Network (RNN) [20] is used to manage this situation. However, in some cases, the most significant data is not necessarily the most recent one, and it might be separated far away from the data. Hence there is a need to process the data with a new method that is invariant to the order or permutation. One example of a permutation invariant model is attention.

Vashwani et al [21] manage to find correlation between each element in the data without their order information, and make it invariant to permutation. By encoding the data into numerical values, it uses this encoded value to get correlation information with other elements. The attention model uses three linear neural network layer, Key K, Query Q, and value V to process the encoded data. Overall the operation and the output can be denote as:

$$Attention(K, Q, V) = Softmax(\frac{KQ^T}{\sqrt{d_k}})V \quad (14)$$

$$y = x * Attention(K, Q, V) \quad (15)$$

where K, Q, and V are parameters of the network in matrix form, d_k is the scaling factor, y is the correlation output which is invariant to permutation, and x is the encoded input. The illustration of attention is shown in Figure 2.5.

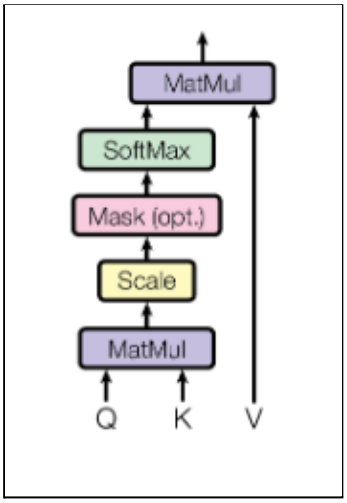

Figure 2.5. Attention Mechanism [21]

This attention mechanism will generate correlation coefficients of all elements to a particular specific element. This operation will be repeated for every input in a sequence until the information about all correlations are retrieved. With this correlation information, it will give more context from this correlation information, while making the order of the input no longer important.

Chapter 3

Methodology

In this project, we will implement reinforcement learning algorithms for several autonomous vehicle simulations. We will focus on two algorithms, DDPG and PPO, to be used to solve the vehicle problems. Both DDPG and PPO are based on actor critic method that uses two neural network structures for their functionalities, which will be discussed later in this report. Before implementing our reinforcement learning agent into our main problem, we first apply DDPG and PPO to previous well known problems and compare the results with previous experimental results.

3.1 Environment Setup

Before conducting the experiment, the environment is set up first. The following tools are used during this experiment:

- Python Programming Language

  Python is used as the main programming language. Python is chosen because it has a lot of support for computational process and visualization. It also provides a lot of open source support for neural networks and reinforcement learning.

- Pytorch

  Pytorch is an open source neural network framework developed by Facebook AI Research (FAIR) lab. Pytorch is chosen as the main neural network framework as it supports all neural network functionalities needed in this project. Other than that, Pytorch has a better fit with another library that will be used later in this project.

- OpenAI Gym

  OpenAI Gym [22] is a RL toolkit created by OpenAI. Gym allows users to run reinforcement learning algorithms directly in the gym environment and provides ease of rendering the results of the algorithms. Gym only runs in Python and until now, there are no other implementations in other programming languages. Gym also provides built-in reinforcement learning

agent and pre-trained RL agent to see the correct result. Gym has a number of RL environments such as Atari games, Robotics Environment, Mujoco, and Lunar Lander. Besides having its own environments, gym can be used to create another custom RL environment. Most of the RL environments are built based on gym implementation.

- Stable Baselines3 Library

  Stable baselines 3 [23] is a library in Python which provides a number of reinforcement learning algorithms and can be implemented directly to the gym environment. It uses Pytorch as the neural network framework and is a Pytorch implementation of stable baselines which use tensorflow. It allows users to customize the neural network structure for the algorithm and integrate with their reinforcement learning implementations. Stable baseline will be used as the main library to implement reinforcement learning algorithms with customized neural networks and parameters.

- PyBulletGame Environment

  OpenAI Gym is the most popular RL environment today. Unfortunately, it needs Mujoco engine support for several challenging control tasks. Currently, Mujoco requires paid license and does not have the flexibility to change hardware between experiments. In order to resolve any Mujoco dependency, PyBulletGym [24] runs on top of PyBullet physics engine. PyBulletGym have several control environments such as HalfCheetah and InvertedPendulum.

3.2 Network Architecture

Neural networks are used in both algorithms. A 4 network architecture is used for DDPG. Two networks as actor network and critic for the main network, and 2 networks as target actor and target critic for the target network. Both networks are initialized the same way in the beginning. We use a similar architecture as used by Lilicrap et al [12], which consists of a network with 2 hidden layers and a ReLU activation function for both actor and critic. For actors, we use tanh activation function at the output layer to bound the result to a certain value. Action will be

injected by OUNoise during the training process, for exploration purposes. For critic value function, it has one additional layer to process action, and one layer to add the value from the processed state layer and processed action layer, before computing the Q function. Weight and bias for each neuron are initialized randomly from a uniform random distribution. Overall the architecture is denoted as in Figure 3.1.

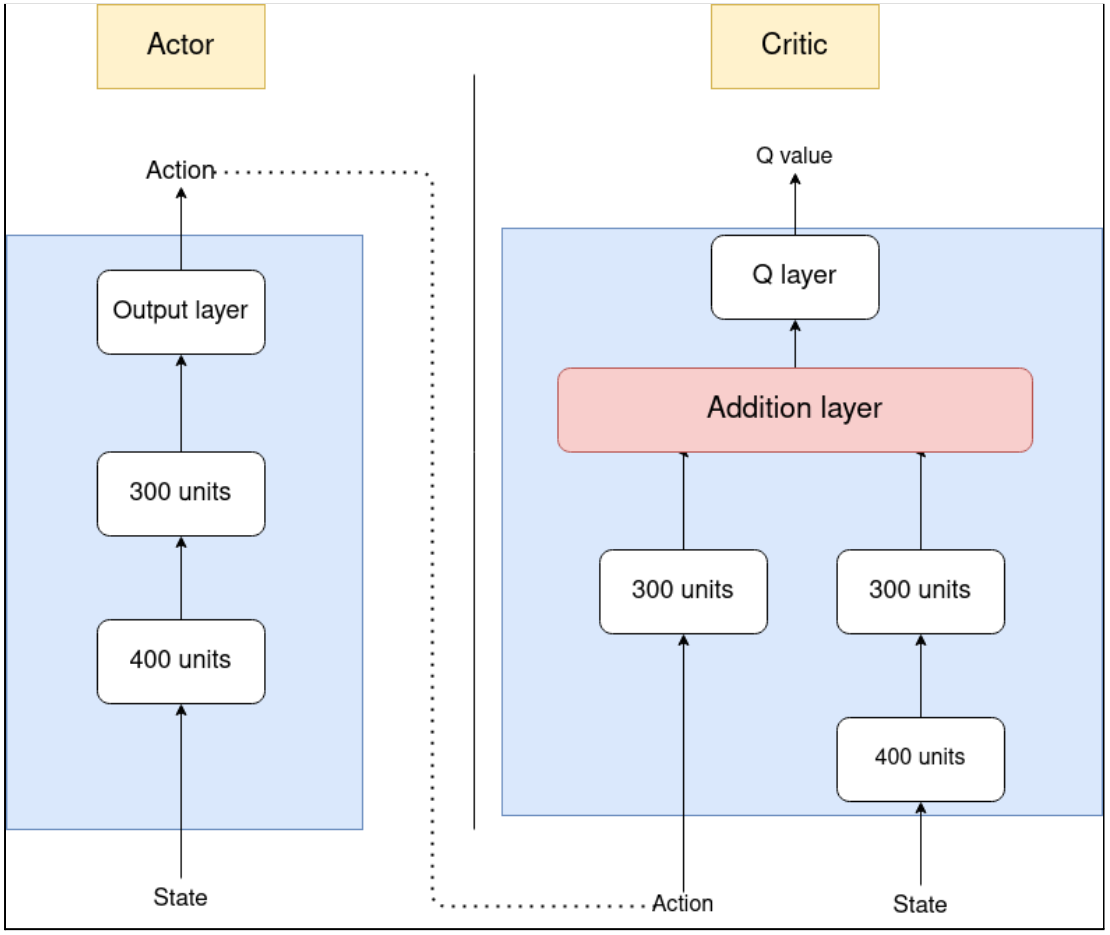

Figure 3.1. DDPG Architecture

For PPO, the actor network uses the same architecture as the actor network in DDPG, except for discrete action space, the output layer will use softmax activation function to output the probability of each action. For Critic network, it uses the same architecture as the critic from DDPG, except there are no layer for the action and addition layer since in PPO, only state value function V needs to be computed. There are no noises injected into PPO since PPO uses stochastic policy which already generates randomness.

Adam Optimizer [25] is used as the optimizer for neural networks on both algorithms, to perform gradient descent/ascent and backpropagation from the loss function for each algorithm.

3.3 Hyperparameter

During the experiment, choosing hyperparameter is a crucial part in balancing between exploration and exploitation. With a low learning rate, the agent will take a long time to learn, but with a high learning rate, the gradient update will be too large and may avoid reaching global or even local optima. In this experiment, parameters are determined from the value in the previous paper. The hyperparameter of DDPG is shown in Table 3.1.

Table 3.1. Hyperparameters of DDPG

Hyperparameter	Description	Value
Batch size	Number of sample actions agent will pick from replay buffer	100
Alpha (α)	Learning rate for actor network	0.0001
Beta (β)	Learning rate for critic network	0.0001
Tau (τ)	Percentages of target network that will be transferred to the main network for '*soft update*'	0.001
Gamma (γ)	Discounting factor	0.99

The learning rate is kept small so that the gradient will not explode during backpropagation on the neural network. The Discount factor is set to be 0.99 since most of our experiments are over 100 steps, so that we can consider future rewards, but need not consider all rewards until the last step in an episode. Tau is set to be small, so that agents can perform soft update for action and critic and not move too far from previous policy during updates.

Table 3.2. Hyperparameters of PPO

Hyperparameter	Description	value
Beta (β)	Learning rate	0.0001
Momentum (p)	Momentum for neural network optimizer	0.9, 0.999
Gamma (γ)	Discounting factor	0.99
Epsilon (ϵ)	Clip parameter for clip surrogate advantage function	0.2
Epoch	Number of times the data will pass to the algorithm	10
Lambda(ƛ)	Parameter for GAE	0.95

From Table 3.2, the learning rate and discount factor are kept small for the same reason as DDPG. Epsilon, which is used for clipping, is set to be 0.2 so the ratio will be kept within the [0.8, 1.2] interval and the policy will be able to learn effectively. Even though Schulman et al. uses advantage function [18], in this implementation, we will use GAE lambda instead of advantage function to balance the bias and variance in the experiments to be discussed later

3.4 Gym Environment

Some environments are chosen for our RL algorithms implementation. Lunar Lander is chosen from classic environments as the initial experiment to compare with the

benchmark value. Lunar Lander is a problem of landing a spacecraft on the uneven surface of the moon. Agents get more reward if the spacecraft landed smoothly and will get negative rewards if accidents happen to the spacecraft. The agent has 1000 steps in one episode to solve the problem.

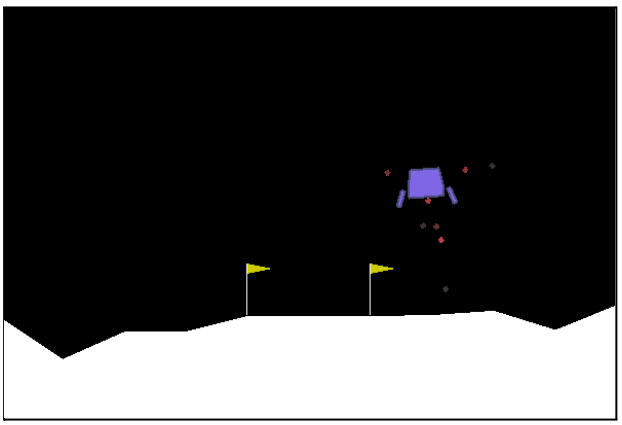

Figure 3.2. Lunar Lander Problem

The spacecraft is floating initially and while spacecraft will automatically go down because of gravity, the agent needs to control the speed and direction to make the spacecraft land on the right spot smoothly. The state representation of the lunar lander is defined by an 8x1 array. The complete state representation is shown in Table 3.3 below:

Table 3.3. State representation of Lunar Lander

Index	Information
1 and 2	Spacecraft position of x and y coordinates with respect to the screen
3 and 4	Velocity in x and y directions
5	Angle of spacecraft with respect to a vertical line
6	Angular velocity of spacecraft
7 and 8	Conditions of left and right landing gears of spacecraft

The episode ends with success when the spacecraft lands both of its landing gears on the ground. It will be considered as a failure if the spacecraft go out of the screen or break one of its landing gears. If the agent reaches 1000 steps, the simulation will be terminated automatically, and the agent will need to start from the beginning again.

There are 2 versions of Lunar Lander problem: discrete action and continuous action. Both PPO and DDPG will be used to solve the Lunar Lander problem. PPO will be used to control discrete action agents. There are 3 possible discrete actions: 0 for move up, 1 to move towards left side, and 2 to move towards right side. For continuous action space, both PPO and DDPG will be used. The action space consists of 1x2 matrix, where the first element denotes the angle that the spacecraft needs to turn to and the second element denotes the acceleration of the spacecraft.

The experiment for each algorithm is conducted for the agent to at least reach 2000 episodes in total, with each experiment running 3 times. This is to test the overall performance by the model. By running the experiment one time only, it might have a lot of bias in that experiment, and the bias might change for the next experiment. By running the experiment multiple times, we can get the overall bias and see the overall performance of the agent.

3.5 PyBullet Environment

To cross validate the algorithms, other environments will be tested and implemented with PPO and DDPG algorithms. To extend the problem to be more challenging and complex, PyBulletGym environment is used as the next experiment. Most complex control problems are related to robotics that use graphical representation and physics engines. PyBulletGym is an open source library for reinforcement learning which uses a physics engine based on PyBullet. Walker2D and HalfCheetah from PyBulletGym are chosen as the next two environments. Walker2d is a problem where an agent needs to teach a robot leg to walk. Rewards are given based on its distance and balance. HalfCheetah is a simulation of a cheetah robot. The goal of this environment is to teach the cheetah robot to run with controlling its leg joints.

Similar to Walker2d, rewards are given to the cheetah based on distance travelled and the efficiency of the motion. Both experiment terminates if the agent fall and fail to maintain it balance, or after the agent has completed 1000 steps.

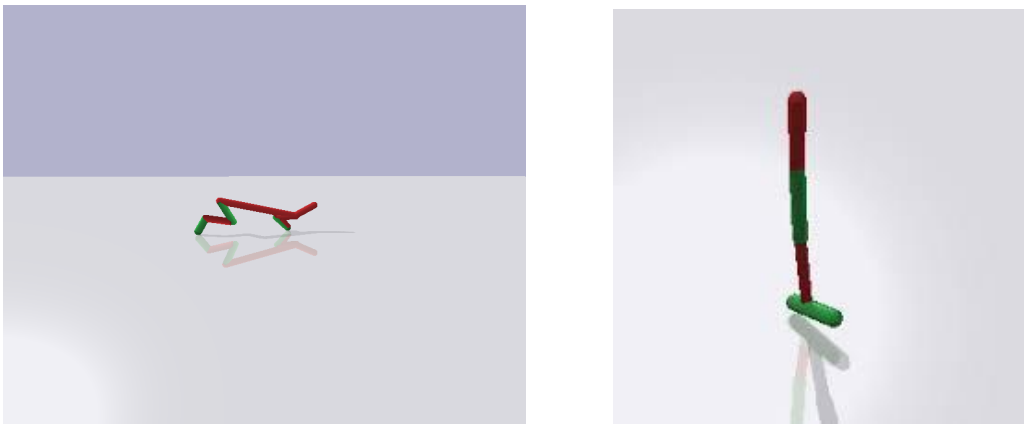

Figure 3.3. Pybullet Environment, HalfCheetah (left) and Walker2d (right)

Both environments have more complex action and state spaces, which increase the difficulty levels of the environments since each has more options to choose from and more combinations of state action pairs. This will affect the convergence of state value function as some states and actions might not be examined or only examined a few times.

Both problems have continuous action space, and both DDPG and PPO algorithms will be used in this PyBulletGym experiment. The experiment will be conducted in the same manner as the Lunar Lander experiment, by conducting the experiment 3 times in each environment. The experiment on PyBulletGym is conducted within at least 3000 episodes, longer than Lunar Lander, since it is a relatively more complex problem than Lunar Lander.

3.6 Highway Environment

As the main environment to investigate in this project, an open source highway environment simulator [27] is used to simulate the autonomous driving car control experience. Based on Pygame, it provides several different simulations of

autonomous car scenarios, such as driving on highway roads, parking simulation, going through intersections, and even roundabouts.

In this experiment, two simulations are used for investigation: autonomous driving highway simulation and parking simulation. The terms ego car in these experiments refer to the car that we control.

3.6.1 Highway Simulation

Highway simulation is a problem to control one or multiple cars moving on highways. The car needs to avoid collision and maintain medium to high speed in order to get positive rewards. The episode will terminate if the car collides with another car or if the car finishes 100 steps. In the highway driving simulation, the state can be adjusted by providing the information about *n* closest cars to the ego car. With this assumption, the controlled car is expected to know all complete information regarding other cars in terms of positions, velocities, and cruising directions. Figure 3.4 shows an illustration of the highway simulation. Our car is denoted by green color and it will try to move forward towards the right in this illustration.

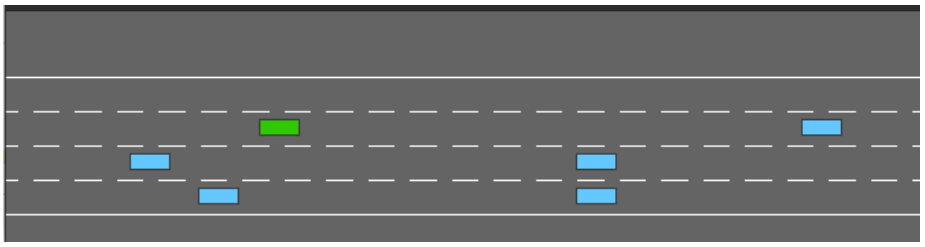

Figure 3.4. Highway Driving Simulation

The state is defined by an *n*x7 array. The first 1x6 items in the array represents the condition of the ego car, such as the position and velocity of the ego car in x and y directions. The rest of the (*n*-1)x7 items in the array represents information about the (*n*-1) other cars. These show the distance between each of the (n-1) cars to the ego car in the x and y directions, and also the difference in speed between the ego car and each corresponding car. The information of (*n*-1) other cars will be displayed in a sorted manner from the nearest to the farthest with respect to the ego car. So in case

the distance changes in the environment, the order of cars in the state representation will also change. For example in Figure 3.5, the second row shows the information from the closest car from the ego car. x and y values denote the distance of the closest car from the ego car which located 25 meter ahead of the ego car in x coordinate, and 0 in y coordinate which indicate they are in the same lane. Vx and vy values also denote the difference between the velocity of the closest car with the ego car, which means the closest car velocity is 25-5=20 m/s in x direction and 0 in y direction.

Presence	X	y	vx	vy	cos_h	sin_h
1	216.423423751556	4	25	0	1	0
1	25.5641381104822	0	-5.197606327613	0	1	0
1	47.4376549169358	-4	-7.84458852751	0	1	0
1	68.0306555798447	8	-10.23060943396	0	1	0
1	92.0573621380198	8	-5.146948475423	0	1	0
1	112.376975676327	8	-9.790507829711	0	1	0
1	133.21633840893	-4	-6.125804053873	0	1	0
1	156.724751974756	8	-7.707534575397	0	1	0
1	177.981550521933	8	-10.30545863126	0	1	0
0	0	0	0	0	0	0

Figure 3.5. Example of State Representation of Highway Simulation

For the action space of the highway simulation, the action is represented in discrete representation from 0 to 4. The action representation is denote by Table 3.4:

Table 3.4. Action space in highway simulation

Action value	Action
0	Turn Left
1	Idle, maintain velocity
2	Turn Right
3	Accelerate
4	Decelerate

In this experiment, we will use the PPO algorithm because the action space is discrete. In addition to PPO, we will add attention layer on the top of PPO, which will be discussed later in this report. We will test the effectiveness and performance of the attention layer. We will also compare the results with standard PPO implementation without attention layer. All experiments will be conducted 3 times, with each of them conducted within 2000 episodes.

3.6.2 Parking Simulation

In parking simulation, an agent need to control the ego car which is denoted by yellow car to park in a specific target place which is denoted by a blue square shown in Figure 3.6. All experiments begin with a car located in the center of a parking lot as pictured in Figure 3.6. The agent needs to park the car with correct alignment against the boundary lines of the blue square within 100 steps. Otherwise, the episode terminates and it will be considered a failure. Parking simulators, on the other hand, have continuous action space. It consists of a 1x2 array where the first element shows acceleration and the second element shows the steering angle with respect to the car alignment.

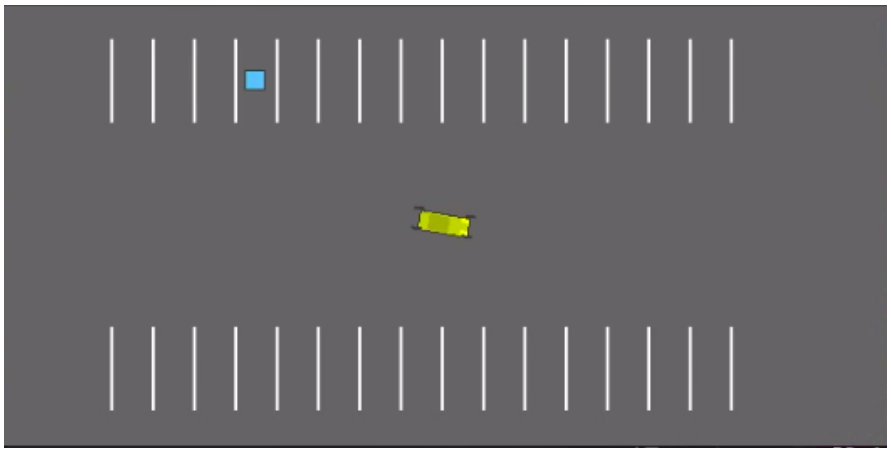

Figure 3.6. Parking Simulation

The state in the parking simulation is represented by a dictionary like in other goal environments in the gym environment. It consists of observation, achieved goal which denotes the current condition or coordinate of the agent, and desired goal which denotes the real objective of the agent. In this case, the observation and

achieved goal are represented by the current position and velocity of the car in the x and y directions, and cos and sin values of the angle between the car and the vertical line. The desired goal represents the specific condition that the agent needs to achieve, like the x and y coordinates of the desired parking space, velocity that is needed in the parking space, which should be 0, and the ideal angle of the car in the parking space.

In this experiment, since it has continuous action space, we will use both DDPG and PPO. In addition, we will add one more experiment by adding DDPG that combines with HER, to examine the effect of using HER to improve agent performance in goal environment situations like the previous experiment conducted by Andrychowicz et al [19]. All experiments will be conducted 3 times with each experiment to be completed within 2000 episodes like the highway experiment.

3.6.3 Reward Design

Different reward formulations are used for each highway and parking simulation. For the parking simulation, negative reward is used so that the agent will try to complete as fast as possible like other goal environment problems. The reward is given in each step by computing the absolute difference between each element in the achieved goal and desired goal. Finally, by taking the negative sum of this value, it will be compared to some scalar benchmark. If it is higher than the benchmark and closer to 0, it means the car is close enough to the target and the episode is terminated as a success. Otherwise, it will keep finding the goal space until 100 steps have been reached, when the episode is then denoted as failure.

For the highway simulation, positive reward functions are used. It divides the reward into 3 parts. The first part rates the speed of the agent. The velocity will be scaled into a fraction within a given interval of min velocity a and max velocity b. The velocity v is rewarded as:

$$scaled\ speed\ = clip(\frac{v-a}{b-a}, 0, 1) \quad (16)$$

The speed reward is clipped between 0 and 1 to maintain a positive value. Then to keep the ego car to the right lane, reward is given based on the lane the agent is located in. The lane reward is given by:

$$lane\ reward\ =\ \frac{current\ lane}{total\ lane} \tag{17}$$

Finally, a penalty is given to the agent if it collides with another car. Penalty is set to be -1 for every collision, and 0 if there is no collision. Some weight is given to the speed reward and lane reward, to let the agent know that it is better to maintain the speed instead of moving slow on the right lane. Combining equation (16) and (17), overall the reward function is given by:

$$r\ =\ crashed\ +\ 0.4\ *\ scaled\ speed\ +\ 0.1\ *\ lane\ reward \tag{18}$$

In the end, the reward is normalized to keep it positive. This is because negative rewards will probably make the agent end the episode as soon as possible by seeking for a collision. If we use negative rewards in this problem, the agent might predict some unexplored states has some negative rewards. As such, rather than exploring other states, the agent will choose to collide since this gives a return that may be better than exploring another action or state.

3.7 Attention Layer Architecture

Attention Layer in this experiment will be used to make the data to be invariant to their order. This will be used only in a highway environment in highway simulation, because the state space in that problem is much correlated with the order. The state representation in highway simulation is represented in a sorted manner, from closest to farthest with respect to the ego car. Consider the scenario in Figure 3.7, where the closest car in the left picture of the figure is the one above the ego car. By right, the second row in the state representation like in Figure 3.5 should display the information about the car above the ego car. However as the car is moving forward as shown in the right picture of Figure 3.7, the controlled ego car now is closer to the car below, and the second row of the state representation now should display the information of the car below. With this sudden change, the agent may still think that the second row still belongs to the car above, which makes confusion for the agent.

With Attention Layer, the ego car will be able to recognize the nearest car at any point in time since the second row of state representation like Figure 3.5 is always updated to reflect the nearest car.

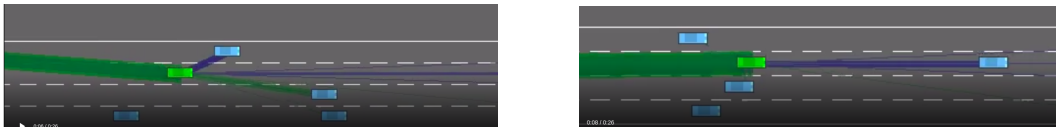

Figure 3.7. Change State Scenario from left to right

As recommended by Eleurent and Mercat [28], [29] attention layer will be used to compile the input state before going into RL agent. So later in the experiment, a PPO agent will only use states that are already invariant to their order. The overall Architecture with attention layer is shown in figure 3.8.

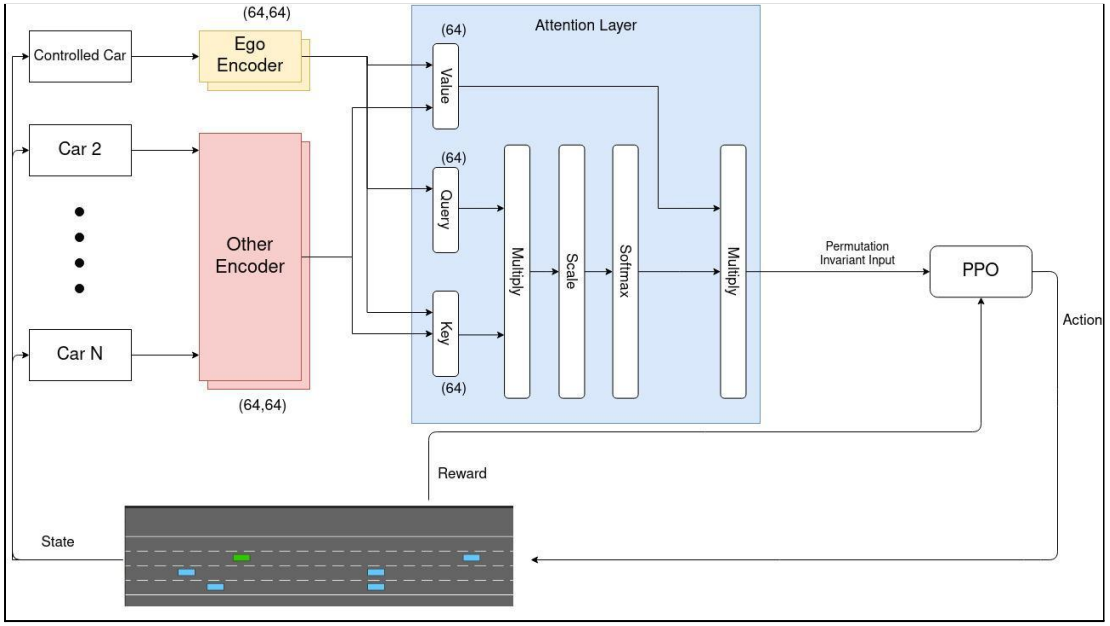

Figure 3.8. Project Architecture

Before applying the states into the attention layer, the states will be encoded by an encoder layer which uses a neural network. There will be two encoders, one to encode the information about the ego/controlled car which is located as the first row of the state representation, and the other encoder to encode the rest of the cars. This is to make the learning consistent since the state location for the ego car will never change i.e. always located in the first row. Both Encoders consist of two layers of

neural networks, with each layer containing 64 neurons. After the input is encoded, it will go to the attention layer.

The attention model will be using a one-head architecture which denotes scaled dot-product attention by Vaswani et al. [21]. We use a one-layer network with 64 units at the key, value, and query layers. The input from the ego encoder will go into all key, value and query layers, while the other encoder will go into key and value layers. The query uses only input from ego encoders, because we only want to get correlation from the ego car.

With using attention, we hope that our autonomous car can identify the other car's identity, and not just knowing the information about other car positions and velocities. Knowing the identity of another vehicle, our autonomous car can then estimate the movement from other cars and improve its perception about the surrounding.

3.8 HER Architecture

In the parking environment, we will implement the HER algorithm together with DDPG as the off-policy algorithm. Since the parking environment is one of the goal environments, the reward is negative, and it is desired for reward to tend to 0.

During the experiment, the DDPG agent will collect the data from experience as in standard DDPG. After storing the information such as state, action, next state, and reward information to the replay buffer, HER will add additional data to the replay buffer, that contain values that were inserted before, with reward being 0 as the goal to achieve.

Chapter 4

Results and Discussion

4.1 Experiment on Lunar Lander

Experiments are conducted on Lunar Lander problems on both continuous and discrete action space. The result will display the average reward per episode, which comes from averaging the rewards of the same episode from the 3 experiments for the same algorithm. The plot will be shaded to indicate the standard deviation of each episode's reward from the 3 experiments to give more information regarding the behaviour of the agents. This is to monitor the consistency of the agent's performance. Large shade indicates the variance of the agent is high, and the agent achieved very different results in those 3 experiments, and thus the agent's performance is varied in each experiment. Small shades indicate low variance from the result and demonstrate the behaviour of agents that perform relatively similar in all 3 experiments.

4.1.1 Experiment on Discrete Lunar Lander

In this experiment, we will display the result for PPO algorithm as this experiment only uses PPO algorithm, We will compare the previous result by Gadgil et al [30] which conducted the Lunar Lander Experiment with another algorithm. The maximum value achieved by [30] is around 200, which is the highest possible reward in one episode for LunarLander, and this indicates that the experiment is successful. The statistics from the experiment is shown in Table 4.1

Table 4.1. Discrete Lunar Lander Experiment Statistics

Algorithm	**Number of steps per Episode**
PPO	414.155419

From Figure 4.1, we observe that the agent achieves similar result with the experiment conducted by Gadgil et al. [30]. We observe the agent's performance

increases significantly before converging into a positive reward area. This reflects that the agent is able to achieve positive rewards and solve the task by landing the spacecraft in most episodes. With an average number of steps to be 414, the agent is able to complete this task with less number of steps compared to maximum value allowed in this problem.

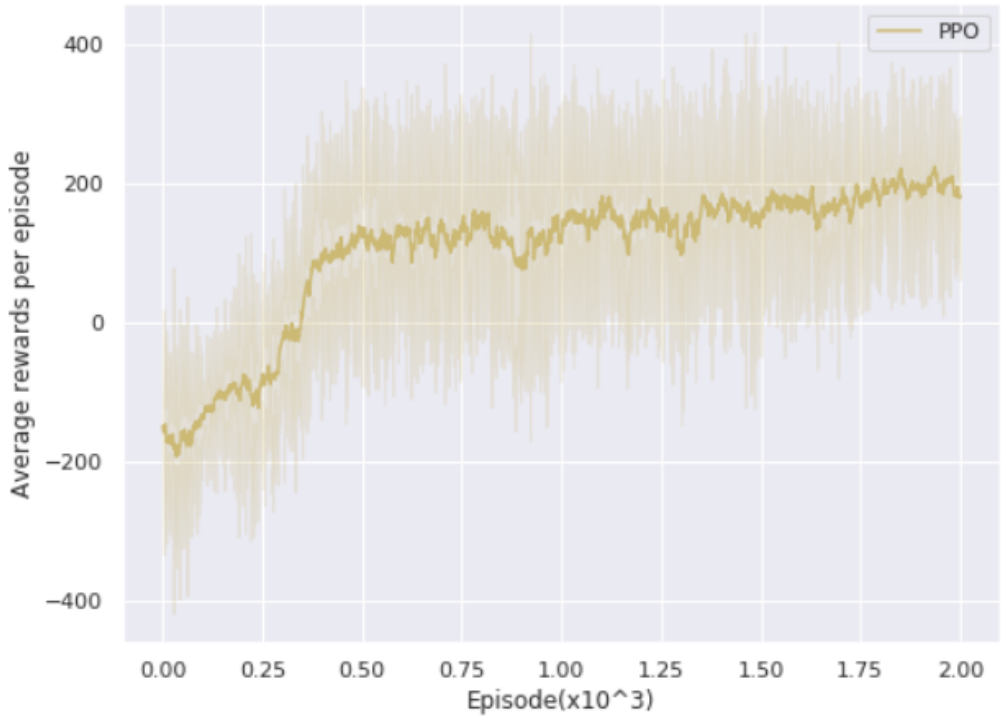

Figure 4.1. Result of Lunar Lander in Discrete Action Space

The variance of PPO agent in this experiment reward is relatively high, which denotes that the agent's performance is quite different in each experiment and the learning is rather unstable. Despite having high variance, with the mean value not fluctuating so much, we can conclude that overall PPO is able to solve the task in all 3 experiments.

4.1.2 Experiment on Continuous Lunar Lander

After completing the experiment in discrete action space, another experiment is conducted on Lunar Lander in continuous action space. We apply a similar method as in the Discrete Experiment, to run the algorithm 3 times and confine each experiment

to be within 2000 episodes. This experiment is conducted using both DDPG and PPO.

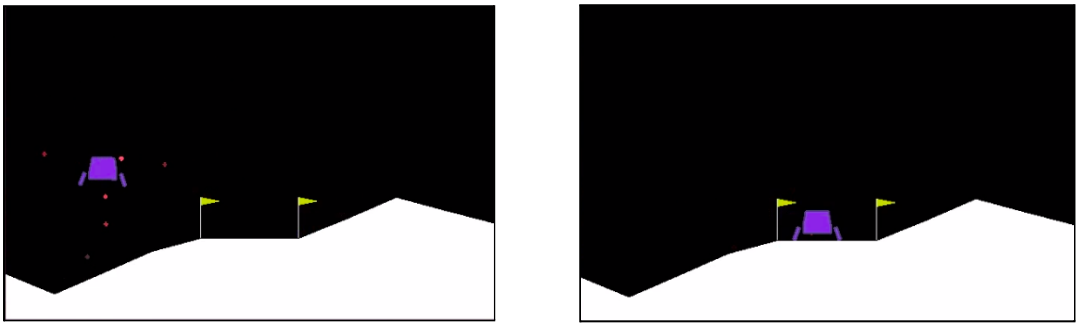

Figure 4.2. PPO Agent Behaviour on Lunar Lander, before (left) and after (right)

Both PPO and DDPG are able to complete the task, and explore most of the possible conditions in the Lunar Lander problem. For example in Figure 4.2, we can see that even though the agent (spacecraft) is far away from the targeted landing location, PPO still manages to find the designated landing spot which indicates the agent has already explored this environment. Similar behaviour is also observed in DDPG. In conclusion, both algorithms are able to achieve optimum solutions. The Result from the experiment shown in Table 4.2 and Figure 4.3.

Table 4.2. Statistics on Continuous Lunar Lander

Algorithm	**Number of steps per Episode**
DDPG	702.610794
PPO	603.685019

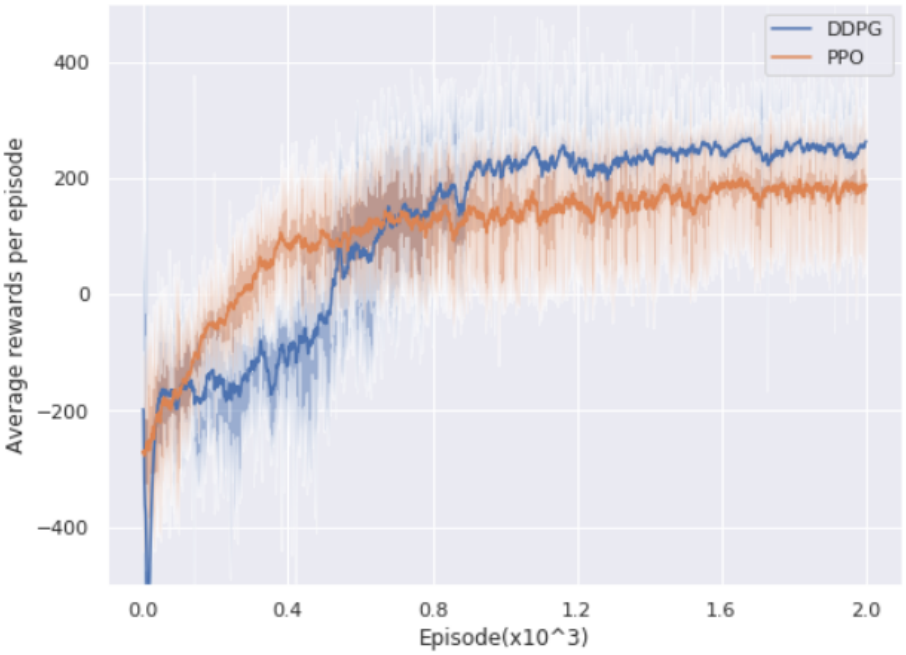

Figure 4.3. Result of Lunar Lander in Continuous Action Space

After training within 2000 episodes, both PPO and DDPG achieve almost similar average reward per episode of around 200. From Figure 4.3, we can see that both PPO and DDPG are able to converge to some value. Even though PPO's average reward per episode value increases slightly faster than DDPG's, DDPG converges to higher average reward per episode in the end. DDPG relatively performs similarly well across all 3 experiments as indicated by the low variance shade. Meanwhile, PPO achieves different values across the 3 experiments since it has slightly larger variance shades.

4.2 Experiment on PyBulletGym Environment

After doing some experiments on Lunar Lander, we conduct another experiment on PyBulletGym, using Walker2d and HalfCheetah environments. As reflected from previous experiments on Lunar Lander, the result will display the average reward per episode from across 3 different experiments. It will also display the variance of the result to assess the stability of the agent.

4.2.1 Walker2D Experiment

The experiment is conducted within 4000 episodes. We will observe the result from both PPO and DDPG in this experiment to assess the performance of both algorithms in more complex problems.

The behaviour of the agent in this experiment is shown in Figure 4.4. In this experiment, PPO agent is able to maintain its balance and walk a few steps. Meanwhile DDPG agent is unable to maintain its balance and keeps falling which makes the episode terminate.

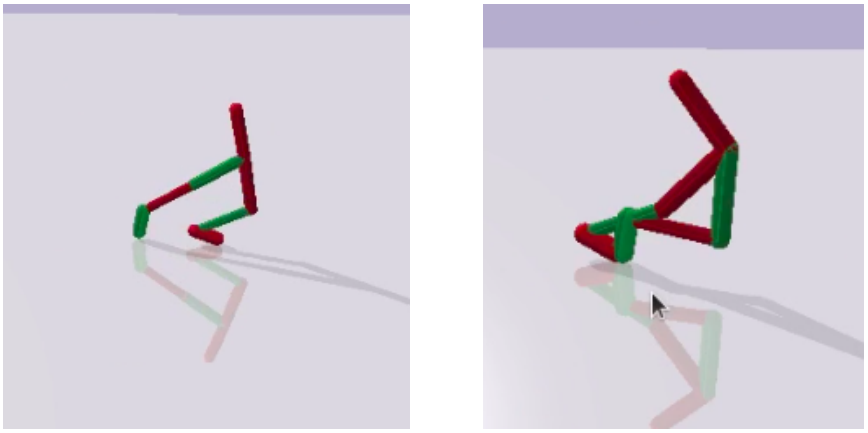

Figure 4.4. Agent Behaviour on Walker2D, PPO (left) and DDPG (right)

The result plot from Walker2d experiment is shown on Figure 4.5. PPO performs much better than DDPG by achieving higher average reward per episode as PPO achieves mean value around 1000 within specified episodes. PPO agent starts the learning slowly. However after 2000 episodes, the performance starts to increase significantly as it has found the corresponding good policy. After finding good policy, PPO agent is able to exploit the good policy and keep learning and achieve higher reward value. Since the graph is still increasing, the agent might have not converged to the expected solution, but if we increase the training episode, it might converge to the optimum solution. However, PPO has a very large variance on its average reward per episode. This means that in some experiments, PPO achieves good results, but fails in other experiments, which lower the mean and increase the variance. This might be because of different parameter initialization on each neuron

in the beginning of each experiment which makes the agent in that experiment fail to converge in several experiments.

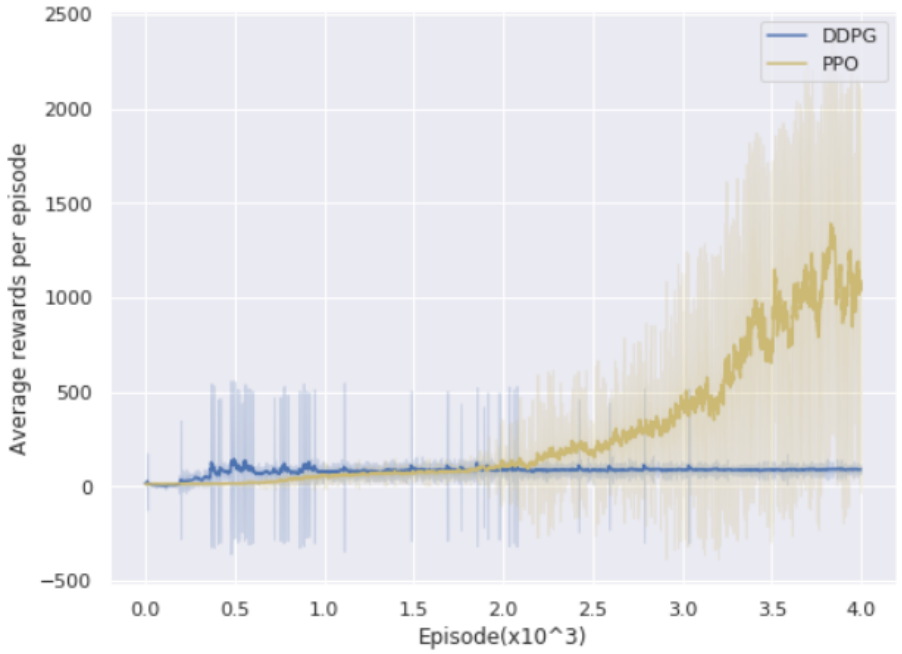

Figure 4.5. Result on Walker2d Experiment

On the other hand, from Figure 4.4, we observe that DDPG keeps getting stagnant and converges to value at around 100, which is very low for this problem, and indicates its failure to complete the task. DDPG has some problems in exploring the environment and possible good action which made DDPG fail to learn the optimal policy. DDPG's average reward per episode has low variance, indicating that DDPG performs similarly and fails in all experiments. In conclusion, PPO is able to complete the task, meanwhile DDPG fails to learn and complete the task

4.2.2 HalfCheetah Experiment

The half cheetah experiment is conducted within 3000 episodes. We will observe the result from both PPO and DDPG like the previous experiment to assess the performance of both algorithms in complex problems.

The result from the experiment is shown by Figure 4.6. In this experiment, PPO performs well by having stable increasing performance after 1500 to the end of the

training. Even though it is relatively slow in retrieving good policies in the beginning, it manages to find good policies and learn from that onwards to get higher rewards. PPO result has not converged to any value, but as we increase the training episodes, it might converge to some value and find optimum solutions. The variance of PPO agent is relatively high, indicating the agent performance varied in each experiment.

Overall, the average reward per episode of PPO at the end of the training achieves similar value as compared to the experiment by openai [28], which indicates the PPO agent performs well in this experiment. Besides, PPO agent behaviour in the experiment also meets the expectation of being able to walk and maintain its balance and prevent it from falling.

In contrast, DDPG does not achieve the expected result in this experiment. It fails to find the optimum solution just like the previous experiment. DDPG keeps failing and is not able to learn and get higher rewards until the end of the training. Even though it has low variance, overall, the performance is far from expectation to be able to solve the task.

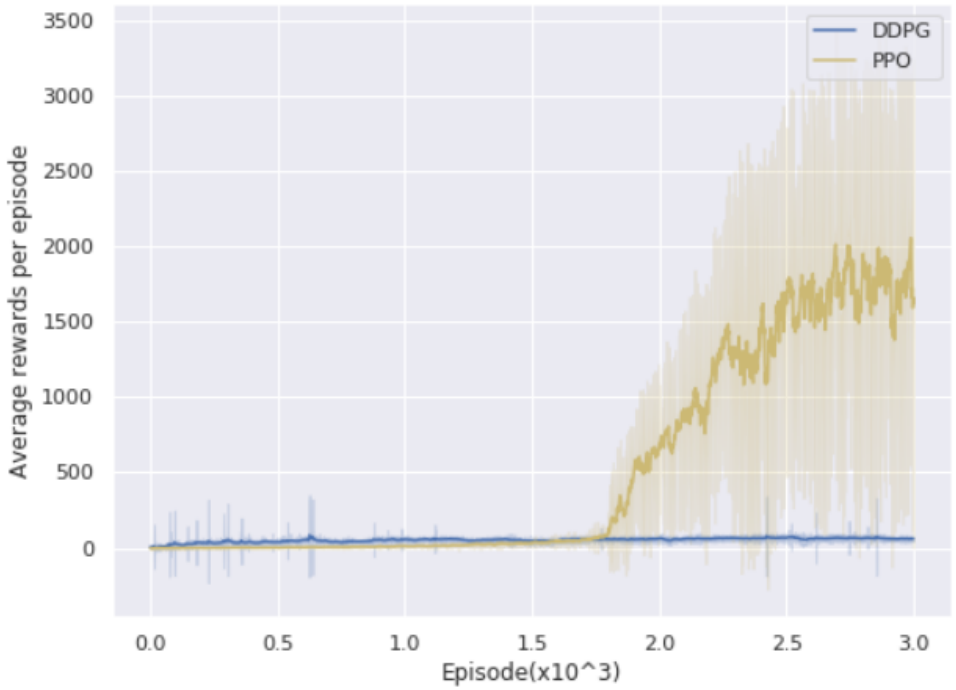

Figure 4.6. Result of HalfCheetah Experiment

There are some possible reasons for the failure of DDPG on both experiments. One of the possible reasons is because of the deadlock situation. In these environments, the episode terminates early when it fails. This will cause the agent unable to get useful actions for the replay buffer and cause deadlock situation. Therefore, DDPG uses randomly chosen values from the replay buffer to learn, however, in those opportunities, DDPG picks all the bad actions since no good actions are obtained. This might ruin the policy and worsen the performance of the agent. Another possible reason might be the usage of deterministic policy that makes DDPG unable to generate random action in the deadlock situation, and keep using bad policy.

Additionally, during the training, the DDPG agent might picks action from the experience buffer with high negative critic value. Because DDPG does not limit the gradient step for the policy, agent will normally make this action less probable and cause extreme changes in policy. It is the reason why clipping in PPO algorithm is needed, so this kind of critic value will not change the policy that much.

4.3 Overall Result

After conducting several experiments, we can draw some conclusions from the algorithms that we use, before applying them to our main experiment. PPO is relatively stable and shows increasing results with low fluctuation in all experiments. With clipping on its policy gradient, it achieves stability. It works well in both continuous action space and discrete action space. Even though in some experiments it fails to achieve the optimum solution, it never had drastic change in the result, which means the policy is relatively stable. The convergence in PPO is guaranteed even though sometimes it converges to local optima only.

Meanwhile for DDPG, the performance is not similar in all experiments. Failure in the last two experiments gives a picture of the unexpected behaviour of DDPG in more complex problems. With deterministic policy, DDPG will have some problems with the kind of environment that terminates early like HalfCheetah and Walker2D.

Other than that, when the policy gradient is too big, it may break the policy and make the agent fail to converge. However in the best case scenario, DDPG may learn faster than PPO if the agent keeps picking good buffer values that maximize action behaviour.

4.4 Experiment on Highway Environment

After we validate the algorithms on the previous environments, we will start to conduct experiments on highway simulation and parking simulation. The experiments are conducted in the same manner as the Lunar lander experiment, with a maximum of 2000 episodes for each of the 3 experiments.

4.4.1 Highway simulation

In this experiment, the experiment is conducted using standard PPO and PPO with an additional attention layer. The experiment is conducted to show the effect of attention in completing the task. In this experiment, we train the agent within 2000 episodes for each experiment.

By using attention, correlation between each car can be visualized, denoted by the green and blue lines shown in Figure 4.7. The blue line becomes thicker as the other car getting closer, and the green line becomes thicker as the other car getting farther.

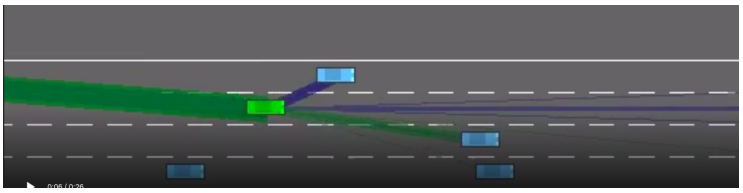

Figure 4.7. Attention Visualization

Table 4.3 shows the statistics from the highway experiment. Completion rate denotes the number of times the ego car finished 100 steps in one episode. The completion rate for PPO with attention is significantly higher than that of normal PPO. The completion rate almost reached 50% during the learning process. Meanwhile, the completion rate is really low for normal PPO.

Table 4.3. Result of Highway Simulation

Algorithm	Average Completion Rate in Percentage
PPO	16.675%
PPO with attention	43.55%

The average reward per episode result from the experiment is shown in Figure 4.8. The result achieved by PPO with attention is significantly higher than normal PPO. And also, the increase of reward is higher in the case with attention layer, which indicates the agent learns faster to solve the problem. Meanwhile in the PPO model, the increase of reward is slower and the agent learns in a slow manner.

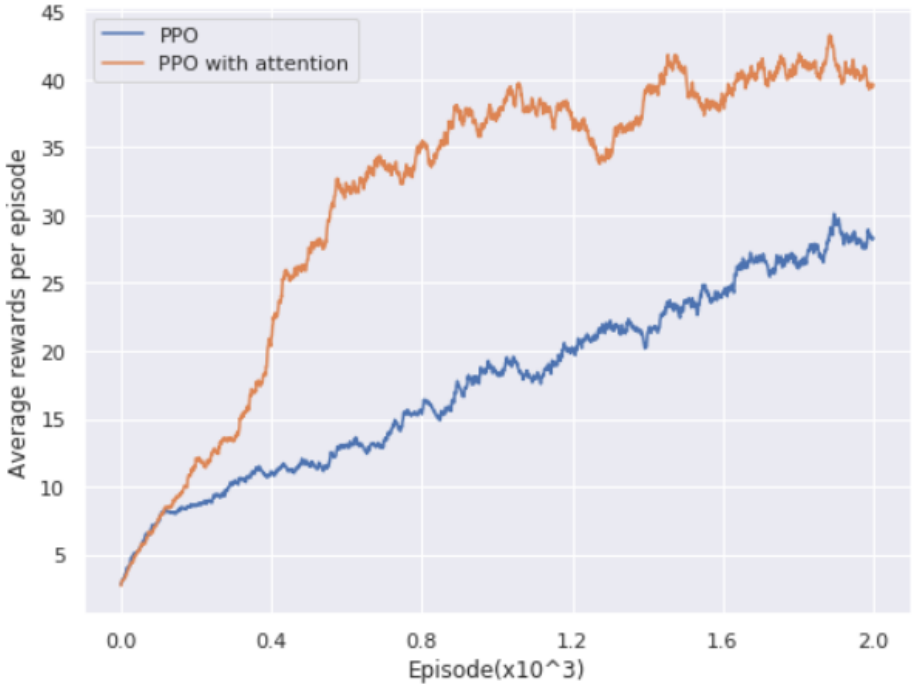

Figure 4.8. Result of Highway simulation

Beside comparing values from the result, the final behaviour of the PPO agent with attention layer is shown in Figure 4.9. The agent with attention layer is able to change lanes to avoid collision. The agent also moved faster and increased its

velocity to get higher rewards. Overall, PPO agent with attention is able to solve the problem.

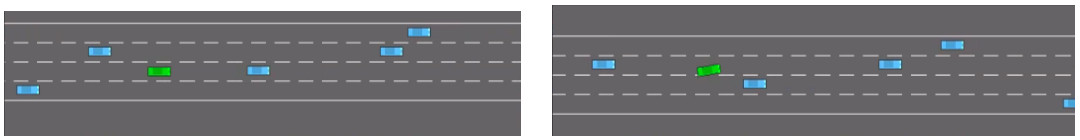

Figure 4.9. PPO Agent Behaviour with Attention, before (left) and after (right)

Meanwhile, without attention layer included, as shown in Figure 4.10, the normal PPO agent fails to decide when to change its lane and collides with another vehicle, even though the other lane is empty. The agent fails to observe the environment from the state given, and is not able to solve the problem within specified training time.

Figure 4.10. PPO Agent Behaviour without Attention, before (left) and after (right)

Overall, the addition of the attention layer really increases the performance of the agent. Compared to normal PPO agent, our model with attention layer included achieves a higher completion rate, and higher reward. The learning also tends to be faster than that of normal PPO. This is because with attention layer, the car gets the correlation information, other than the velocity and position only. This allows the agent to get a more complete representation about the environment and will be able to identify the other cars. As a result, the agent can pick better actions to get higher rewards and avoid collision. Hence, the learning becomes faster for solving this task. Other than this, attention layers might be useful for other reinforcement learning problems where the order of the state is constantly changing.

4.4.2 Parking environment

In this experiment, the experiment is conducted with PPO, normal DDPG, and DDPG that extended with HER. The experiment is conducted to show the effectiveness of each algorithm. Since it uses negative rewards to trigger the agent to

complete the task as fast as possible the average reward per episode in this experiment will be in a negative manner.

From the behaviour of the agents, as shown in Figure 4.11, DDPG agent is not able to decide how to go to the target position, and in most of the experiments, the car is moving away from the target and the parking area. This denotes that all training is not useful and the agent does not learn anything about how to get closer to the target. The similar behaviour is also observed from PPO agent.

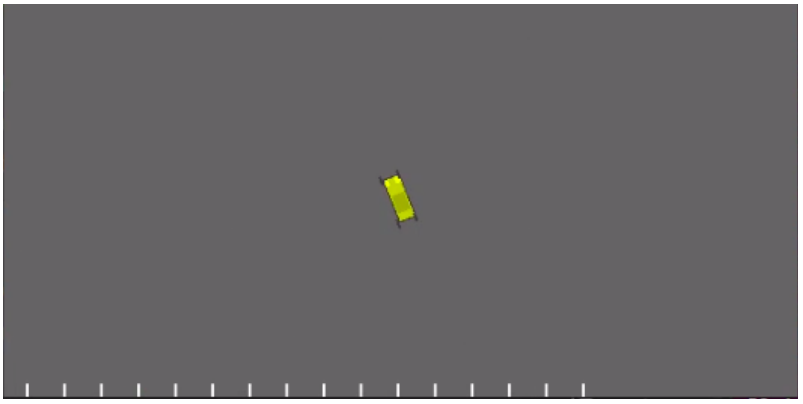

Figure 4.11. DDPG Agent in Parking Simulation, failed to find solution

On the other side, HER agent is able to control the car to move closer to the designated target area in almost all the cases as shown in Figure 4.12. Most of the time the car stucks near the target as it is afraid to do another action that would increase the penalty, but sometimes the car able to hit the target and align in the correct position.

Figure 4.12. HER Agent Behaviour, stuck in local optima (left) and reaches target point (right)

The performance of the agent is shown in Table 4.4. Both PPO and DDPG fail to complete the challenge above 10% of the time, and even PPO unable to succeed above 1% of the time. Meanwhile, HER achieves the highest successful rate of about 20%, significantly better than DDPG and PPO. This percentage might be increased if the number of experiments and episodes increase as the agent already found the optimum solution, and will exploit the optimum solution from there.

Table 4.4. Completion Rate of Parking Simulation

Algorithm	**Average Completion Rate in Percentage**
DDPG	1.7920 %
HER	21.829 %
PPO	0.5202 %

From Figure 4.13, both PPO and DDPG algorithms fail to retrieve higher rewards within 2000 episodes. They tend to be stagnant at some levels, and still in the state of exploration. Both PPO and DDPG have not found any solution to solve the problem.

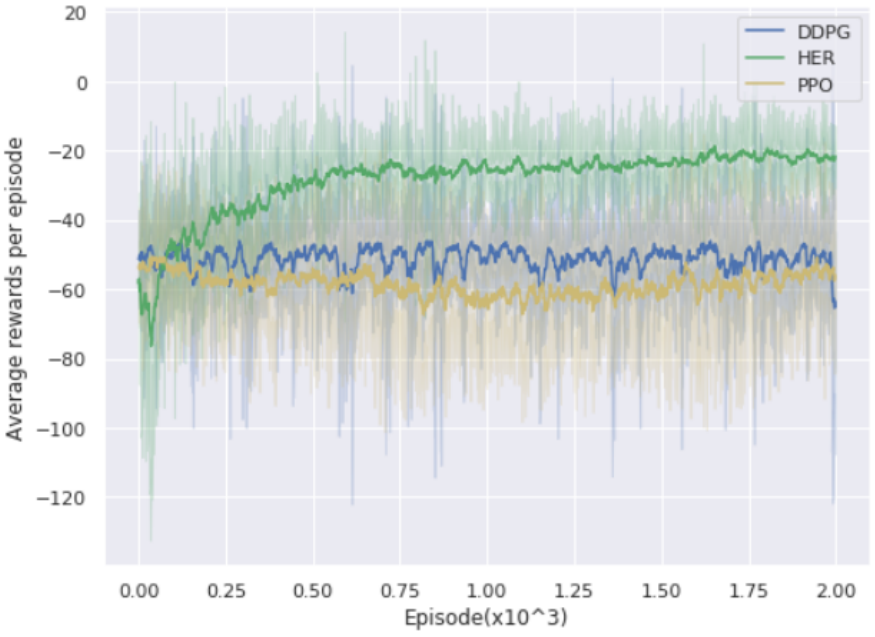

Figure 4.13. Result of Parking Simulation

On the other hand, using Hindsight Experience Replay (HER), it really improves the agent in finding optimum solutions, and hastens the exploration process. After 750 episodes, the agent tends to explore all possible solutions needed to complete the task, and from there it can maximize all the steps that lead to the solution. HER method, which uses fake goal state to let the agent assumes that it has achieved the goal in each iteration, has proven to improve the agent in solving exploration problems and finding optimum solutions faster.

During the experiment with a goal environment like this parking simulation, a normal reinforcement learning algorithm needs longer time to learn completing the task. Especially when the environment is really big, the exploration will be very rich. Using standard PPO and DDPG, the agent failed to find any solutions and get out of the exploration state within specified episode, since all rewards given are similar and the agent cannot compare one state with another state. By using HER, after finishing one episode and terminates at some last state, this last state will act as a fake goal, for this episode, and it makes the agent receive positive feedback for this state in this episode. The agent has more clear information about achieving this last state, and overall, if it applied to more states, the agent will have more information about the environment. Now the agent knows which state is better to achieve this last state which may lead to the real goal state in other episodes. Later if the agent finds the goal state that is already visited as the last state in an episode, the agent can easily achieve this goal state.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

Throughout this one year project, we have proposed a new method in solving the problem in connected autonomous vehicles using PPO algorithm combining with attention layer. We begin the project with no prior information in reinforcement learning, and manage to solve the problem and meet the benchmark of the result from previous reinforcement learning implementation. After we managed to solve this classic environment, we were trying to solve our main challenge in the highway environment. We managed to solve the highway environment problem using PPO that has not been implemented in the environment. We found the limitation on this problem using standard PPO algorithm and managed to identify the issue that causes the failure, which is a sequencing issue. With an attention transformer, the problem can be solved, and a PPO algorithm can solve the problem. Moreover, in general, it can solved any reinforcement learning problem that depends on the sequence of the data.

Other than that, we solved the parking simulation using DDPG combined with HER. After we found the limitation on standard PPO and DDPG, by identifying the problem on reward function and type of environment, we managed to propose and recommend HER to be added to DDPG, which is suitable for this kind of environment. We are thus able to solve the exploration problem in large continuous state-action space, and overall it can be implemented as well to another goal environment which has large space.

5.2 Future Work

5.2.1 Integrate with Further Control

In this project, we managed to solve the problem of highway simulation and parking simulation. For highway simulation which has discrete actions, the set of actions can be developed into further detailed action. For example, for turning left and right, it can be developed into controlling the degree of steering of a real vehicle. And for accelerating and maintaining velocity of the car, it can be developed into controlling the level of the brake and throttle in real vehicles in future development. Same application can be applied to the parking simulation as well

5.2.2 Traffic Addition to Simulation

We managed to solve the problem in highway driving simulation and parking environment. However, to reflect the simulation to be more similar to real life scenarios, more traffic is needed to simulate real life autonomous driving car simulation. More scenarios can be added, such as going through intersections with traffic lights, encountering more vehicles, and also having more pedestrians. Moreover, with adding traffic like pedestrians, this can provide training for avoiding pedestrians, to improve the safety of the user in the vehicle and the pedestrian, which must be implemented in future development.

5.2.3 Addition of Sensor Data

In this project, we only used the data from the concept of connected autonomous vehicles, i.e. we only used data from other cars. However, there are more considerations to be accounted for, like things that cannot be detected by the connected autonomous vehicle, such as pedestrians, cats, and road barriers. In order to solve this problem, more sensors such as cameras and 3D scanners need to be added to the vehicle and simulation so that vehicles can collect these data and then learn to solve the problem that involves these objects. This way, autonomous vehicles can then move on highways and roads safely and reliably.

References

[1] D. Elliott, W. Keen, and L. Miao, “Recent advances in connected and automated vehicles,” *Journal of Traffic and Transportation Engineering (English Edition)*, vol. 6, no. 2. pp. 109–131, 2019, doi: 10.1016/j.jtte.2018.09.005.

[2] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. A Bradford Book, 2018.

[3] OpenAI *et al.*, “Dota 2 with Large Scale Deep Reinforcement Learning,” *arXiv preprint arXiv:1912.06680*, Dec. 13, 2019.

[4] S. Aradi, T. Becsi, and P. Gaspar, “Policy Gradient Based Reinforcement Learning Approach for Autonomous Highway Driving,” *2018 IEEE Conference on Control Technology and Applications (CCTA)*. 2018, doi: 10.1109/ccta.2018.8511514.

[5] A. Feinberg, “Markov Decision Processes: Discrete Stochastic Dynamic Programming (Martin L. Puterman),” *SIAM Review*, vol. 38, no. 4. pp. 689–689, 1996, doi: 10.1137/1038137.

[6] W. G. I. Sammut C., “Bellman Equation,” *Encyclopedia of Machine Learning. Springer, Boston, MA.*, 2011. https://doi.org/10.1007/978-0-387-30164-8_71 (accessed Mar. 28, 2021).

[7] J. R. Kirkwood, *Markov Processes*. CRC Press, 2015.

[8] G. B. Orr and K.-R. Müller, *Neural Networks: Tricks of the Trade*. Springer, 2003.

[9] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel, “High-Dimensional Continuous Control Using Generalized Advantage Estimation,” *arXiv preprint arXiv:1506.02438*, Jun. 08, 2015.

[10] R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour, “Policy gradient methods for reinforcement learning with function approximation,” in *Proceedings of the 12th International Conference on Neural Information Processing Systems*, Cambridge, MA, USA: MIT Press, 1999, pp. 1057–1063.

[11] V. Mnih *et al.*, “Playing Atari with Deep Reinforcement Learning,” Dec. 19, 2013.

[12] T. P. Lillicrap *et al.*, “Continuous control with deep reinforcement learning,”

arXiv preprint arXiv:1509.02971, Sep. 09, 2015.

[13] V. R. Konda and J. N. Tsitsiklis, "OnActor-Critic Algorithms," *SIAM J. Control Optim.*, vol. 42, no. 4, pp. 1143–1166, Jan. 2003.

[14] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, "Deterministic Policy Gradient Algorithms," in *Proceedings of the 31st International Conference on Machine Learning*, vol. 32, Beijing, China: PMLR, 2014, pp. 387–395.

[15] G. E. Uhlenbeck and L. S. Ornstein, "On the Theory of the Brownian Motion," *Physical Review*, vol. 36, no. 5. pp. 823–841, 1930, doi: 10.1103/physrev.36.823.

[16] K. Wampler and Z. Popović, "Optimal gait and form for animal locomotion," *ACM Trans. Graph.*, vol. 28, no. 3, pp. 1–8, Jul. 2009.

[17] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel, "Trust Region Policy Optimization," *International conference on machine learning*, vol. 37, pp. 1889–1897, Feb. 2015, Accessed: Nov. 07, 2020. [Online].

[18] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal Policy Optimization Algorithms," *arXiv preprint arXiv:1707.06347*, Jul. 20, 2017.

[19] M. Andrychowicz *et al.*, "Hindsight Experience Replay," *arXiv preprint arXiv:1707.01495*, Jul. 05, 2017.

[20] A. Sherstinsky, "Fundamentals of Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) Network," *Physica D: Nonlinear Phenomena*, vol. 404, no. March 2020, Aug. 2018, doi: 10.1016/j.physd.2019.132306.

[21] A. Vaswani *et al.*, "Attention Is All You Need," *arXiv preprint arXiv:1706.03762*, Jun. 12, 2017.

[22] G. Brockman *et al.*, "OpenAI Gym," Jun. 05, 2016.

[23] DLR-RM, "DLR-RM/stable-baselines3." https://github.com/DLR-RM/stable-baselines3 (accessed Mar. 28, 2021).

[24] benelot, "benelot/pybullet-gym." https://github.com/benelot/pybullet-gym (accessed Mar. 28, 2021).

[25] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[26] “Benchmarks for Spinning Up Implementations — Spinning Up documentation.” https://spinningup.openai.com/en/latest/spinningup/bench.html (accessed Mar. 28, 2021).

[27] eleurent, “eleurent/highway-env.” https://github.com/eleurent/highway-env (accessed Mar. 28, 2021).

[28] “Benchmarks for Spinning Up Implementations — Spinning Up documentation.” https://spinningup.openai.com/en/latest/spinningup/bench.html (accessed Mar. 28, 2021).

[29] E. Leurent and J. Mercat, “Social Attention for Autonomous Decision-Making in Dense Traffic,” *arXiv preprint arXiv:1911.12250*, Nov. 27, 2019.

[30] S. Gadgil, Y. Xin, and C. Xu, “Solving The Lunar Lander Problem under Uncertainty using Reinforcement Learning,” *arXiv preprint arXiv:2011.11850*, Nov. 24, 2020.

Appendix

Appendix A. DDPG Original Algorithm

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

Appendix C. HER Algorithm

Algorithm 1 Hindsight Experience Replay (HER)

Given:

- an off-policy RL algorithm $\mathbb{A}$, ▷ e.g. DQN, DDPG, NAF, SDQN
- a strategy $\mathbb{S}$ for sampling goals for replay, ▷ e.g. $\mathbb{S}(s_0, \ldots, s_T) = m(s_T)$
- a reward function $r : \mathcal{S} \times \mathcal{A} \times \mathcal{G} \rightarrow \mathbb{R}$. ▷ e.g. $r(s, a, g) = -[f_g(s) = 0]$

Initialize $\mathbb{A}$ ▷ e.g. initialize neural networks
Initialize replay buffer R
for episode = 1, M **do**
  Sample a goal g and an initial state s_0.
  for $t = 0, T-1$ **do**
    Sample an action a_t using the behavioral policy from $\mathbb{A}$:
      $a_t \leftarrow \pi_b(s_t||g)$ ▷ $||$ denotes concatenation
    Execute the action a_t and observe a new state s_{t+1}
  end for
  for $t = 0, T-1$ **do**
    $r_t := r(s_t, a_t, g)$
    Store the transition $(s_t||g,\ a_t,\ r_t,\ s_{t+1}||g)$ in R ▷ standard experience replay
    Sample a set of additional goals for replay $G := \mathbb{S}$(**current episode**)
    for $g' \in G$ **do**
      $r' := r(s_t, a_t, g')$
      Store the transition $(s_t||g',\ a_t,\ r',\ s_{t+1}||g')$ in R ▷ HER
    end for
  end for
  for $t = 1, N$ **do**
    Sample a minibatch B from the replay buffer R
    Perform one step of optimization using $\mathbb{A}$ and minibatch B
  end for
end for