CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Path Planning using Reinforcement Learning and Objective Data

Master's thesis in Master Programme Systems, Control and Mechatronics

TIAN XIA

Master's thesis in Automotive Engineering

ZIJIAN HAN

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2017

Master's thesis 2017

Path Planning using Reinforcement Learning and Objective Data

TIAN XIA
ZIJIAN HAN

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2017

Path Planning using Reinforcement Learning and Objective Data
TIAN XIA
ZIJIAN HAN

Supervisor: Christos Dimitrakakis, Department of Computer Science and Engineering
Supervisor: Hannes Eriksson, Department of Computer Science and Engineering
Advisor: Name, Company or Institute (if applicable)
Examiner: Devdatt Dubhashi, Department of Computer Science and Engineering

Master's Thesis 2018
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: TORCS Simulation Environment

Gothenburg, Sweden 2018

Path Planning using Reinforcement Learning and Objective Data
TIAN XIA
ZIJIAN HAN
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

With the rapid development of autonomous driving vehicles, decision making for path planning has become advanced and challenging topics. Traditional planning and control methods are usually limited by the difficulty to find good solutions, so deep machine learning has become engineers' focus in order to solve these problems. Several related works using reinforcement learning have been done in the simulation environment TORCS. This thesis will focus on training an vehicle to learn driving at certain target speed on high way condition without collision. A complete learning structure is designed for vehicle system, and a hierarchical learning algorithm will be used with deep reinforcement learning methods. Deep Q learning is used to learn option level of policy, and deep deterministic policy gradient is used to learn primitive action level of policies. Neural networks are used to approximate the value functions. The training results are tested on various set up of opponents vehicles on the track, with the probability of damage recorded and compared.

Keywords: reinforcement learning, TORCS, policy, option, DQN, DDPG.

Acknowledgements

The thesis targets on an interesting and challenging topic of decision making in automotive industry, which hasn't been very much looked into. We would like to thank our supervisor Nasser Mohammadiha and Zenuity to give us such a great opportunity to grab a view on the current development trend of autonomous driving technology.

We also would like to thank to our supervisors Christos Dimitrakakis and Hannes Eriksson from Chalmers for their continuous support. They gave us sufficient guidance and constructive suggestions. Without their help and suggestions, we couldn't finish the thesis work as we expected.

Thanks to our examiner Devdatt Dubhashi for checking the report and following our thesis progress, and also to our opponents Liting and Daoyuan for reviewing our report and offering us useful suggestions.

Tian Xia, Zijian Han, Gothenburg, September 2018

Contents

List of Figures

List of Tables

1

Introduction

1.1 Background

Research on autonomous driving has become a topic of significant interest among industries. The goal of autonomous driving is to let the intelligent vehicles understand the environment and make proper decisions on controlling the motion of vehicles autonomously and safely without human drivers' supervision. With fast development of artificial intelligence algorithms and computational power, machine learning has been widely applied in decision making processes such as autonomous driving tasks. Compared with conventional optimization algorithms such as model predictive control or stochastic optimization algorithms, machine learning algorithms have advantages when handling multiple environments.

The actual driving scenarios in reality are much more complex than in ideal simulators. For example, the vehicle can drive into unknown environments that the behaviour trained by supervised learning may not guarantee the safety of vehicle. Reinforcement learning, among various of proposed methods, is probably the most general framework in this case to learn the self-learning ability of decision making and safe path planning for autonomous vehicles. As a result, reinforcement learning combined with deep neural networks has become a popular and powerful method in training self-driving vehicles to accomplish complex tasks that outperforms conventional methods.

1.2 Related Work

1.2.1 Reinforcement Learning Algorithms for Autonomous Driving

Reinforcement learning algorithms have been widely applied to decision making problems, such as financial market operations and video games strategies. To solve complex autonomous driving tasks which may contain different maneuvers, a number of reinforcement learning algorithms have been experimented in some researches to perform certain self-driving tasks.

Q-learning or deep Q-learning has been widely applied to many self-driving tasks such as simply driving along the track and overtaking. The method of learning overtaking using simple Q-learning for primitive actions is introduced in [8]. In [3], more complex behaviours including overtaking and blocking are learned by Q-learning and the study of overtaking mainly focuses on the generalization issue, which is learning to overtake opponent cars with different behaviours. When using Q-learning in these tasks, the primitive actions in continuous space have to be discretized into limited fixed values as control signals. However, more smooth control signals are required when controlling a vehicle in reality, planning and control in continuous action space would be preferred from drivers' perspective.

Compared with Q-learning, policy gradient methods are considered to be more powerful and have better performance in solving complex tasks in continuous space[4]. Different policy gradient methods that have been applied in robotics are discussed in [9], including finite-difference method, likelihood ration method, "Vanilla" policy gradient and natural Actor-Critic approaches. Instead of stochastic policy gradient, [13] proposed a deterministic policy gradient algorithm with continuous actions. It argues that deterministic policy gradient can be much more efficient than usual stochastic policy gradient by comparing the training results of Atari games with two kinds of methods. Based on the deterministic policy gradient algorithm, [19] uses deep deterministic policy gradient to learn obstacle avoidance of self-driving vehicles with primitive actions. However, there are several problems if the policy gradient method is applied directly to our tasks. First of all, learning primitive actions directly can lead to jerky behaviors. Secondly, it is extremely difficult to determine the reward function. Moreover, methods in [19] cannot learn the policy successfully anymore when there are more dynamic obstacles with complex behaviours on the track.

The methods mentioned above are able to perform well with simple tasks, while for more complex tasks which may consist of several sub-tasks, it is not easy to use a simple learning method to achieve good results anymore. Therefore, the idea of decomposing a complex task into several simply sub-tasks and learning them separately, which is the concept of option framework came to people's mind. Option framework was firstly proposed by Sutton in 1998 in [15], which is a framework for temporal abstraction with the purpose of scaling up learning and planning in reinforcement learning. The idea of the thesis is based on using option framework to create a general solution for complex driving tasks. The option framework method has been developed further in [1], where an option-critic architecture is proposed to tackle the problem of creating abstractions autonomously from data. The architecture uses an actor-critic structure to update intra-policy, it is able to learn both internal policies and the termination conditions of options without extra reward functions and sub-goals. However, it is proved that applying fully general learning algorithm with no prior knowledge can be extremely inefficient in our learning task. In other implementations of autonomous driving, sub-tasks are often assigned with specific customized reward functions and exploration methods. A hierarchical temporal abstraction method called “Option Graph” was used by Mobileye in [12] to perform autonomous lane merging at the junction for self-driving vehicles. In the "Option Graph" each sub-tasks are designed with specific goals such as merging left, merging right and getting ready. In [2] a model-based hierarchical reinforcement learning method combines options framework, model-based planning and Bayesian active learning was implemented to perform taxis' path planning in TORCS.

Based on the implementations above, this thesis introduces a hierarchical reinforcement learning structure for path planning of autonomous driving vehicles in high way scenarios with Q-learning for option level learning and deep deterministic policy gradient for intra-policy level learning. The method combines the deterministic policy gradient[13] and the way of building hierarchical learning layers[2]. However, different from [1] where all the internal policies are automatically learned, the intra-policy and option level policy are trained independently. Since very specific sub-tasks with customized reward and exploration can be defined in this case, it would be more efficient to pre-train each intra-policy and easier to integrate them in the option level. With such learning structure, we believe more sub-tasks with specific purpose can be easily added into the complete task without re-training everything from the beginning.

1.2.2 Virtual Environments for Autonomous Driving Cars

There are several virtual environments specifically used for autonomous driving vehicles simulation, including OpenAI Gym which is specially created to train agents for reinforcement learning, Udacity self driving car simulator which performs better in self driving behavioral cloning rather than reinforcement learning algorithms because it can only handle an ego car with limited numbers of tracks, CARLA which is a newly developed simulation engine for autonomous driving but still lack of many virtual modules to simulate real driving scenarios.

An racing car simulator names "TORCS" is selected for the simulation and reinforcement learning algorithm development in this thesis. TORCS is a state-of-art open source car racing simulator that provides a full 3D visualization, a sophisticated physics engine and accurate vehicle dynamics[7]. Compared with other simulators, TORCS contains more complete physical proprieties that users can customize the environment, for example, tracks can be designed according to requirements and opponent cars with different behaviours can be programmed and added into the environment. In TORCS, the connection between game and client bots are based on UDP communication and each car are programmed as separate client modules. The software architecture of the TORCS are shown in figure 1.1.

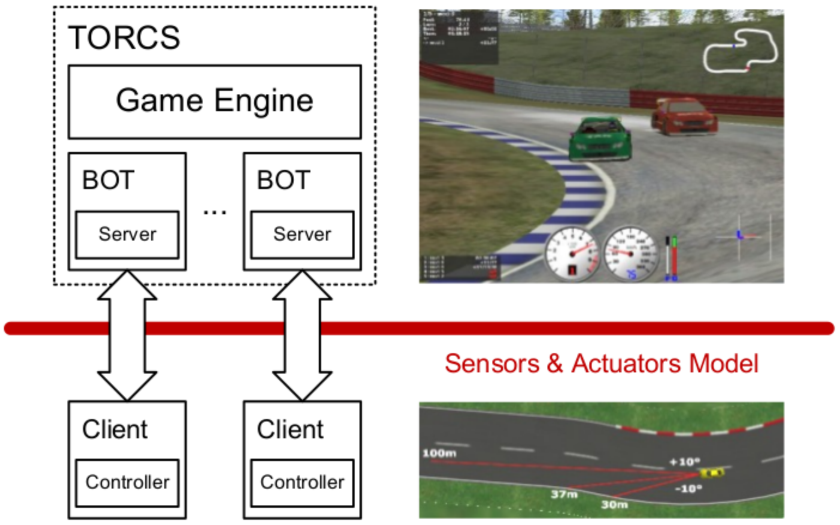


Figure 1.1: The architecture of the competition software[7]

TORCS has been a tool for doing research on reinforcement learning for a long time and some related works have been done, for example, in [8], [3] and [19], algorithms are all tested in TORCS environment.

1.3 Purpose

The main goal of this thesis is to investigate and simulate the behavior of reinforcement learning method applied on an autonomous driving vehicle to perform the overtaking scenario. More specifically, to investigate how the policy search method handles the safe path planning tasks for overtaking. Safety would be the critical consideration when performing the task and efficiency of how the task is finished may also be evaluated.

1.4 Objective

The main objective is to build an agent that could autonomously control the steering, acceleration and braking of a vehicle in some driving tasks. The input information of the intelligent agent is limited to object-level data which is collected from TORCS environment, the object-level data includes states of the ego vehicle(location on the track, speed, acceleration, heading angles, etc), location and speed of surrounding vehicles, lane-markers and also supplementary driving information like speed limits. The agent is supposed to detect a proper triggering time for the vehicle to start the overtaking task and suggest a high-resolution traveling trajectory that followed by a designed controller. Safety and efficiency of driving would be the main concern of agent.

The focus of this thesis will be on the reinforcement learning techniques, including but not limited to imitation learning and policy search for path prediction. The baseline is to implement a policy search algorithm that could find the optimal trajectory for the ego vehicle to overtake opponent vehicles. Safety constraints will be added on the policy during learning to guarantee the safe driving of vehicle. In the autonomous driving scenario, option framework is used in the highest hierarchy to make high level decisions including overtaking and following. Policy gradient, which is the method we mainly focus on, will be used to learn to plan path for some time horizon in the future under the overtaking scenario. Meanwhile, a low level PID controller will be used to follow the generated path. The agent will then be evaluated in several traffic scenarios where the opponent vehicles have different behaviours. The agent should make proper decisions when to begin overtaking or following and be able to complete these behaviors safely and efficiently.

1.5 Limitations

Since it is impossible to test our algorithms in real driving scenario currently, there are a lot of limitations in this thesis project because of the difference between simulation and reality. The real autonomous driving problem is extremely difficult because of the complexity and uncertainty of the environment, the action of other vehicles could be unpredictable sometimes and the sensor data is usually generated with noise.

However, the simulation environment is much more ideal and simplified. First of all, objective data is provided in the simulator so that we get absolute accurate data from the environment without of any noise. Furthermore, road conditions and driving behaviours of other vehicles are strictly designed or limited so that the basic idea of the algorithm can be verified efficiently. Last but not least, our priority is to accomplish overtaking task by generating a path, efficiency, comfort and other factors will also be considered but not with a very high priority.

2
Theory

2.1 Basis of Reinforcement learning

2.1.1 Elements of Reinforcement Learning

In this section the basic elements and concepts in reinforcement learning, including the concept of policy, reward, value function, models, MDPs will be introduced. Besides, the characteristics and categories of reinforcement learning will be discussed as the background knowledge of the methods used in this thesis.

Policy - A policy reflects the behaviour of the agent, it defines how the agent will take actions according to its perceived states[14]. A policy can be very simple, such as a function matching states with actions or a lookup table, it can also be a searching process with more complex computation or represented by a neural network. Generally, policy is extremely important to us because it decides how the agent interacts with the environment.

Reward - Reward is what the agent received from the environment every time after it takes an action. If the agent gets a high reward, we consider the corresponding (state, action) tuple as 'good' states and if the agent gets a low reward we consider these states as 'bad' states. Set a proper reward function is also extremely important because it is the only thing that connect the agent with the environment and all that the agent learns is based on the reward it gets.

Value Function - Value function is the long-term reward. The value of a states means the total discounted reward that the agent receives from the present state to a few steps further in the future. The discounted reward means that reward in the future can be multiplied with a discount factor as its weight. Value is meaningful in reinforcement learning because the behaviour of an agent should not be only decided by the immediate reward, instead, looking further to the future can lead to a better learning result.

Model - A model is a set of the information from the environment that can be used by the agent to predict the following state and reward after it takes an action. The two most important kinds of model are statistic models, or we can call it distribution models, and sample models. The statistic models predicts the probabilities of the next possible states and rewards. The sample models sample according to the possibilities and finally produce only one of them as the result.

Agent and Environment - The agent and the environment are two key element in reinforcement learning, the agent learns from its interaction with the environment and make decision according to the *state* and *policy*. At each time step of the learning process, the agent receive information from the environment as its *state* and takes an *action* based on its *policy*, the agent will then go into a new *state* and the environment will feed back a *reward* to the agent.

2.1.2 Characteristics of Reinforcement Learning

There are several aspects that make reinforcement learning different from other machine learning methods. First of all, there is no supervisor in reinforcement learning, instead, a reward signal will tell the agent what is good and what is bad. Secondly, the feedback is delayed, which means the agent cannot get the feedback immediately but in the future, thus the agent need to take the reward in the future into consideration. Last but not least, what the agent do will affect the environment and the subsequent information it receives on the future will be different.

2.1.3 Categorizing of Reinforcement Learning

Generally, there are two kinds of reinforcement learning methods, value based reinforcement learning and policy based reinforcement learning. In value based reinforcement learning, the value, or action-value is approximated according to the parameters θ and the policy is directly generated from the value function. However, in policy based reinforcement learning, there is no value function and the policy is parameterized directly.

$$\pi_\theta(s,a) = P[a|s,\theta] \tag{2.1}$$

Equation 2.1 shows that the policy is parameterized by θ and the policy is learned by updating the parameters θ. For some cases like tasks with large and continuous state and action space, policy based reinforcement learning has several advantages compared to value based reinforcement learning. It has better convergence properties with higher efficiency in high-dimensional and continuous action spaces, and can learn stochastic policies. However, a local optimum may occur and it is usually hard to evaluate such policy.
Actor-Critic is a special method that combines both value function and parameterixed policy, which means that the value function and policy are both learned, this method will be introduced later.

2.1.4 MDPs and Bellman Equations

Finite Markov Decision Processes(MDPs) are the framework of standard reinforcement learning which formally describe an environment for reinforcement learning. In MDPs, the environment should be fully observable which means that the current state can characterize the process completely.
MDPs follow the following property,

$$P[S_{t+1}|S_t] = P[S_{t+1}|S_1, ..., S_t] \tag{2.2}$$

which means that the current state includes all the necessary information that the agent needs and what happens in the future is only decided by the present state and independent of the past.
In the MDPs framework, the agent interacts with the environment at discrete and lowest-level time scale[15], at each time step t=0,1,2,... the agent receives information from the environment and consider itself at a *state* $s_t \in S$. The agent chooses an action $a_t \in A_t$ according to its policy, this action will lead the agent to a new state s_{t+1} and the environment will give a reward r_{t+1} as feedback to the agent, the agent will then improve its policy according to the MDPs.
The MDPs can be represented by the following sequence:

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, ... \tag{2.3}$$

$$\begin{aligned} v_\pi(s) &= E_\pi[G_t|S_t = s] \\ &= E_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)[r + \gamma E_\pi|G_{t+1}|S_{t+1} = s'] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')] \end{aligned} \tag{2.4}$$

Bellman equation shows how to represent the value of the current state by the states in the future, 2.4 shows that the value of a state s can be expressed by the sum of the immediate reward and the value in the next state multiplied by a discount factor γ which represents the weight of the next state's value, the larger γ is, the more we look ahead from the current state and take future rewards into consideration.

2.2 Value Function Approximation

Function approximation is using a function to estimate the true value which is difficult to calculate directly. For large MDPs with too many states and actions in memory, it takes too much space to store all the state value or state-action value pairs in the look-up table, and it is also very slow to learn the value of each state individually. To solve this problem, function approximation is used for large MDPs to estimate value function by $\hat{v}(s, \mathbf{w}) \approx v_\pi(s)$ or $\hat{q}(s, a, \mathbf{w}) \approx q_\pi(s, a)$ instead of using a lookup table.

There are many types of function approximators, including linear combinations of features, neural network, decision tree, nearest neighbour, etc. Here in our case, only differentiable function approximators are taken into consideration, like linear combinations of features, and neural network.

Furthermore, with a differentiable function approximators (neural network is mainly used in this thesis), a training method suitable for non-stationary data is required. Gradient descent is the most commonly used method to update the weights of function approximators.

2.2.1 Stochastic Gradient Descent

Assuming $J(w)$ to be a differentiable objective function with parameter vector w. The gradient of $J(w)$ is defined as:

$$\bigtriangledown_{\mathbf{w}} J(\mathbf{w}) = \begin{pmatrix} \frac{\partial J(\mathbf{w}_1)}{\partial \mathbf{w}_1} \\ \vdots \\ \frac{\partial J(\mathbf{w}_n)}{\partial \mathbf{w}_n} \end{pmatrix} \tag{2.5}$$

To find the local minimum of $J(\mathbf{w})$, gradient descent will adjust parameter $\mathbf{w}$ in the direction of negative gradient by:

$$\triangle \mathbf{w} = -\frac{1}{2}\alpha \bigtriangledown_{\mathbf{w}} J(\mathbf{w}) \tag{2.6}$$

where α is the learning rate.

And value function approximation by stochastic gradient descent is to find the parameter vector $\mathbf{w}$ the minimize the mean-squared error between approximate value function $\hat{v}(s, \mathbf{w})$ and true value function $v_\pi(s)$. The mean-squared error is defined as:

$$J(\mathbf{w}) = \mathbb{E}_\pi[(v_\pi(S) - \hat{v}(S, \mathbf{w}))^2] \tag{2.7}$$

The gradient descent of equation 2.7 can be calculated as:

$$\begin{aligned} \triangle \mathbf{w} &= -\frac{1}{2}\alpha \bigtriangledown_{\mathbf{w}} J(\mathbf{w}) \\ &= \alpha \mathbb{E}_\pi[(v_\pi(S) - \hat{v}(S, \mathbf{w})) \bigtriangledown_{\mathbf{w}} \hat{v}(S, \mathbf{w})] \end{aligned} \tag{2.8}$$

While stochastic gradient descent samples the gradient by:

$$\triangle \mathbf{w} = \alpha(v_\pi(S) - \hat{v}(S, \mathbf{w})) \bigtriangledown_{\mathbf{w}} \hat{v}(S, \mathbf{w}) \tag{2.9}$$

And the expected update of weight parameters using equation 2.9 is equal to full gradient update.

2.3 Q learning

Q-learning is known as an off-policy TD control algorithm first introduced by Watkins in 1992 [17], which was an early breakthrough in reinforcement learning. The decision making process of Q learning is based on action-value of the current state, and the action with highest value would be considered the best. During learning, the concept of update action-value is to use the difference between actual value and estimated value, times a discount factor to update the value. The main update equation used in Q learning is:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \tag{2.10}$$

where $Q(s_t, a_t)$ is the action-value function, α is learning rate, r_t is the reward received after taking action a_t.

In Q learning an action-value function Q is learned to directly approximates $q*$, the optimal action-value function, without taking care of the policy. While the policy still have effect during learning, since it will decide which state-action pairs to visit.

2.3.1 Tabular Q-learning

The state and action for original Q learning algorithm are both discrete, and they can be represented by pairs or in a tabular format. Therefore here we use tabular Q-learning for common Q learning algorithm to give a different name with the DQN mentioned below. The basic algorithm [14] is given below.

```
Tabular Q-learning;
Initialize the action-value function Q(s,a) arbitrarily, for all $s \in \mathcal{S}^+$, $a \in \mathcal{A}(s)$;
for $episode = 1$ to $k$ do
    Initialize s;
    for step = 1 to MAX_STEPS or termination do
        Choose action $a$ from $s$ using policy derived from Q (e.g., $\epsilon$-greedy);
        Taking action $a$, observe reward $r$ and next state $s'$;
        Update Q with $Q(s,a) \leftarrow Q(s,a) + \alpha[r_t + \gamma \max_a Q(s',a) - Q(s,a)]$;
        Update latest state $s \leftarrow s'$;
    end
end
```

Algorithm 1: Q-learning(off-policy TD control) for estimating optimal policy

2.3.2 Deep Q Network (DQN)

As the increasing complexity of tasks as well as state space changing from discrete to continuous, original tabular Q learning is no longer able to approximate the value function. Therefore, Deep Q network (DQN) is developed with a combination of deep neural networks and Q learning to accomplish the function approximation in continuous state space.

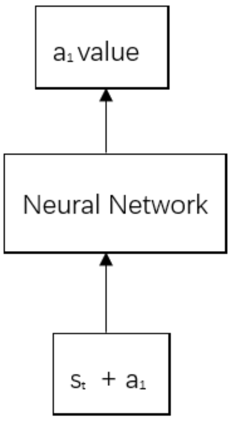
The overall algorithm can be described as:

```
Deep Q-learning with experience replay
Initialize reply memory D to capacity N;
Initialize action-value function Q network with random weights $\theta$;
Initialize target action-value function Q network with weights $\theta^- = \theta$;
for $episode = 1$ to $k$ do
    Initialize state $s_1$ and pre-process the state as the input format of neural
      network $\phi_1 = \phi(s_1)$;
    for step = 1 to MAX_STEPS or termination do
        With probability $\epsilon$ select a random action $a_t$, otherwise select
          $a_t = \arg\max_a Q(\phi(s_t), a; \theta)$;
        Execute action $a_t$, observe reward $r_t$ and next state $s_{t+1}$;
        Process state $\phi_{t+1} = \phi(s_{t+1})$;
        Store trasition $(\phi_t, a_t, r_t, \phi_{t+1})$ in memory D;
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D;
        Set $y_j = ...$;
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the
          network parameters $\theta$;
        Every C steps reset target network weights $\hat{Q} = Q$;
    end
end
```

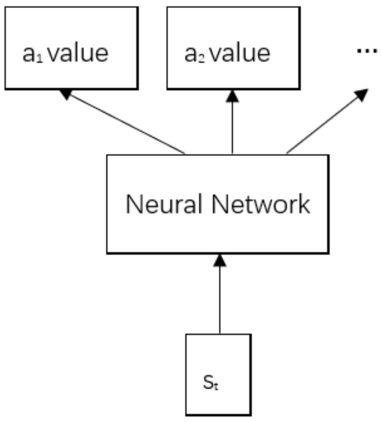
Algorithm 2: Deep Q-learning with experience reply

2.3.2.1 Use of Neural Network

Neural network is used as function approximator to estimate the action-value function in Q learning, which means it can take continuous state and action space as input for network, and output the value for action (shown in figure 2.1a), or take the state as input and output the value for all possible actions(shown in figure 2.1b).



(a) Q function network structure 1



(b) Q function network structure 2

Figure 2.1: DQN neural network structure

2.3.2.2 Update of Neural Network

The update method of neural network is similar as equation 2.10, using gradient descent. Temporal difference is used to estimate the difference between estimated and actual Q value for current state, and using this as the error to update the weights of neural network.

2.3.2.3 Experience Replay and Fixed Q-targets

Another two important benefits for DQN are that it uses experience reply and fixed Q-targets.

It is found that approximation of Q-value using non-linear functions like neural network is not very stable. The most common method to overcome this problem is to use experience replay. Experience replay is a method that remember the previous sampled experience. This means during the updating of network, samples of experience are randomly selected from the memory buffer instead of most the recent transition, which will greatly improve the stability and efficiency of network.

Take value updating for example, given experience consisting of <state, value> pairs $D = \{\langle s_1, v_1^\pi \rangle, \langle s_2, v_2^\pi \rangle, ..., \langle s_T, v_T^\pi \rangle\}$, following two steps will be repeated: (1) sample a state-value pair from experience;(2) apply stochastic gradient descent update $\triangle_{\mathbf{w}} = \alpha(v_\pi(S) - \hat{v}(S, \mathbf{w})) \bigtriangledown_{\mathbf{w}} \hat{v}(S, \mathbf{w})$, until it converges to optimal solution.

Fixed Q-targets is another way used in DQN to break the relevance between data samples and policy, as the concept of Double Q learning introduced in [16]. It uses two networks with same structure but different updating frequencies: one evaluation network(used to calculate estimated Q value $Q(s)$) with regular frequency of update, and one target network(used to calculate actual TD Q value $r + \gamma \max Q(s')$) which update at a lower frequency.

2.4 Policy Gradient

In policy based reinforcement learning, the generation of action is different from value based reinforcement learning which output the action based on values. Instead, action is given directly through policy. A policy can be directly represented by an independent function approximator, such as a linear combination of the input states or a neural network that takes states as input and actions as output with its own parameters.

Compared with value based learning like Q learning, the greatest advantage of estimating action directly is that it is able to deal with problems in continuous state and action space. While for Q learning, it would be impossible to save the values for too large number of state and action space.

2.4.1 Policy Objective Functions

Given a policy $\pi_\theta(s, a)$, the best θ for the policy is supposed to be found and the policy objective function $J(\theta)$ is used to measure the quality of the policy.
The policy objective function is usually defined as a function related to the value function, for example, in episodic environments, the objective function is defined as the start value $J_1(\theta) = V^{\pi_\theta}(s_1) = E_{\pi_\theta}[V_1]$. In continuing environment, it is defined as the average value $J_{av}V(\theta)$ and it can be also defined as the average reward per time step in some conditions.

2.4.2 Policy Gradient Theorem

Let $J(\theta)$ be the objective function of the policy, the policy gradient searched a local maximum in $J(\theta)$ by ascending the gradient of the policy with respect to θ.
Theorem 1 (Policy Gradient) *For an MDP with a objective function,*

$$\frac{\partial \rho}{\partial \theta} = \sum_s d^\pi(s) \sum_a \frac{\partial \pi(s, a)}{\partial \theta} Q^\pi(s, a), \tag{2.11}$$

where $d^\pi(s)$ is a discounted weighting of states encountered starting at $s(0)$ and then following $\pi : d^\pi(s) = \sum\limits_{t=0}^{\infty} \lambda^t Pr\{s_t = s|s_0, \pi\}$.
Theorem 2 (Policy Gradient Theorem) *For any differentiable policy $\pi_\theta(s, a)$ with any objective function, the policy gradient is*

$$\nabla_\theta J(\theta) = E_{\pi_\theta}[\nabla_\theta log \pi_\theta(s, a) Q^{\pi_\theta}(s, a)] \tag{2.12}$$

2.4.3 REINFORCE: Monte Carlo Policy Gradient

With policy gradient theorem, the policy function can be updated from the sample of experiences in continuous action space. The most basic policy gradient framework call REINFORCE was introduced in [18]. The algorithm is given in algorithm 3 below.

```
function REINFORCE;
Input : a differentiable policy parameterization $\pi(a|s, \theta)$
Initialize parameter $\theta$ arbitrarily;
for episode $\{s_1, a_1, r_2, \cdots, s_{T-1}, a_{T-1}, r_T\}$ do
    for t = 1 to T do
        $\theta \leftarrow \theta + \alpha \bigtriangledown_\theta \log \pi_\theta(s_t, a_t) v_t$
    end
end
return $\theta$;
```

Algorithm 3: REINFORCE method for policy gradient

Note that the REINFORCE is a Monte Carlo algorithm, which means it uses the complete episode for update. That is, the return from time t, which includes all future rewards up to the end of episode will be used. In this case REINFORCE is defined for episodic case.

2.5 Actor-Critic

Although policy gradient method like REINFORCE can learn policy directly for continuous action space, it is still not sample-efficient and learns slowly for large space. Bootstrapping methods like temporal-difference can eliminate these inconveniences. In order to gain these advantages with policy gradient methods, actor-critic methods with a bootstrapping critic was introduced[14].

Actor-critic method is a combination of Q learning and policy gradients. The actor is the policy for selecting actions in continuous space, and critic is value-based learning, which learns the value function and gives feedback to actor. Actor-critic method enables bootstrapping that could update the actor for every step, instead of updating until the episode ends in policy gradients.

2.5.1 Deep deterministic policy gradient

Despite the advantages of actor-critic methods, the two neural networks involved are being updated in the continuous state and action space every step, and all of the updates are strongly related, which might lead the network hard to learn. To solve this problem, a modified algorithm based on the structure of actor-critic algorithm called Deep deterministic policy gradient (DDPG) was introduced by Google DeepMind[6].

Deep deterministic policy gradient (DDPG) is a combination of Actor-critic method with the concept of DQN, and it learns more efficiently on continuous space. This is due to the fact that for deterministic policy gradient, the expected gradient of Q function can be more efficiently estimated. As an off-policy actor-critic, deterministic policy gradient outperform their stochastic counter-parts in high-dimensional action space[6].

For DDPG, the deterministic policy $a = \mu_\theta(s)$ is used to estimate the fixed actions for each state, rather than stochastic policy $\pi_\theta = P[a|s;\theta]$

2.5.1.1 Gradients of Deterministic Policies

The majority of model-free of reinforcement learning is policy iteration, which includes policy evaluation and policy improvement. Policy evaluation is to estimate action-value function by MC sampling or TD learning. While policy improvement updates the policy with regard to estimated action-value function. However, in continuous action spaces, greedy policy improvement becomes problematic[6]. Instead of globally maximizing Q, policy gradient is used to move the policy in the direction of Q value gradient, shown in equation 2.13.

$$\bigtriangledown_\theta Q^{\mu^k}(s, \mu_\theta(s)) = \bigtriangledown_\theta \mu_\theta(s) \bigtriangledown_a Q^{\mu^k}(s, a) \tag{2.13}$$

While for deterministic policy $\mu_\theta : \mathcal{S} \to \mathcal{A}$ with parameter vector $\theta \in \mathbb{R}^n$, the policy gradient theorem would also be used to update the policy. Since we have the performance objective $J(\mu_\theta) = \mathbb{E}[r_1^\gamma|\mu]$, and the probability distribution $p(s \to s^{'}, t, \mu)$, and discounted state distribution $\rho^\mu(s)$, the performance objective can be expressed as:

$$\begin{aligned} J(\mu_\theta) &= \int_{\mathcal{S}} \rho^\mu(s) r(s, \mu_\theta(s)) \mathrm{d}s \\ &= \mathbb{E}_{S\sim\rho^\mu}[r(s, \mu_\theta(s))] \end{aligned} \tag{2.14}$$

Therefore, the gradient of it can be calculated as:

$$\begin{aligned} \bigtriangledown_\theta J(\mu_\theta) &= \int_{\mathcal{S}} \bigtriangledown_\theta \mu_\theta(s) \bigtriangledown_a Q^\mu(s,a)|_{a=\mu_\theta(s)} \mathrm{d}s \\ &= \mathbb{E}_{S\sim\rho^\mu}[\bigtriangledown_\theta \mu_\theta(s) \bigtriangledown_a Q^\mu(s,a)|_{a=\mu_\theta(s)}] \end{aligned} \tag{2.15}$$

2.5.1.2 Deterministic Actor-Critic Algorithms

The overall algorithm of DDPG is given in [6], which is shown below.

```
DDPG Algorithm;
Randomly initialize actor $\mu(s|\theta^\mu)$ and critic network $Q(s,a|\theta^Q)$ with weights $\theta^\mu$
  and $\theta^Q$;
Initialize target network $\mu'$ and $Q'$ with same weights as learned network;
Initialize replay buffer R;
for episode = 1 to k do
    Initialize a random process N for action exploration;
    Get and save the initial state $s_1$;
    for step = 1 to T do
        Select action $a_t = \mu(s_t|\theta^\mu) + N_t$ according to current policy and exploration
          noise;
        Execute the actions $a_t$ and observe reward $r_t$ and next state $s_{t+1}$;
        Store the transition $(s_t, a_t, r_t, s_{t+1})$ in buffer R;
        Sample a random mini-batch of N transitions $(s_t, a_t, r_t, s_{t+1})$ 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: $L = \frac{1}{N}\sum_l (y_i - Q(s_i, a_i|\theta^Q))^2$;
        Update the actor policy using sampled policy gradient;
        $\bigtriangledown_{\theta^\mu} J = \frac{1}{N}\sum_i \bigtriangledown_a Q(s,a|\theta^Q) \bigtriangledown_{\theta^\mu} \mu(s|\theta^\mu)|s_i$;
        Update the target networks:;
        $\theta^{Q'} = \tau\theta^Q + (1-\tau)\theta^{Q'}$;
        $\theta^{\mu'} = \tau\theta^\mu + (1-\tau)\theta^{\mu'}$
    end
end
```

Algorithm 4: DDPG Algorithm

It is also stated in [6] that the learning may suffer from convergence issues due to both bias introduced by function approximator, and also instabilities caused by off-policy learning. A more principled approach would be using compatible function approximation and gradient TD learning.

Also it is mentioned that the most challenging part of learning in continuous action space is the exploration. However, for DDPG the problem of exploration can be treated independently from the learning algorithm, by adding noise sampled from a noise process N to the actor policy:

$$\mu'(s_t) = \mu(s_t|\theta_t^\mu) + N \tag{2.16}$$

2.6 Hierarchical Reinforcement Learning

For complex tasks like overtaking, which might contains several operations of the vehicle, including changing lanes to left or right, accelerating or braking in certain sequences, it may be less efficient to learn from primitive actions. Option framework can be a better choice since temporal abstraction is key to scaling up learning and planning in reinforcement learning[1].

The concept of temporal abstractions can be understand as some kind of high-level policy, with more abstract actions that could cover multi-steps for low-level actions. Options in reinforcement learning is such kind of temporal abstraction that contains more than one step of action.

The concept of options in reinforcement learning was first proposed by Richard S. Sutton in 1998. Options are closed-loop policies for taking actions over a period of time, and options enable temporally abstract knowledge and action to be included in the reinforcement learning framework in a natural and general way [15]. The option framework is the minimal extension of reinforcement learning framework that allows a general treatment of temporal abstraction.

2.6.1 Options

As mentioned above, options are generalization of primitive actions to include temporally extended courses of action.

A Markovian option ω is a triple $\langle I_\omega, \pi_\omega, \beta_\omega \rangle$ which contains three components: intra-option policy $\pi_\omega : S \times A \rightarrow [0, 1]$; a termination condition (function) $\beta_\omega : \mathrm{S}^+ \rightarrow [0, 1]$; and an initiation set $I_\omega \subseteq S$. The option is available in state s_t if and only if $s_t \in I$. If the option is taken, the primitive actions are taken according to policy π_ω until the option terminates stochastically according to β_ω.

Once an option is selected, an example for state-action trajectory with options can be described as: First, the next action a_t is selected according to probability distribution $\pi_\omega(s_t, \cdot)$. Take the action and observe the next state s_{t+1}, where the option either terminates with probability $\beta_\omega(s_{t+1})$, or else continue with next selected action. When this option terminates, the agent will select another option according to the policy over options.

The termination of options can not only depend on the termination function, sometimes it would be useful for options to "timeout", which means if it has failed to reach any certain state within some period of time, the option has to terminate. This is not possible for Markov options because they are only based on the current state. However, with option framework with semi-Markov options that keeps record of all prior events since an option is initiated, it is possible to do so [15]. Assuming an option is initialized at time t and terminates after steps of k in state s_{t+k}. Within this option, for Markov options at each intermediate time τ, the decisions of intra-policy is dependent only on s_τ, whereas for semi-Markov options, the policy and termination condition may depend on the entire preceding sequence $s_t, a_t, r_{t+1}, ..., r_\tau, s_\tau$, denoted as $h_{t\tau}$, the history from t to τ.

Policies over options $\mu : S \times \Omega \rightarrow [0,1]$ are the policies to choose an option ω based on state s_t according to probability distribution $\mu(s_t, \cdot)$. Note that here even the policy over options is Markov, however, the intra-option policy is semi-Markov because the options are multi-step and the intra-option policy depends on the entire history within that option.

2.6.2 Value functions

Based on the concept of options discussed above, the corresponding value functions for options framework can also be defined.

The value function of a state $s \in S$ under a semi-Markov intra-option policy π_ω as the expected return given the policy is initiated in state s can be expressed as:

$$V^{\pi_\omega}(s) = E\{r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ...|\varepsilon(\pi_\omega, s, t)\} \tag{2.17}$$

where $\varepsilon(\pi_\omega, s, t)$ denotes the intra-option policy initiated in state s at time t. Since the policy over options will have a certain intra-option given a state, the value of general policy μ can be defined as the value of the state under corresponding intra-option policy: $V^\mu(s) = V^{\pi_\omega|\mu}(s)$

The option-value function generalized from action-value function $Q^\mu(s, \omega)$ can be defined as the value of taking option ω in state s under policy μ:

$$Q^\mu(s, \omega) = E\{r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ...|\varepsilon(\omega\mu, s, t)\} \tag{2.18}$$

2.6.3 SMDP methods

As mentioned before, options are related to the decision making problems known as semi-Markov decision process, or SMDP [11]. explain what is semi-MDP[2]:

The theorem of relationship bewteen MDP and SMDP are described in [15] as:
Theorem 3 (MDP+Options=SMDP) *For any MDP, and any set of options defined on that MD, the decision process that selects only among those options, executing each to termination, is an SMDP.*

Based on the relationship between MDPs, options and SMDPs, the methods for planning and learning with options can be developed.

Planning with options need to know the update of state-value function from Bellman equations. For any option ω, let $\varepsilon(\pi_\omega, s, t)$ denote the event of option ω being initiated in state s at time t. Then the reward part of model of ω for any state s is:

$$r_s^\omega = E\{r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... + \gamma^{k-1} r_{t+k} | \varepsilon(\omega, s, t)\} \tag{2.19}$$

where $t + k$ is the time at which this option terminates.
The state-prediction part can be expressed as:

$$p_{ss'}^\omega = \sum_{k=1}^{\infty} p(s', k)\gamma^k \tag{2.20}$$

where $p(s', k)$ is the probability that the option terminates in s' after k steps. Thus, the transmit probability is a combination of the likelihood that option terminates in different steps, which describes the outcome of an option at potentially many different times. This is called a multi-time model [10].

For multi-time models, the Bellman equations for general policies and options can be written as:

$$\begin{aligned} V^\mu(s) &= E\{r_{t+1} + ... + \gamma^{k-1} r_{t+k} + \gamma^k V^\mu(s_{t+k}) | \varepsilon(\mu, s, t)\} \\ &= \sum_{\omega \in \Omega} \mu(s, \omega)[r_s^\omega + \sum_{s'} p_{ss'}^\omega V^\mu(s')] \end{aligned} \tag{2.21}$$

$$\begin{aligned} Q^\mu(s, \omega) &= E\{r_{t+1} + ... + \gamma^{k-1} r_{t+k} + \gamma^k V^\mu(s_{t+k}) | \varepsilon(\omega, s, t)\} \\ &= E\{r_{t+1} + ... + \gamma^{k-1} r_{t+k} + \gamma^k \sum_{\omega' \in \Omega} \mu(s', \omega') Q^\mu(s_{t+k}, \omega) | \varepsilon(\omega, s, t)\} \\ &= r_s^\omega + \sum_{s'} p_{ss'}^\omega \sum_{\omega' \in \Omega} \mu(s', \omega') Q^\mu(s', \omega') \end{aligned} \tag{2.22}$$

Finally, the generalizations of optimal value functions and optimal Bellman equations to options and policies over options can be given. Denote the set of options by Ω, and the set of all policies selecting only from options in Ω by $\Pi(\Omega)$. Then the optimal value function give set of options Ω is:

$$\begin{aligned} V_\Omega^*(s) &= max_{\mu \in \Pi(\Omega)} V^\mu(s) \\ &= max_{\omega \in \Omega_s} E\{r_{t+1} + ... + \gamma^{k-1} r_{t+k} + \gamma^k V_\Omega^*(s_{t+k}) | \varepsilon(\omega, s, t)\} \\ &= max_{\omega \in \Omega_s} [r_s^\omega + \sum_{s'} p_{ss'}^\omega V_\Omega^*(s')] \\ &= max_{\omega \in \Omega_s} E\{r + \gamma^k V_\Omega^*(s') | \varepsilon(\omega, s)\} \end{aligned} \tag{2.23}$$

where $\varepsilon(\omega, s)$ denotes option ω being initiated in state s.
And the Bellman equation for optimal option value is:

$$\begin{aligned} Q_\Omega^* &= max_{\mu \in \Pi(\Omega)} Q^\mu(s, \omega) \\ &= E\{r_{t+1} + ... + \gamma^{k-1} r_{t+k} + \gamma^k max_{\omega' \in \Omega_{s_{t+k}}} Q_\Omega^*(s_{t+k}, \omega') | \varepsilon(\omega, s, t)\} \\ &= r_s^\omega + \sum_{s'} p_{ss'}^\omega max_{\omega' \in \Omega_{s'}} Q_\Omega^*(s', \omega') \\ &= E\{r + \gamma^k max_{\omega' \in \Omega_{s'}} Q_\Omega^*(s', \omega') | \varepsilon(\omega, s)\} \end{aligned} \tag{2.24}$$

With equations given above, the approximations to V_Ω^* or Q_Ω^* become the main goal for planning and learning methods with options.

2.6.4 The Option-Critic Architecture

After the options framework was defined, discovering temporal abstractions autonomously has been the subject of extensive research efforts [1]. With the majority of existing work focusing on finding the subgoals, in [1] a new approach based on policy gradient called Option-Critic architecture was proposed, that is able to perform gradual learning process of intra-option policies, termination functions and the policy over options at the same time.

The process of algorithm is presented below in algorithm 5. The option-level policy is learned with Q learning, while the intra-policy and termination function is updated using policy gradient theorem.

```
Option-critic with tabular intra-option Q-learning;
Initialize the state $s \leftarrow s_0$
Choose an option $\omega$ according to an $\epsilon$-soft policy over options $\mu(s)$
for episode = 1 to k do
    Choose action $a$ according to intra-policy $\pi_{\omega,\theta}(a|s)$;
    Take action $a$ in $s$, observe $s'$ and $r$;
    1. Option evaluation:
    $\delta \leftarrow r - Q_U(s, \omega, a)$
    if $s'$ is non-terminal then
        $\delta \leftarrow r + \gamma(1 - \beta_{\omega,\vartheta}(s'))Q_\Omega(s', \omega) + \gamma\beta_{\omega,\vartheta}(s')\max_{\bar{\omega}} Q_\Omega(s', \bar{\omega})$
    end
    $Q_U(s, \omega, a) \leftarrow Q_U(s, \omega, a) + \alpha\delta$
    2. Options improvement:
    $\theta \leftarrow \theta + \alpha_\theta \, Q_U(s, \omega, a)$
    $\vartheta \leftarrow \vartheta - \alpha_\vartheta \, (Q_\Omega(s', \omega) - V_\Omega(s'))$
    if $\beta_{\omega,\vartheta}$ terminates in $s'$ then
        choose new $\omega$ according to $\epsilon$-soft($\pi_\Omega(s')$)
        $s \leftarrow s'$
    end
end
```

Algorithm 5: Option-critic with tabular intra-option Q-learning

3
Methods

3.1 Environments and Data

3.1.1 Track design

Since driving on the high way is a complex operation, the planning and controlling decisions of a vehicle can vary from case to case. In order to verify that reinforcement learning can be used to complete some autonomous driving tasks including overtaking, a designed track is used in this case to create a simple and limited scenario.

The layout of the designed path can be seen in figure 3.1 below, which consists of two straight tracks with 1000 meters in length and two curves with radius of 100 meters. The track width is 8m and the track is flat.

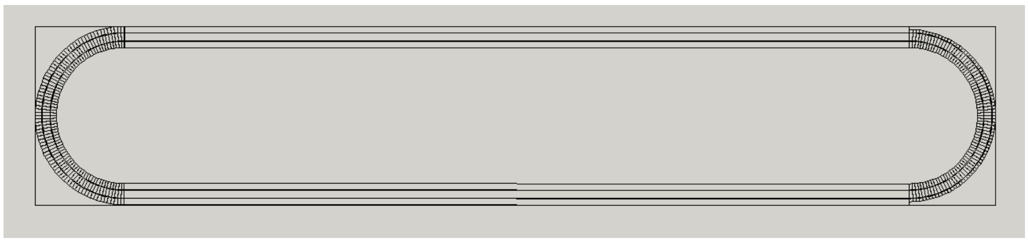


Figure 3.1: Designed track in TORCS track editor

3.1.2 Opponent vehicle set up

In order to simulate the driving environment, several opponent vehicles are added on the track. Since TORCS is a racing car simulator, the opponent vehicles are only allowed to be on the specified start positions in TORCS, thus the initial position of these vehicles cannot be very flexible. As a result, opponent vehicles with different speed are set up, so they can drive on different sides of the track in certain order.

There are in total two different kinds of designed opponent vehicles used during learning. Each kind of vehicle has a specific driving behaviour: the first kind of opponent vehicle("RightFix") is designed to drive on right track with fixed speed; the second kind of opponent vehicle("LeftFix") is designed to drive on left side of track with fixed speed. Depending on the number of vehicles added on the track, the designed highest speed for the same kind of vehicle varies from 40km/h to 92km/h.

As a result, there are many different combinations of opponent vehicles so that different training and testing scenarios can be created.

3.1.3 Traffic Rules constraints

When the self-driving vehicle is driving with certain behaviour (lane following or overtaking), it is essential to obey the traffic rules (traffic light, track lines and speed limit). Since there is no traffic light in TORCS environment, only track line constraints and speed limit will be considered. For the track line constraint it is allowed to go across the dash lines, while solid lines are not allowed to be crossed. The speed limit for self-driving vehicle is set to 120 km/h.

3.1.4 Sensors and variables

In TORCS there are several virtual sensors located on the vehicle. The sensors can be used to get objective-level information of the environment which can be used as the input of the neural network. These inputs include an angle sensor measuring the angle between the car direction and the direction of the track axis, 19 range finder sensors measuring the distance between the track edge and the car with in a range of 200 meters, 36 opponent sensors measuring the distance of the closest opponent in the covered area, 3 speed sensors measuring the speed of the vehicle in X, Y and Z directions, a track position sensor measuring the distance between the vehicle and the track axis, 4 wheel rotational speed sensors measuring the rotational speed of wheels, and 1 engine speed sensor for engine rotational speed. There are in total 65 variables used as input state. The virtual sensors simulate the GPS, Lidar and ultrasonic radar. The input variables from these sensors are listed in table 3.1 below.

Table 3.1: Input variables from virtual sensors

Name	Range (unit)	Description	Sensor
TrackAngle	$[-\pi, +\pi]$(rad)	Angle between vehicle heading direction and track axis	GPS
TrackPos	$(-\infty, +\infty)$	Lateral position of vehicle on the track, 0 when in the middle of track, -1 when on the right edge of track and +1 when on the left edge of the track; smaller than -1 or lager than +1 means vehicle is out of track	GPS
Track	$[0, 8]$(m)	19 sensors, each returns the distance from track edge to the vehicle	Ultrasonic radar
SpeedX	$(-\infty, +\infty)$(km/h)	Longitudinal speed of the vehicle	GPS
SpeedY	$(-\infty, +\infty)$(km/h)	Lateral speed of the vehicle	GPS
SpeedZ	$(-\infty, +\infty)$(km/h)	Vertical speed of the vehicle	GPS
Opponents	$[0, 200]$(m)	Distance from ego vehicle to opponents; a vector of 36 opponents with each vector a span of 10 degrees around the vehicle	Lidar
WheelSpinVel	$[0, +\infty)$(rad/s)	Vector of 4 sensors representing the rotational speed of wheels	Wheel Encoders
Rpm	$[0, +\infty)$(rpm)	Number of rotations per minute of the car engine	N/A

A general figure shows the perception values is given in figure 3.2 below. As can be seen in the figure, the thick dark lines on two sides of the figure are the left and right road boundaries, with a dash line in middle of the figure. 36 arrows around the vehicle indicate the Lidar sensors detection range. The yellow part of the arrows indicates the frontal collision emergency braking detection range, the purple part indicates the side collision detection range on the left side and the right side will be the same. The angles, speedX and speedY are also illustrated in the figure.

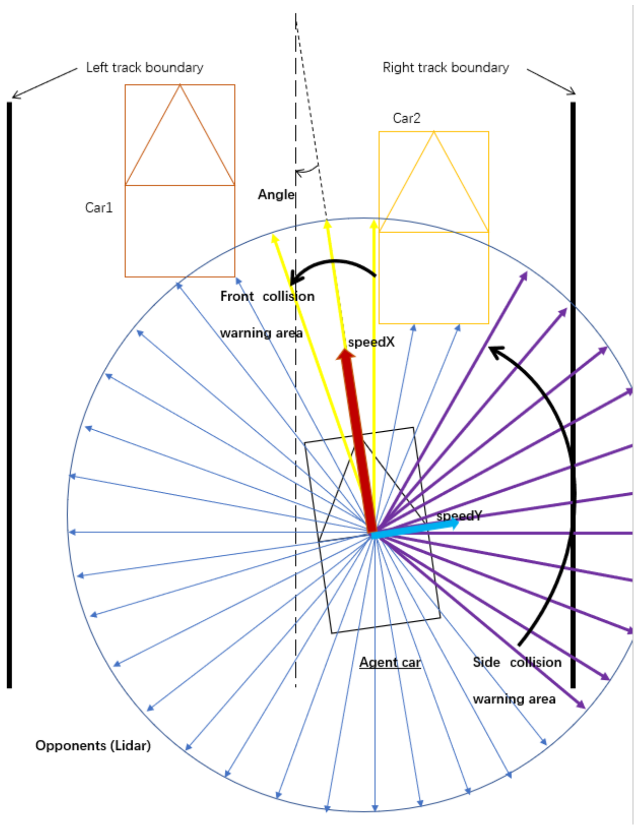


Figure 3.2: Sensors layout

3.2 Planning and Controlling System Design

In order to control the motion of the self-driving vehicle, different path planning strategies are implemented and experimented, including learning of primitive actions and learning of abstract path representation.

Abstract path representation means that the path is represented by abstract variables, for example, a third order polynomial or a combination of target position and speed. The planned path will then be transformed into primitive actions using a low level controller.

However, during the experiments it is found out that learning primitive actions directly will result in very unsmoothed behavior of the agent vehicle, because the learned policy may generate actions that varies a lot between each previous step. Other ways including polynomial representation of path cannot be easily learned by neural network as well. As a result, the learning of target lateral position on track and target speed turns out to be the best solution among all.

After the path is planned, the path representation will be used as the input to the low level controller, which transforms the path into primitive actions. In the end, to guarantee safety during driving, a safety controller is implemented. This is essential due to the fact that there is no guarantee that the learned path can be 100% safe. Although there are penalties for collision and critical situations in the reward function, the value for penalty has to be extremely large to further lower the risk of danger, which will also bring large variance to the learning result. Therefore, inspired by the solution from [12], a safety controller will be added that would actively intervene when critical situation happens.

As is described above, the overall perception, planning and controlling structure are shown in figure 3.3. The path planner, which includes option level policy and intra-policies, receives state information from environment, and outputs the learned abstract path representation which is then transformed into primitive actions and passed through the safety controller for the agent to take actions.

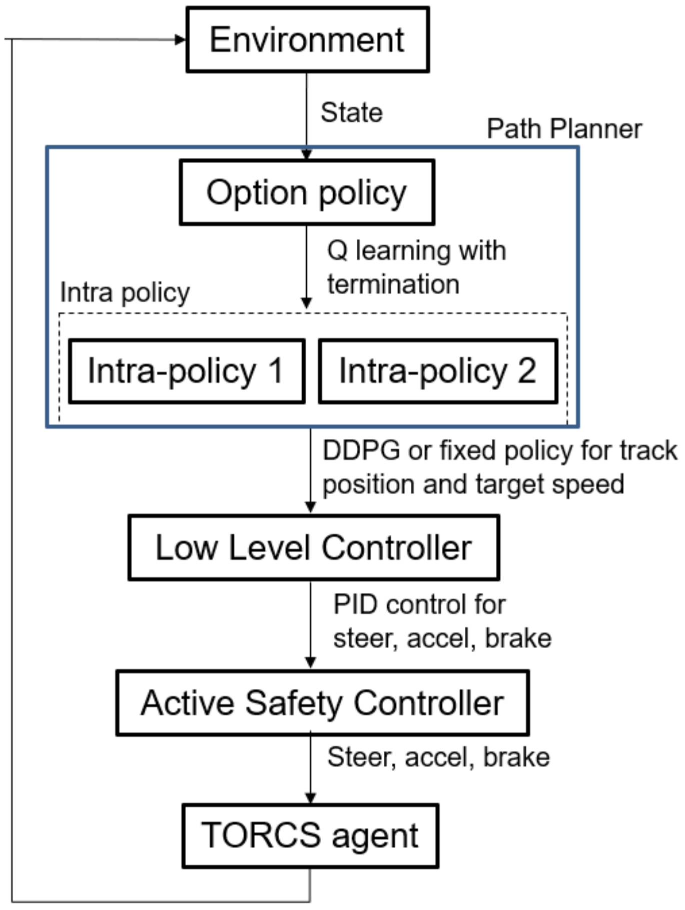


Figure 3.3: The algorithm architecture

3.2.1 Path Planner

As is mentioned above, path planner is the core component of self-driving agent. In this case the planner will learn middle-level of actions: target lateral position on track and target speed.

The learned middle-level actions are listed in table 3.2 below.

Table 3.2: Path-planner-level of actions

Name	Range	Description
Target Lateral Position	$[-1, +1]$	Target track position in lateral direction, -1 means on most left side of the track and +1 means on most right side
Target Speed	$[0, 1]$	Target speed within set minimal and maximal speed range, 0 means minimal speed, and 1 means highest speed

3.2.2 Low Level Controller

The function of the low level controller is to transform the abstract path representation into three primitive actions(steering, acceleration and braking) that can be directly used into TORCS. The variables for these three primitive actions are listed in table 3.3 below.

Table 3.3: Primitive action variables for vehicle controlling

Name	Range	Description
Steering	$[-1, +1]$	Steering wheel angle, -1 and +1 means full right and left respectively
Acceleration	$[0, 1]$	Virtual acceleration value for gas pedal (0 means no acceleration and 1 means full throttle)
Braking	$[0, 1]$	Virtual braking value for brake pedal (0 means no braking and 1 means full braking)

The low level controller are separated into two parts: a lateral controller and a longitudinal controller. They will control the steering and throttle of agent vehicle separately using PID control law.

Before being given to the controller, the target lateral position d_{target} and the target speed v_{target} will be constrained by their boundaries as given in equation 3.1 below:

$$\begin{aligned} d_{target} &\subseteq [-1, 1] \\ v_{target} &\subseteq [0, 1] \end{aligned} \tag{3.1}$$

3.2.2.1 Lateral Controller

The lateral controller will transform target track positions into steering wheel angles according to the current vehicle lateral position using equation 3.2:

$$a_\theta = \tanh(K_p^{\text{angle}}(\text{ob}_{\text{angle}}) - K_p^{\text{position}}(\text{ob}_{\text{trackPos}} - d_{\text{target}})) \tag{3.2}$$

where a_θ is the steering angle shown in table 3.3, ob_{angle} is the angle between the vehicle heading direction and the track middle axis shown in table 3.1, $\text{ob}_{\text{trackPos}}$ is the current lateral position of the vehicle on the track in table 3.1, d_{target} is the target lateral position shown in table 3.2 and K_p^{angle} and K_p^{position} are the coefficients for controller. By tuning the controller in TORCS, the coefficient K_p^{angle} is chosen to be 5 and K_p^{position} is chosen to be 0.5.

3.2.2.2 Longitudinal Controller

The longitudinal controller mainly have two functions: transforming target speed into throttle control signals (acceleration and braking) and decreasing the acceleration when there is risk of sliding.

The inputs of the longitudinal controller are the observations from the environment and the target speed. The first step of the controller is to transform the target speed fraction(in range [0,1]) into the real target speed (in unit of km/h) between the set of minimal and maximal vehicle speed using equation 3.3:

$$s_{\text{target}} = s_{\text{min}} + s_{\text{fraction}} \cdot (s_{\text{max}} - s_{\text{min}}) \tag{3.3}$$

where $s_{fraction}$ is the target speed fraction in range of 0 and 1, s_{target} is real target speed in unit of km/h, s_{min} and s_{max} is the minimal and maximal vehicle speed. s_{min} is set to be 10km/h and s_{max} is 120km/h.

Based on the current vehicle speed, the target speed will be transferred into throttle control signals using a PD controller given in equation 3.4 below:

$$\begin{aligned} a_{\text{acc}} &= \begin{cases} 0, \text{ob}_{\text{speedX}} \geq s_{\text{target}} \\ \tanh(-K_p^{\text{brake}}(s_{\text{target}} - \text{ob}_{\text{speedX}}) - K_d^{\text{brake}} a_{\text{speedX}}), & \text{otherwise} \end{cases} \\ a_{\text{brake}} &= \begin{cases} \tanh(K_p^{\text{acc}}(s_{\text{target}} - \text{ob}_{\text{speedX}}) + K_d^{\text{acc}} a_{\text{speedX}}), & \text{ob}_{\text{speedX}} \geq s_{\text{target}} \\ 0, & \text{otherwise} \end{cases} \end{aligned} \tag{3.4}$$

where a_{acc} is the acceleration control signal, a_{brake} is the braking control signal, $\text{ob}_{\text{speedX}}$ is the the current observed vehicle speed, a_{speedX} is the calculated acceleration of the vehicle, K_p^{brake} and K_p^{acc} are the P controller coefficient for braking and acceleration components which are both set to 0.6, K_d^{brake} and K_d^{acc} are the D controller coefficient for braking and acceleration components which are both set to 0.05.

For the function of traction control, the acceleration will be decreased based on the rotational speed difference between two axles of wheels:

$$\begin{aligned} &\text{if } \ (\text{ob}_{\text{ws[2]}} + \text{ob}_{\text{ws[3]}} - \text{ob}_{\text{ws[0]}} - \text{ob}_{\text{ws[1]}} > \text{threshold}_{\text{TC}}) : \\ &a_{\text{acc}} = a_{\text{acc}} - a_{\text{acc}}^{\text{TC}} \end{aligned} \tag{3.5}$$

where $\text{ob}_{\text{ws[i]}}$ is the observation of the wheel spinning velocity of front wheels and rear wheels, $\text{threshold}_{\text{TC}}$ is the maximum allowed spinning velocity difference, which is set to be 5 rad/s, and $a_{\text{acc}}^{\text{TC}}$ is the decreased part on acceleration, which is set to be 0.2 m/s^2.

3.2.3 Active Safety Controller

The learned policy cannot guarantee 100 percent safety of the vehicle, so active safety controller will intervene at critical situations. Active safety controller will use Lidar sensors as perception system(described in opponents part in observation). Front collision emergency braking system and side collision steering assistance will be the two main functions of the active safety controller.

The front collision emergency braking system will detect the distance to the vehicles in the frontal range, the collision steering assistance will detect the side distance to the opponent vehicles. Commonly, this value for distance representation should be time to collision(TTC), which is the distance between two vehicles divided by relative speed between them. However, it is hard to do object tracking in TORCS environment and the speed of opponents vehicles cannot be accurately estimated due to the low frequency of observation. Therefore, the relative distance will be used to evaluate the critical situation to simplify the case both in the collision emergency braking system and side collision detection system.

For the front collision emergency braking system, the intervention of throttle control signal will be given in equation 3.6, and for the lateral steering assistance, the control law for steering with be give in equation 3.7:

$$\begin{aligned}&\text{if} \;\; \text{ob}_{(opponents[FCRange]} < \text{Safety}_{\text{Long}} : \\ &\begin{cases} a_{\text{acc}} = 0 \\ a_{\text{brake}} = 0.4 \end{cases}\end{aligned} \tag{3.6}$$

$$\begin{aligned}&\text{if} \;\; \text{ob}_{\text{opponents[SCRange]}} < \text{Safety}_{\text{Lat}} : \\ &a_{\text{steer}} = a_{\text{steer}} \pm K_{\text{SC}} \cdot v_x \cdot (1 - \frac{\text{ob}_{\text{opponents[SCRange]}}}{\text{Safety}_{\text{Lat}}})\end{aligned} \tag{3.7}$$

where ob is the observation from environment, a_{acc} indicates the acceleration, a_{brake} indicates braking deceleration, a_{steer} indicates the steering wheel angle, v_x is the longitudinal velocity. $\text{Safety}_{\text{Long}}$ is set to 15 meters, $\text{Safety}_{\text{Lat}}$ is set to 5 meters and steering assistance factor K_{SC} is set to 3.0 in this case.

3.3 Learning Option-level Policy with Fixed Intra-policy using Q-Learning

Based on the scenario set-up and learning structure mentioned above, an important concept of learning a complex task such as autonomous driving is to decompose it into several simpler sub-tasks by using option framework. As the first trial of method, fixed intra-policy for each option is used in this section to evaluate the possibility of learning option-level policy. Deep Q network(DQN) will be used in this section since the number of the output of option-level policy is finite.
For the option framework, another important concept besides policy is the termination function. To simplify the learning process, the learning process of termination

function is performed in another format in our case: the duration of option is not represented by a function approximator, instead, a group of options are assigned with different operation duration. Thus, the learning of termination function is encoded in the learning of selecting options.

3.3.1 Implementation of DQN

The implementation of DQN to learn option-level policy follows the algorithm of DQN given in algorithm 2, the Q network will learn the state-option value which means the agent only learns to choose options. After an option is chosen, the pretrained intra-policies will execute the option.

```
DQN for option learning with fixed intra-policy;
Initialize the TORCS environment and learning agent
Randomly initialize the Q network $Q(s, o|\theta^Q)$ with weights $\theta^Q$
for episode = 1 to k do
    Reset the environment and get initial state $s_1$;
    for step = 1 to done or MAX_EP_STEPS do
        Select an option based on Q-value $o_t = \arg\max_{\theta Q} Q(s, o)$ or randomly select
          an option according to $\epsilon$ -greedy exploration policy;
        for primitive step = 1 to done or current option's duration do
            Get the primitive action $a_t$ using low level controller;
            Execute the primitive action $a_t$, observe the primitive reward
              $r_{\text{primitive}}$and next state $s_{t+1}$;
            Accumulate the discounted reward within this option by
              $r_{\text{option}} = r_{\text{option}} + \gamma^{primitivestep} \cdot r_{\text{primitive}}$;
        end
        Save the latest state $s_{t+1}$;
        Store the transition $(s_t, o_t, r_{\text{option}}, s_{t+1})$ in buffer R;
        Sample a random mini-batch of N transitions $(s_t, a_t, r_t, s_{t+1})$ from R;
        Set $y_i = r_i + \gamma \max Q'(s_{i+1}, a_{i+1}|\theta^{Q'})$;
        Update Q network by minimizing the loss:;
        $L = \frac{1}{N} \sum_l (y_i - Q(s_i, a_i|\theta^Q))^2$;
        Update the target networks:;
        $\theta^{Q'} = \tau\theta^Q + (1 - \tau)\theta^{Q'}$;
    end
end
```

Algorithm 6: Learning option with fixed intra-policy using DQN

The training process begins with the initialization of the TORCS environment and Q network. For each episode, firstly the initial observation will be saved and transformed to the format of neural network's input. Then, for each option-level step, an option is selected using ϵ-greedy exploration and then executed for certain steps or until it terminates. The discounted rewards are accumulated as the reward for this option and the transition is save din the memory buffer. Afterwards, a mini-batch of transition is sampled from the memory randomly and the loss is then minimized by gradient descent. Finally the weights in target network are updated. The overall algorithm is given in algorithm 6 above.

3.3.2 Q network

The neural networks used to approximate Q-value function include a training network and a target network, with exactly the same structure. The inputs of the Q-value networks are the 65 state variables explained in section 3.1.4. The outputs are the state-option values for each option according to the current state. The 4 options designed in our case are:

- option 1: driving on the left track with duration of 15 primitive steps;
- option 2: driving on the left track with duration of 30 primitive steps;
- option 3: driving on the right track with duration of 15 primitive steps;
- option 4: driving on the right track with duration of 30 primitive steps;

The duration for each option are set to be 15 and 30 according to the result of the experiments, it is found out that a smooth lane changing maneuver would take approximately 12 to 18 primitive steps. Therefore, two different duration time will be set for the agent to choice during the learning process, which means the agent would learn to plan the path either for a longer time or a shorter time.

The networks used in this case are all fully connected networks with 2 hidden layers including 200 neurons. The activation function is chosen as "RELU" for each hidden layer and "linear activation" for the output layer. Furthermore, Mean-squared error is used as loss function.

3.3.3 Design of Reward Function

Reward function is one of the most important parts in reinforcement learning. According to the results of experiments, the reward function used in this section consists of the following four components:

1. reward represents target track position: r_{track};
2. reward represents target speed: r_{speed};
3. reward represents critical situation (danger): r_{safety};
4. reward represents collision and running out of the track: r_{damage};

The calculation for each part of the reward function is given as below. As can be seen in equation 3.8, the track position reward penalizes the difference between the vehicle track position and the target track position(in this case the vehicle would keep on the right side of the track). The speed reward increases when the vehicle has a higher longitudinal speed along the track direction and decreases when there is lateral speed. Furthermore, if critical situation(either triggering safety condition boundary or crashing) occurs, the agent will receive a large negative reward.

$$\begin{aligned} r_{\text{track}} &= -v_x|\text{trackPos} - \text{offset}| \\ r_{\text{speed}} &= v_x \cos\theta - |v_x \sin\theta| \\ r_{\text{safety}} &= -1.0 \\ r_{\text{damage}} &= -5.0 \end{aligned} \tag{3.8}$$

The offset is set to 0.5 in the experiment which means that the prioritized driving position is on the right side.

The total reward for each episode is calculated using equation 3.9 below.

$$r = \begin{cases} \gamma_{\text{track}} r_{\text{track}} + \gamma_{\text{speed}} r_{\text{speed}}, \text{if no danger or damage} \\ r_{\text{safety}}(\text{terminate}), \text{if danger but not damage} \\ r_{\text{damage}}(\text{terminate}), \text{if damage} \end{cases} \tag{3.9}$$

where γ_{track} and γ_{speed} are scaling factor for track position reward and speed reward components. In order to balance the value between each part of the reward. They are set to 120 and 240 separately. If critical situation including danger and damage occurs, the current episode will terminate so that the agent can learn from more samples of the safe condition.

3.3.4 Design of Exploration

The balance between exploration and exploitation will have a great effect on learning results. Exploitation is how much the agent act to gain more reward according to its learned policy while exploration means how much the agent will explore the unknown world . In order to avoid local optimal situation and maximize the future reward, the percentage for exploitation and exploration need to be balanced, otherwise the agent can either be stuck in local optimal or be unable to learn anything from its experience. The exploration in DQN is commonly performed by an approach called ϵ-greedy policy which is shown in 3.10. With this method, the agent will try to select random actions for ϵ percentage of time and select actions according to the learned policy in the rest time. The randomness in exploration will help the agent occasionally try some new behaviours in action space. The randomness variable ϵ decays as the option-level steps increases during the entire training process. Equation 3.11 shows the decay of ϵ:

$$o_t = \begin{cases} \arg\max_o Q(s, o), \text{if } \zeta \geq \epsilon \\ \text{random(options)}, \text{otherwise} \end{cases} \tag{3.10}$$

where o_t is the option, $Q(s, o)$ is the option value function, ζ is a randomly generated number between 0 and 1.

Also, the randomness decreases as the number of training episodes increases as is described in equation 3.11:

$$\epsilon = \max(\eta^{i-1}\epsilon_0, \epsilon_{\text{min}}) \tag{3.11}$$

where η is the epsilon decay factor which is selected to be 0.997 in this case, ϵ_0 is the initial epsilon value which is set to be 1.0, ϵ_{min} is the minimal allowed epsilon value which is set to be 0.01 and i is the total option-level steps during the training process.

3.3.5 Update of Target Network

The weights of target network is updated with a frequency lower than the learned network, which is performed using equation 3.12 below.

$$\theta^{Q'} = \tau\theta^{Q} + (1-\tau)\theta^{Q'} \tag{3.12}$$

where τ is the update factor which is set to 0.001 in this case, θ^{Q} is the weight matrix for the learned network and $\theta_{Q'}$ is the weight matrix for the target network.

3.4 Learning intra-policy using DDPG

Although option-level policy can be easily learned using DQN, it is still heavily relay on the hard-coded intra-policy(control law to transform options to primitive actions). In order to get a better and more adaptation for general scenarios, intra-policy within options are considered to be learned directly from the observation using an end-to-end learning strategy. As is shown in figure 3.3, the intra-policy will now be learned with a path planner instead of simply using a fixed controller. Consider the fact that the middle-level action space (target track position and target speed) is continuous, deep deterministic policy gradient is used to learn the primitive action level policy.

In this case, two kind of intra-policies will be learned:

1. Intra-policy 1: Overtaking - The agent vehicle will try to drive at a target speed on the right side of the track. If there is a slower vehicle in front of the agent vehicle, the agent vehicle will try to overtake it from the left side of the track.
2. Intra-policy 2: Following - The agent vehicle will try to drive at a target speed on the right side of the track. If there is a slower vehicles in front of the agent vehicle, agent vehicle will try to slow down and follow the vehicle ahead and keep a relative safe distance.

Based on the same structure, different intra-policies will be learned with different exploration parameters and reward functions.

3.4.1 Implementation of DDPG

The implementation of DDPG for intra-policy learning is shown in algorithm 7 which follows the algorithm description in section 2.5.1 below.

```
Intra-policy learning with DDPG;
Randomly initialize the actor $\mu(s|\theta^{\mu})$ and critic network $Q(s,a|\theta^{Q})$ with weights $\theta^{\mu}$
  and $\theta^{Q}$;
Initialize the target network $\mu'$ and $Q'$ with the same weights as in learned network;
Initialize the replay buffer R with buffer size (100000);
for episode = 1 to k do
    Initialize a random process for the action exploration;
    Reset the environment and get the initial state $s_1$;
    for step = 1 to done or maxSteps do
        Select a target action $a_t = \mu(s_t|\theta^{\mu}) + N_t$ according to the current policy and
          exploration noise (Ornstein-Uhlenbeck process $dx_t = \theta(\mu - x_t)dt + \sigma dW_t$);
        Transform the target action into primitive actions using low level controller
          $a_{\text{primitive}} = f(a_t)$;
        Execute the primitive actions $a_{\text{primitive}}$, observe the reward $r_t$ and next state
          $s_{t+1}$;
        Store the transition $(s_t, a_t, r_t, s_{t+1})$ in buffer R;
        Sample a random mini-batch of N transitions $(s_t, a_t, r_t, s_{t+1})$ from R;
        Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$;
        Update the critic by minimizing the loss:;
        $L = \frac{1}{N}\sum_l (y_i - Q(s_i, a_i|\theta^{Q}))^2$;
        Update the actor policy using sampled policy gradient;
        $\bigtriangledown_{\theta^{\mu}} J = \frac{1}{N}\sum_i \bigtriangledown_a Q(s,a|\theta^{Q}) \bigtriangledown_{\theta^{\mu}} \mu(s|\theta^{\mu})|s_i$;
        Update the target networks:;
        $\theta^{Q'} = \tau\theta^{Q} + (1-\tau)\theta^{Q'}$;
        $\theta^{\mu'} = \tau\theta^{\mu} + (1-\tau)\theta^{\mu'}$
    end
end
```

Algorithm 7: DDPG with primitive action

Similar to learning option policy, the learning process of intra-policy begins with initialization the neural networks and replay memory buffer. Then for each episode, firstly the TORCS environment is reset and an initial observation is obtained. Within the maximum allowed episode steps (note: this is primitive steps here, while for option level learning it is option-level steps), select the target action according to exploration process, and then use a low level controller to transform the target actions into the primitive actions that can be executed by TORCS agent. Afterwards, the observed transition is stored into the memory buffer. In the update process, a random mini-batch of 32 transitions are selected before the actor and critic networks are updated according to the policy gradient theorem.

3.4.2 Structure of Neural Networks

Neural networks are used as function approximators for both actor and critic networks. Keras and tensorflow are used as tools to build up the network model.

3.4.2.1 Actor Network

The actor network is used to approximate the deterministic policy function $\mu(a|s, \theta^{\mu})$. The input layer consists of 65 variables of states and the final layer generates 2 continuous actions: target track position(between -1 and 1) which is a single unit with tanh activation function, and target speed(between 0 and 1) which is also a single unit with sigmoid activation function. The two outputs are combined with a Keras function called Merge which is used to combine variables with different activation functions. The two hidden layers are both fully connected layers with "relu" activation functions with 300 and 600 hidden units respectively. The final layer is initialized with $\mu = 0$ and $\sigma = 10^{-4}$ to make sure that the initial outputs of the policy are not zero.

3.4.2.2 Critic Network

The critic network is used to approximate the action-value function $Q(s, a|\theta^{Q})$, taking both 65 state variables and 2 action variables as input. First of all, the state variables are processed by the first hidden layer with 300 hidden units and "relu" activation function. Secondly, the processed state and action variables are sent into two fully connected layer with 600 hidden units and linear activation functions and the outputs are merged together. Then, the merged result are then processed by the third hidden layer with 600 hidden units and "relu" activation function. In the end, the final fully connected layer with linear activation function generates the value for the state-action pair. The model is trained with Adam optimization and mean-square loss.

3.4.2.3 Target Network

Similar to the update process of DQN networks, the DDPG target network is also used to overcome the unstable during learning process with neural networks. The target networks are exactly the same as the actor and critic networks, used to calculate the target values and updated by slowly track the learned networks using equation 3.13:

$$\theta^{'} = \tau\theta Q + (1 - \tau)\theta^{'} \tag{3.13}$$

where θ is the weight matrix of learned networks, and θ' is the weight matrix of target networks, τ is set to 0.001 in this case.

3.4.3 Design of Reward Function for Overtaking and Following

As mentioned before, one of the advantage of learning intra-policy with DDPG is that different intra-policies can be learned with the same algorithm framework, with only different reward functions. In this case, the main difference between learning overtaking and following is how the reward function is designed.
Since the target behaviour is overtaking and following that is similar as introduced in DQN before, the reward function of the overtaking intra-policy is also the same, consisting of four components:

1. reward regarding target track position: r_{track};
2. reward regarding target speed: r_{speed};
3. reward regarding critical situation (danger): r_{safety};
4. reward regarding collision and running out of the track: r_{damage};

The rewards are calculated with equation 3.9 and equation 3.8. The only difference between the reward functions of overtaking and following is that the coefficient of r_{speed} and r_{track} are both set to 120 when training the following behaviour.

3.4.4 Design of Exploration Behavior

Finding a proper way to perform exploration algorithm in continuous action space is also a critical issue of learning DDPG. However, according to the experiment result in the overtaking learning case, the most commonly used ϵ-greedy exploration is not working very well with primitive action level of learning in TORCS environment because two actions in continuous space will be taken at the same time. It usually doesn't make much sense if the combinations of actions are chosen randomly from uniform random distribution.

Therefore, the exploration are performed by adding noise on actions using Ornstein-Uhlenbeck process introduced in [6] and [5].

3.4.4.1 Ornstein-Uhlenbeck Process

Ornstein-Uhlenbeck process is a stochastic process with mean-reverting properties[19] commonly used for exploration in continuous domain. The Ornstein-Uhlenbeck process can be expressed as the following stochastic differential equation:

$$dx_t = \theta(\mu - x_t)dt + \sigma dW_t \tag{3.14}$$

where W_t represents a standard Brownian motion. θ is the rate of mean reversion which represents how fast the variables revert towards the mean. μ is the long term mean of the process, which represents the mean value of the variable. σ is the volatility or average magnitude per square-root time which represents the volatility degree of the process.

The Ornstein-Uhlenbeck process has a mean-reverting property. When the last random fluctuations part in the process is ignored, it can be seen that x_t has an overall drift trend towards the mean μ, which means the process x_t reverts to this mean at the rate of θ.

3.4.4.2 Exploration for Overtaking

The value for parameters used in OU(Ornstein-Uhlcnbeck) function for overtaking is given in table 3.4 below. For overtaking, the agent vehicle is expected to accelerate towards the target speed, so the mean for target speed is set to be 1.0, while the mean for target position is set to 0.0 to achieve an fair exploration between left and right side of the track.

Table 3.4: Ornstein-Uhlenbeck parameters for Overtaking

Action	θ	μ	σ
Target track position	0.6	0.0	0.3
Target speed	0.9	1.0	0.1

3.4.4.3 Exploration for Following

For training following, it is found out that using OU exploration function cannot reach a good result. Therefore, the ϵ-greedy exploration policy is still used in this case when training the following behaviour. Instead of picking a random action, we generate a random noise and multiply a coefficient with it, adding it on the output of the neural network. The final action to be executed is shown as follows:

$$a = a_{original} + \alpha N \tag{3.15}$$

where N is a random number generated with in the set $[-1, 1]$ for the target position and $[0, 1]$ for the target speed, α is a coefficient to tune the scalar of the noise.

3.5 Learning Option-level policy with learned intra-policy

In this section, the option-level policy learned with DQN and intra-policy learned with DDPG are combined to create the entire learning strategy for the agent. Similar to the previous training process in section 3.3, the option-level policy will still be trained with DQN, while the fixed intra-policy will be replaced with the pre-trained intra-policy in section 3.4. The purpose of implementing the path planning part in this way is to evaluate the possibility of training the agent to learn the entire end-to-end planning process, and be more adaptive to more complex and unseen scenarios.

In this case, it is found out that the structure of Q network in DQN with fixed intra-policy is not able to learn the overall task very well, so the number of hidden layers and hidden units are increased. In this case the Q network contains 3 hidden layers, with 300, 600 and 300 neurons for each hidden layer separately. The other parameters of the networks remain the same.

The number of options in this case is increased to 6 with different duration, which are:

- option 1: overtaking with duration of 10 primitive steps;
- option 2: overtaking duration of 20 primitive steps;
- option 3: overtaking duration of 30 primitive steps;
- option 4: following with duration of 10 primitive steps;
- option 5: following with duration of 20 primitive steps;
- option 6: following with duration of 30 primitive steps;

The reward function is same as in equation 3.9, where:

$$\begin{aligned} r_{\text{track}} &= -v_x|\text{trackPos} - \text{offset}| \\ r_{\text{speed}} &= v_x \cos\theta - |v_x \sin\theta| \\ r_{\text{safety}} &= -5.0 \\ r_{\text{damage}} &= -30.0 \end{aligned} \tag{3.16}$$

Compared with equation 3.8, the negative reward for the safety component is set to -5.0 and for the damage component is set to -30.0 in order to give more penalty to danger behavior. The four components of the reward function are shown in equation 3.16.

The exploration uses ϵ-greedy policy with a decaying factor of 0.997 for DQN.

The implementation of the algorithm is given in algorithm 8, which is basically the same as algorithm 6. The only difference is that after selecting the option from value function, the primitive target actions will be generated from learned intra-policy in the last section.

```
DQN for option learning with learned intra-policy;
Initialize the TORCS environment and agent
Load the model for intra-policies with termination steps
Randomly initialize the Q network $Q(s, o|\theta^Q)$ with weights $\theta^Q$
for episode = 1 to k do
    Reset the environment and get initial state $s_1$;
    for step = 1 to done or MAX_EP_STEPS do
        Select an option based on Q-value $o_t = \arg\max_{\theta Q} Q(s, o)$ or randomly select
          an option according to $\epsilon$ -greedy exploration policy;
        for primitive step = 1 to done or current option's duration do
            Get the target action from learned intra-policy model;
            Get the primitive action $a_t$ using low level controller;
            Execute the primitive action $a_t$, observe the primitive reward
              $r_{\text{primitive}}$and next state $s_{t+1}$;
            Accumulate the discounted reward within this option by
              $r_{\text{option}} = r_{\text{option}} + \gamma^{primitivestep} \cdot r_{\text{primitive}}$;
        end
        Save the latest state $s_{t+1}$;
        Store the transition $(s_t, o_t, r_{\text{option}}, s_{t+1})$ in buffer R;
        Sample a random mini-batch of N transitions $(s_t, a_t, r_t, s_{t+1})$ from R;
        Set $y_i = r_i + \gamma \max Q'(s_{i+1}, a_{i+1}|\theta^{Q'})$;
        Update Q network by minimizing the loss:;
        $L = \frac{1}{N} \sum_l (y_i - Q(s_i, a_i|\theta^Q))^2$;
        Update the target networks:;
        $\theta^{Q'} = \tau\theta^Q + (1 - \tau)\theta^{Q'}$;
    end
end
```

Algorithm 8: Learning option with learned intra-policy using DQN

4

Results

4.1 Learning Option-level Policy with Fixed Intra-policy by Q Learning

The agent is trained for 5000 episodes on the designed track described in section 3.1.1, using ϵ-greedy exploration process with decay rate 0.997. The training process should not last too long so that the overfitting problem can be prevented.

According to the opponent vehicles set up described in section 3.1.2, the opponent vehicles are designed as followed: two "RightFix" vehicles driving on right side of the track, with average speed of 92km/h and 80km/h separately; two "LeftFix" vehicles driving on left side of the track, with average speed of 92km/h and 80km/h separately. The agent vehicle starts in the end position on the right side of the track, figure 4.1 shows the initial queue of all the vehicles.

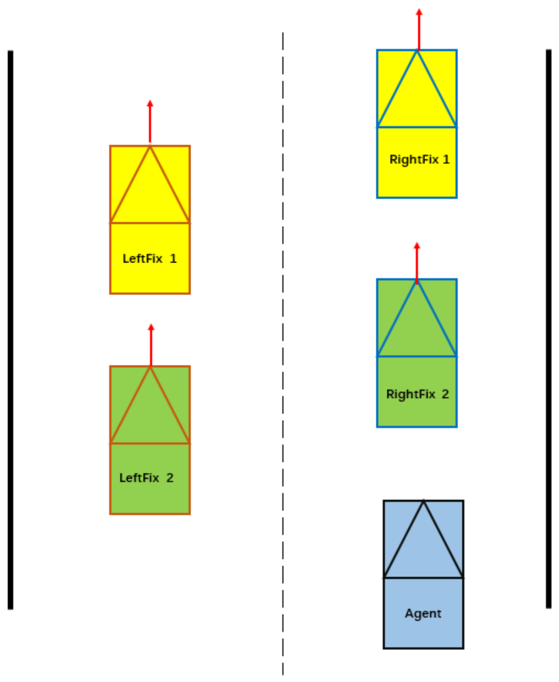
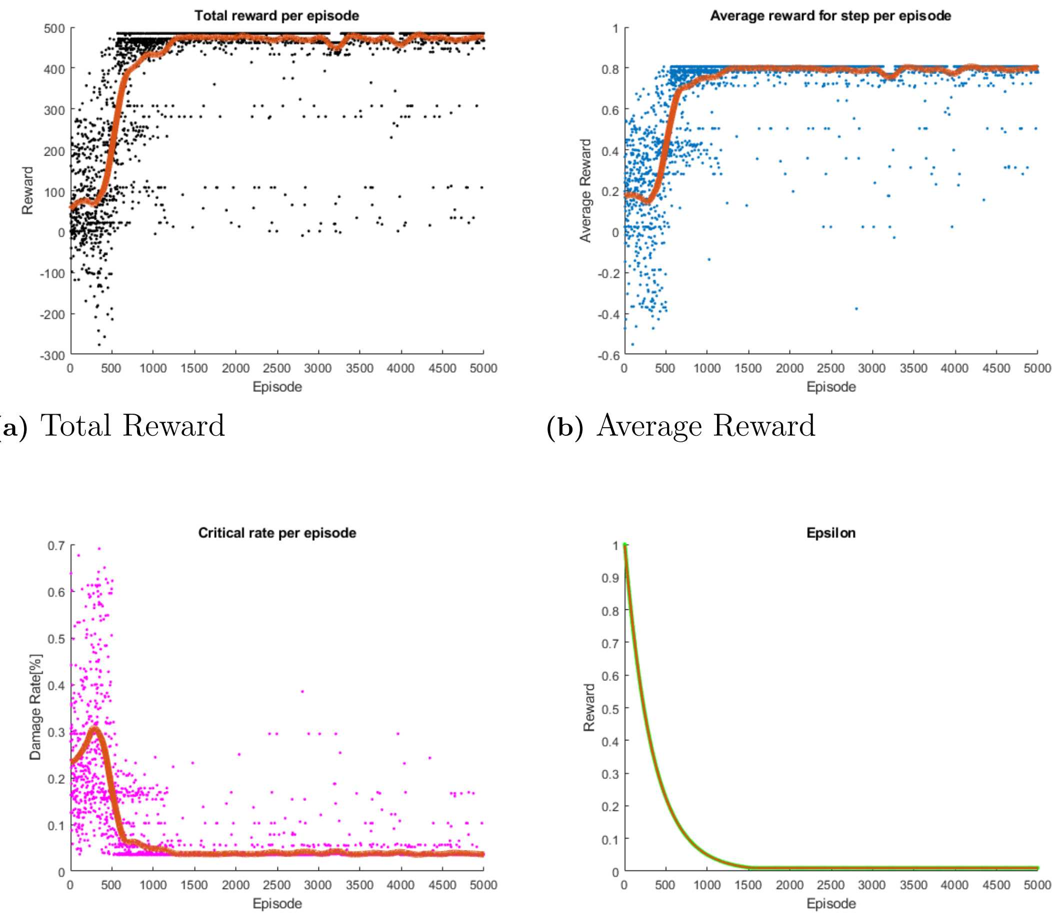


Figure 4.1: Vehicle start positions for option learning with fixed intra-policy

There are several variables reflecting the quality of the training process, figure 4.2 shows the key variables, including accumulated rewards for the entire training episode, average step rewards, rate of critical situations and epsilon.



(a) Total Reward **(b)** Average Reward

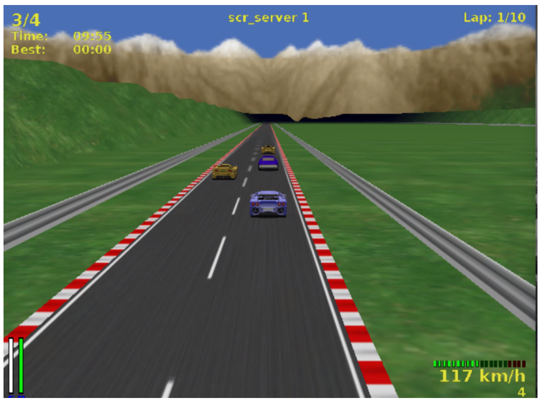
(c) Critical Situation Rate **(d)** Randomness in policy

Figure 4.2: Training process for option learning with fixed intra-policy using DQN

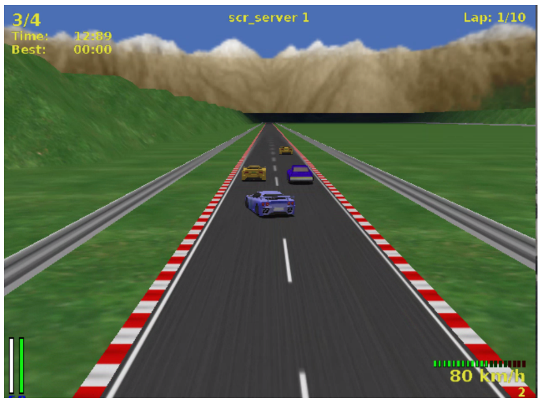
Figure 4.2a shows the total reward per episode, indicting how good the agent performs in the overtaking task, 4.2b shows the average reward per step per episode, which is the more proper variable to reveal the effect of learning indicates the overall performance during the operation time. Figure 4.2c shows the critical situations(collision and in danger of collision) percentage in each episode. Figure 4.2d shows the value of ϵ during exploration, which indicates the randomness rate of actions in each episode.

As can be seen from figure 4.2a and 4.2b, the agent obtains the ability of overtaking after around 1300 episodes, during which the average reward for each episode rapidly raises to approximately 0.8 for most episodes and the number of episode with bad performance(low rewards) decreases. It can also be seen in figure 4.2c that after 1000 episodes, the probability of going into critical situations drops obviously, meanwhile, With the randomness rate decreasing closely to zero, the performance of the agent tends to be stable after 1500 episodes.

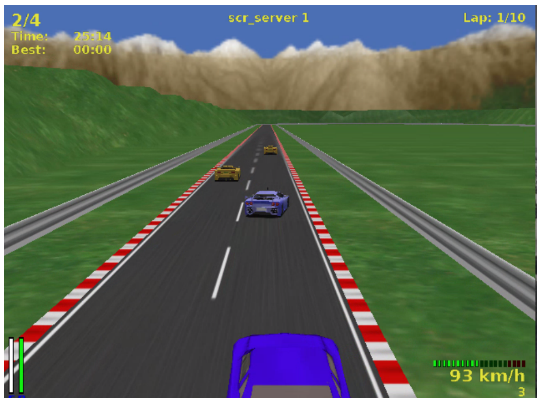
Some typical driving scenarios can be seen in figure 4.3, including following and lane merging.



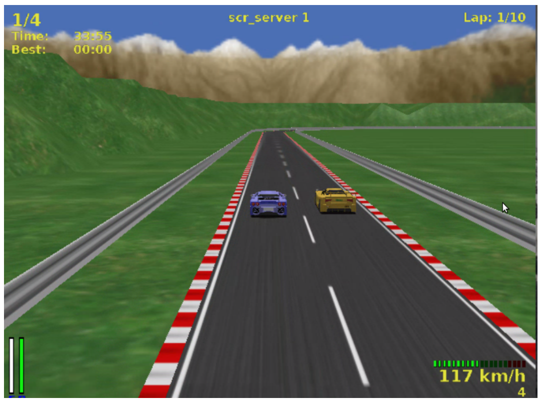
(a) Approaching front vehicle



(b) Merging to left



(c) Merging to right



(d) Overtaking

Figure 4.3: Scenarios of running option policy with fixed intra-policy

After 5000 episodes' training, the learned policy is tested in several environments and the agent can have excellent performance. The test environments are given in table 4.1. It can be seen that the agent successfully learns to drive at the desired maximum speed and executes safe overtaking without collision in all the test cases, with the number of opponent vehicles varying from one to four.

No.	Num. of opponents	Set up	Task Finished	Damage
1	1	RightFix(80[km/h])	Yes	No
2	1	RightFix(92[km/h])	Yes	No
3	2	RightFix(92+80[km/h])	Yes	No
4	3	RightFix(92+80[km/h]) + LeftFix(80[km/h])	Yes	No
5	4	RightFix(92+80[km/h]) + LeftFix(92+80[km/h])	Yes	No

Table 4.1: Tested environment for option learning with fixed intra-policy

Example of the monitored variables for test cases 5 are shown in figure 4.4, where the first sub-figure gives the reward received at each primitive step, the second sub-figure shows the selected options at current primitive step, the third sub-figure shows the vehicle track position (-1 means left and 1 means right) and the last sub-figure shows the speed change. In this test case there are two opponents on the left track and two opponents on the right track so the agent has to execute two times of overtaking to come to the first position. As shown in sub-figure 2, the agent decides to start overtaking at around step 55 and step 150, where the option jumps from 1 to 3 and the track position successfully changes at the same time. The merging back behaviors can also be seen clearly from sub-figure 2 and 3. After overtaking 2 opponents on the right track, the agent continues to drive at almost a constant speed (around 117km/h) on the right track.

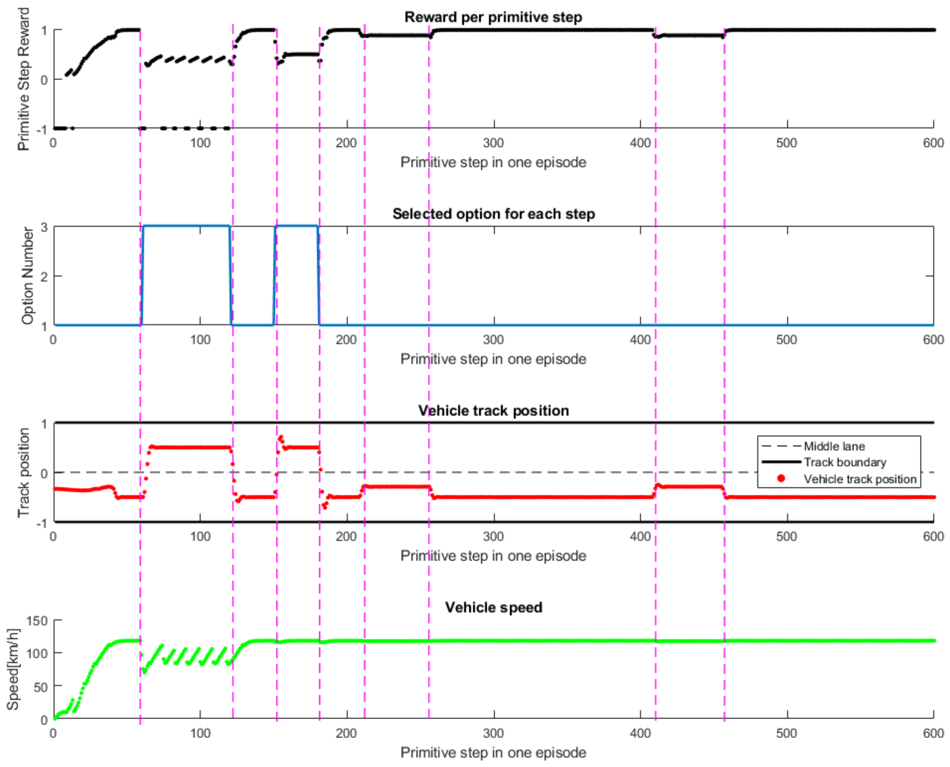


Figure 4.4: Trained results for option learning with fixed intra-policy using DQN

It is noticed that in this test case the agent only selects between option 1 and 3, which means the path planning duration is 30 option-level steps, the shorter planning period with 15 option-level steps are not selected.

The learning of primitive actions is able to perform the overtaking maneuver within 200 episodes of learning. It is obvious that the ego vehicle tends to drive stably on the right side of the track at the target speed and when the opponent vehicle appears ahead with a lower speed than the ego vehicle, the ego vehicle will merge lane to the left side of the track. After taking over the opponent vehicle, the ego vehicle will merge back to the right lane of the track.

4.2 Learning Intra-policy using DDPG

In this section, two intra-policies are trained separately. Since they will be later combined with option-level policy for complex set-up, the training set-up for these two intra-policy is not very complex. Besides, for continuous action space of learning, it is found out that it is extremely difficult to learn a proper policy when setting too many opponent vehicles.

4.2.1 Learning Overtaking

For learning overtaking, the agent is trained for 1800 episodes for each time of training on the designed track described in section 3.1.1 using the Ornstein-Uhlenbeck process for exploration, with decay rate of 0.998 in each episode. The training set-up in this training process includes only two opponent vehicles on the right track in front of the agent: 2 "RightFix" vehicles driving on the right side of the track with average speed 92km/h and 80km/h separately. The agent vehicle starts on the right side of the track in the last position as shown in figure 4.5.

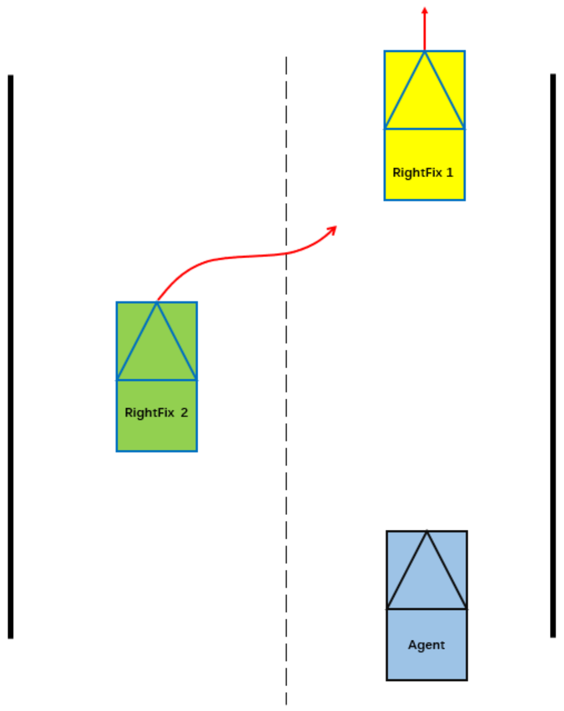
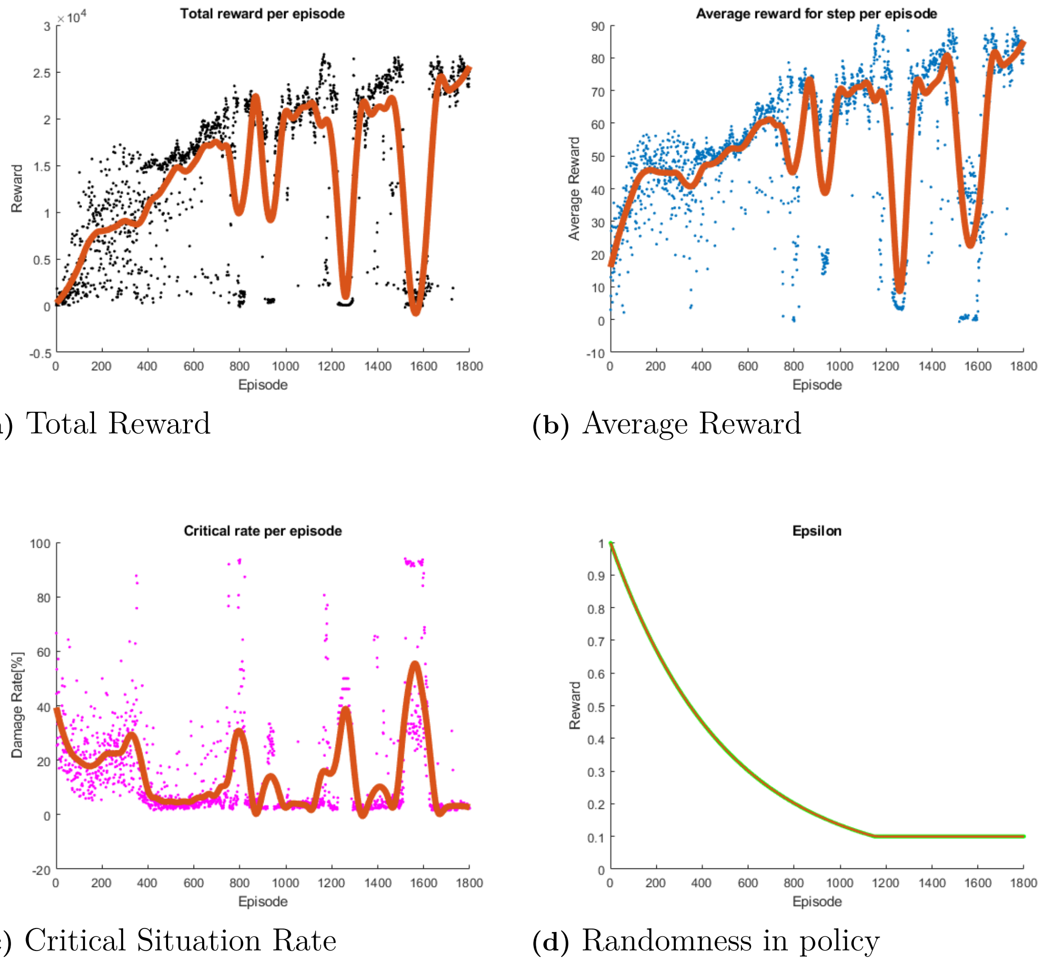


Figure 4.5: Vehicle start positions for learning overtaking

The learning process is shown by monitoring the same variables as in DQN learning process which are given in figure 4.6. As can be seen from figure 4.6a and 4.6b, the agent begins to obtain the lane changing ability after around 300 episodes. Besides, merging back from left to right is much harder to learn then the first lane change from right to left. After 1500 episodes of training, the agent is able to show obvious intention of perform the overtaking maneuver. However, after testing with several experiments and trials with different weighting factors in reward function, it is found out that the learned operation is very hard to be collision free. This is because that the agent shows the intention of performing two times of lane changing, but during the merging process the agent sometimes merges back a little bit earlier and scratches with the opponent vehicles. This is obviously unacceptable during real life driving.



(a) Total Reward **(b)** Average Reward **(c)** Critical Situation Rate **(d)** Randomness in policy

Figure 4.6: Training process for learning overtaking using DDPG

After training, the policy is tested on the environment set-ups given in table 4.2

No.	Num of Oppo.	Set up	Finishing task	Danger	Collision
1	1	RightFix(92[km/h])	Yes	0%	No
2	1	RightFix(80[km/h])	Yes	0%	No
3	2	RightFix(92+80[km/h])	Yes	3.34%	Yes
4	3	RightFix(92+80+62[km/h])	No	3.01%	Yes
5	4	RightFix(92+80+62+57[km/h])	Yes	6.25%	Yes

Table 4.2: Tested setup and results for learning overtaking without safety controller

The testing results shows that the agent can finish most of the cases with a rather low critical situation rate. The danger situation is due to the same situation as in training case, that when the agent vehicle tries to merge back to right track, it always merges a little bit early. Also the start position will have an effect on the damage rate because some critical situations happened in the beginning when the agent is too close to other vehicles. It is also noticed that the agent gets lower rewards at higher speed because the early merging back collision may cause a worse result at high speed.

One example of test cases with four "RightFix" opponent vehicles is shown in figure 4.7. The agent starts on the right side of the track, with two opponent vehicles merging from left to right with speed around 80km/h and another two opponent vehicles driving straight on the right track. The overtaking behavior is shown with dash lines in the figure. The pink line indicates the start of overtaking and the blue line indicates the end of overtaking. In the beginning the agent is blocked by the left vehicle which tries to move to the right track and some collision happened. The agent vehicle changes lane to the left side at around 13 steps and starts to accelerate to 117km/h at 30 steps. After the agent overtakes the third opponent vehicle it merges back to the right side at 40 steps and starts the next overtaking at 45 steps. In the second overtaking process, the agent decides to overtook the two opponent vehicles ahead at the same time. It stays on the left track until around 73 steps and merges back at around 73 steps, at 95 steps, the overtaking is finished after which the agent stays on the right track with a constant speed.

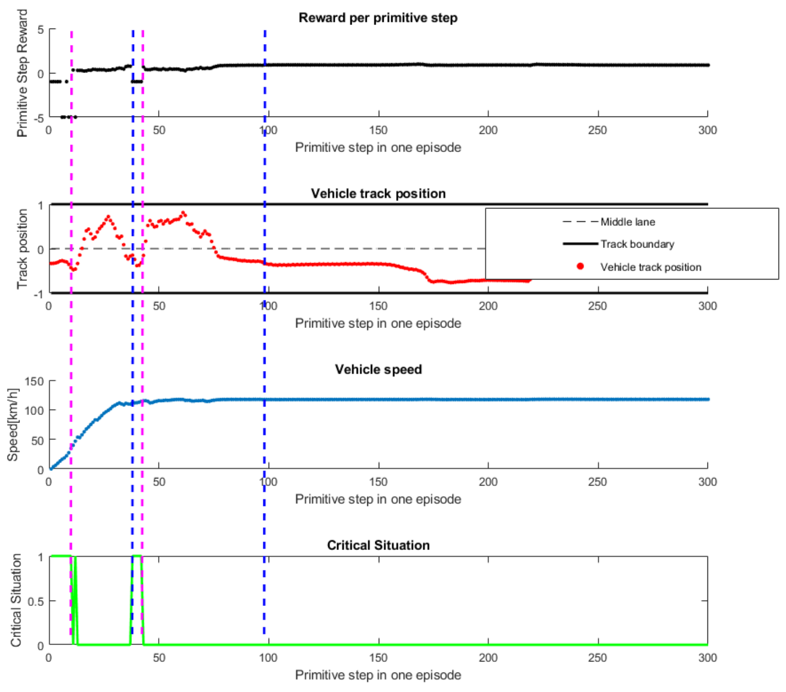


Figure 4.7: Trained results for leaning overtaking without safety controller

To solve the problem of collision during overtaking, a safety controller is activated as the ADAS system to assist the agent, as mentioned in section 3.2.3. With the assistance of safety controller, the agent could successfully avoid the problem of getting too close to the opponent vehicles, which can be seen from table 4.3. The probability of having danger situation is greatly reduced. No collision happens in test cases except in the start position when the agent is very close to the opponents.

The same variables for test environment with four opponents are shown in figure 4.8. Compare with figure 4.7, it is obvious that in sub-figure 1 in figure 4.8, there is no reward of -5 received which means no collision happens. The performance of the agent with safety controller is much better than the original one.

No.	Num. of Oppo.	Set up	Finishing task	Danger	Collision
1	1	RightFix(92[km/h])	Yes	0%	No
2	1	RightFix(80[km/h])	Yes	0%	No
3	2	RightFix(92+80[km/h])	Yes	1.672%	No
4	3	RightFix(92+80+62[km/h])	Yes	2.007%	No
5	4	RightFix(92+80+62+57[km/h])	Yes	5.251%	No

Table 4.3: Tested setup and results for learning overtaking with safety controller

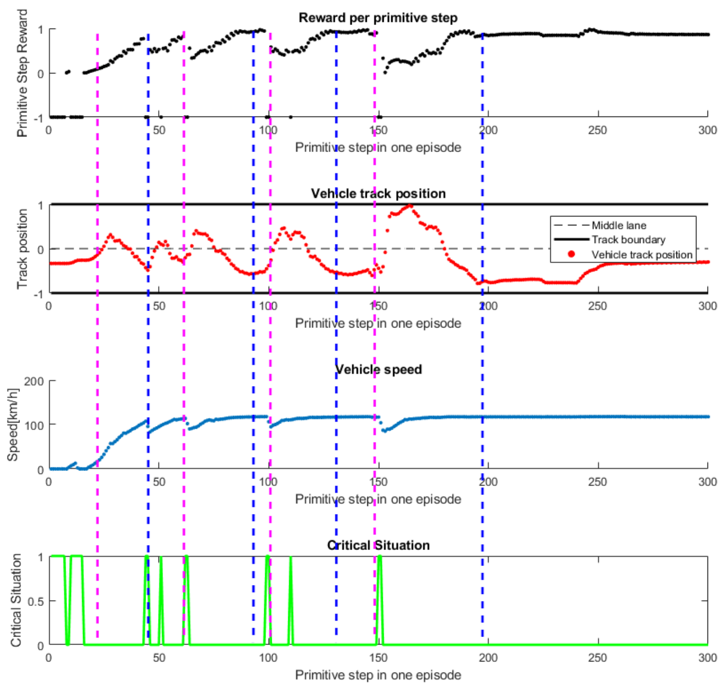


Figure 4.8: Trained results for leaning overtaking with safety controller

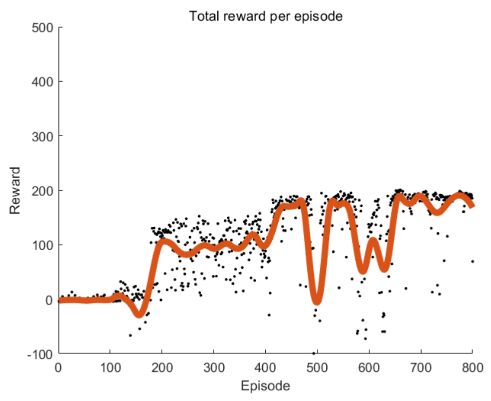
4.2.2 Learning Following

The following behaviour is much easier than overtaking and only after 800 episodes the agent learns this behaviour. The training process is almost the same with training overtaking and the only difference is the exploration policy and reward function.

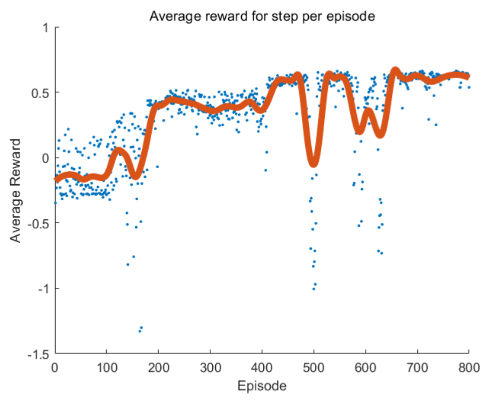
In the following intra-policy, the agent is always supposed to follow the vehicle in front of it no matter which lane the ego car is on, learning to accelerate when the distance is too large and brake when it is too small. The training scenario includes two opponent cars in front of the ego car on the two lanes separately, keeping the same speed and there is only a narrow space between them so the ego car can hardly find room to overtake.

When learning the following behaviour, noise is add to the action to be executed and ϵ-greedy exploration policy is used. The noise added on the target speed is a random number between -1 and 1 which means the right and left edge of the track. Besides,the noise added on the target speed is a random number between -5 and 10 so that the agent can learn to speed up easier, the target speed will remain unchanged if it is a negative number after the noise is added.

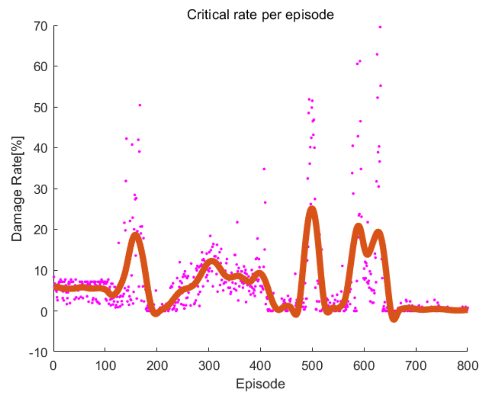
There is also small difference when training the following behaviour, when the distance detection sensor detects that the distance to the vehicle in front of the agent is smaller than 20 meters, the agent will receive a negative reward, the smaller the distance is, the more punishment the agent will get so that the agent will not reach a too close position to the vehicle ahead meanwhile drives as fast as possible on the right side of the track.



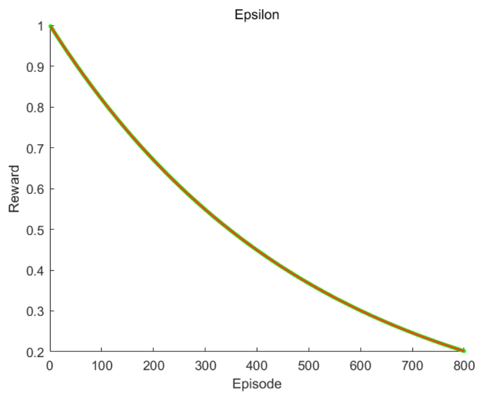
(a) Total Reward



(b) Average Reward



(c) Critical Situation Rate



(d) Randomness in policy

Figure 4.9: Training process for learning following using DDPG

Figure 4.9 shows the training process of the following intra-policy, Figure 4.9a illustrates the total reward of each episode and 4.9b illustrates the average reward per step during training. In the beginning 180 episodes, the agent basically learned nothing and the reward is almost zero. The reward suddenly increased at after 180 episodes but is still unstable as the distribution of the reward is very noisy which means the agent is still doing a wide range of exploration. However, both the total reward and the average reward seem to reach a high and relative stable position after 420 episodes and reach the peak and most stable position after 650 episodes when the agent finally learned the over take behaviour.

Figure 4.9c illustrates the damage rate during training the following behaviour. In the beginning 120 episodes, the damage rate stays at a low standard which is below 9, this is because the agent can not even learn to drive inside the track and kept running out of the track in the beginning and the training terminated before it caused more crash with the opponent vehicles. The agent learned to run inside the track and tried to follow the vehicle in front of it after 120 episodes more collision appeared because the agent could stay longer inside the track before the episode terminated. The critical rate becomes extremely low after 640 episodes and that is exactly when the agent could learn the following behaviour will. However, as there are still some random behaviour during training, collision occurs with a low rate and the simulation test result shows that the agent can follow the vehicle ahead without any collision.

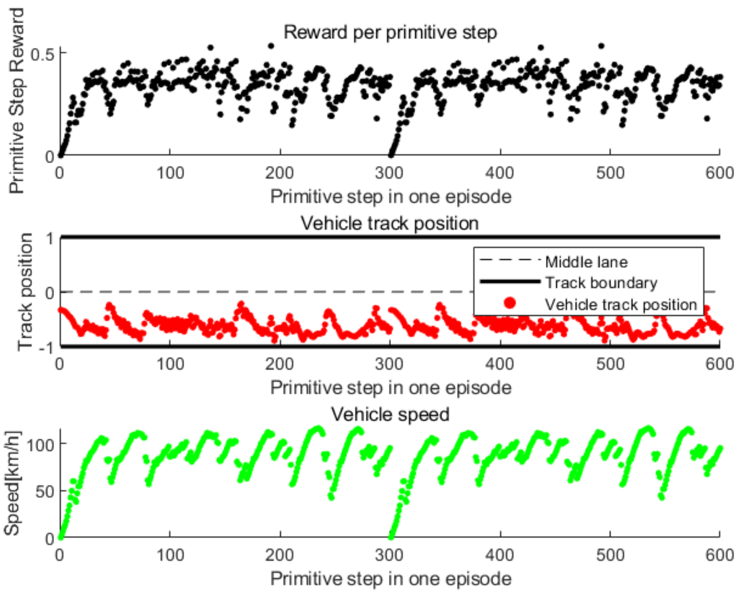


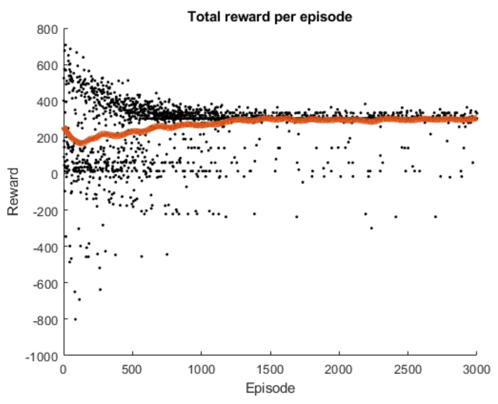
Figure 4.10: Trained results for leaning following without safety controller

Figure 4.10 illustrates the training result of the following behaviour. It can be seen that the primitive reward per step generally has the same tread with speed because the reward function is multiplied with speed. The second sub-figure shows that the agent can stay on the right side of the road, there are some oscillation which shows that the agent keeps adjusting its position. The third sub-figure illustrates that the agent tries to control its speed so that it can keep a proper distance to the vehicle in front of it. The agent keeps its speed around 80 km/h, accelerating when it is too far from the vehicle in front of it an d braking when the distance is too short.

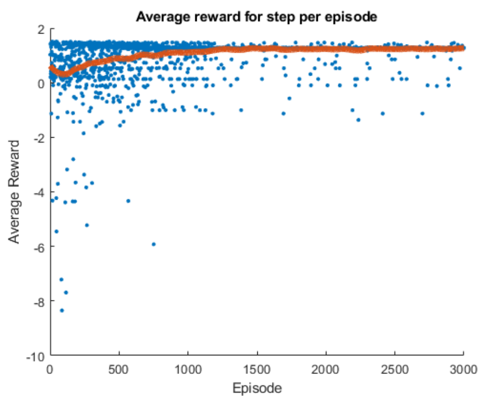
4.3 Learning Option-level Policy with Learned Intra-policy

The training set-up of option level policy with learned intra-policy is the same as learning option with fixed intra-policy. The only difference is that in this case the pre-trained intra-policies using DDPG will be loaded to replace the fixed hard-coded controller.

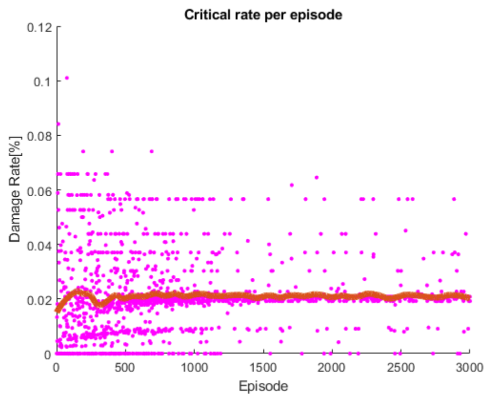
The agent is trained for 3000 episodes with ϵ-greedy exploration policy and the same start position set-up as shown in figure 4.1. The monitored variables during learning process is shown in figure 4.11 below.



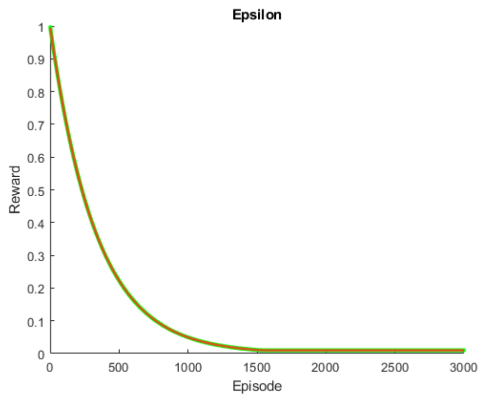
(a) Total Reward



(b) Average Reward



(c) Critical Situation Rate

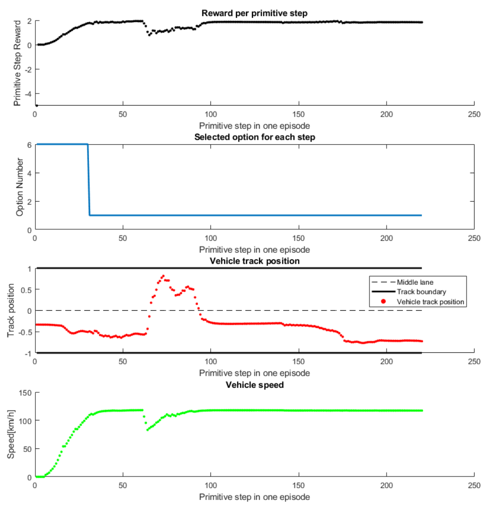


(d) Randomness in policy

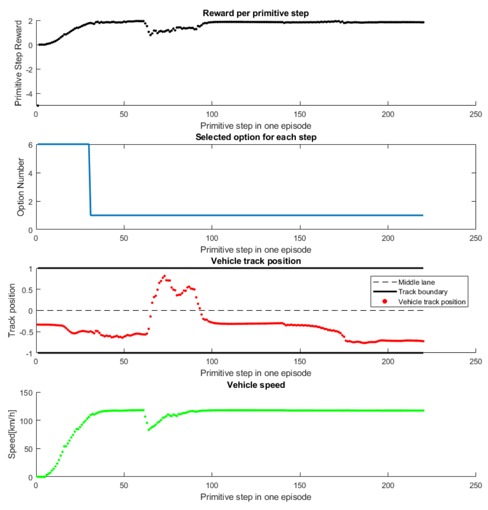
Figure 4.11: Training process for option learning with learned intra-policy using DQN

From figure 4.11a we can see that the total reward in the episode gradually converges to around 400 with a dropping trend from the beginning. However, the learning process can be more clearly seen from figure 4.11b where the average reward per step is showing a converging trend and from figure4.11c where the critical situation rate is getting down. The dropping trend in episodic reward is due to the fact that at beginning of learning, the agent sometimes always try to selecting overtaking with some collision, but high speed from beginning. And that can lead to unexpected high episodic reward.

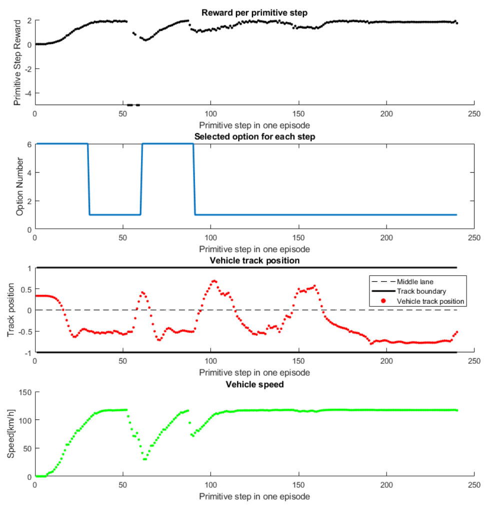
The tested cases are given in table 4.4 with four different combinations of vehicles on both side of the track. To further see the performance of agent vehicle, detailed variables of each test cases are also given in figure 4.12 below.



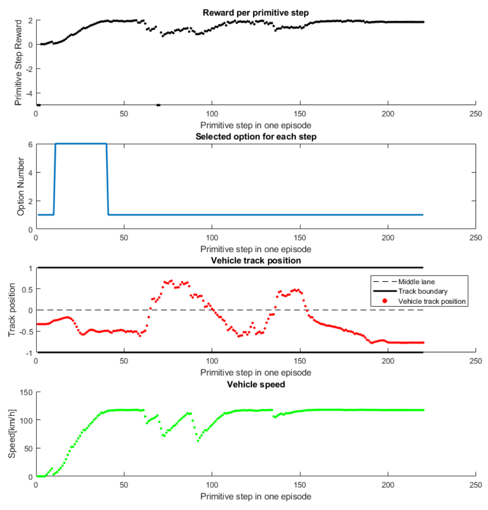
(a) Test case No.1



(b) Test case No.2



(c) Test case No.3



(d) Test case No.4

Figure 4.12: Tested cases key variables for option learning with learned intra-policy

It is obvious that in test case 1 and 2, where there is only one opponent vehicle on right track to overtake, the agent vehicle select overtaking policy and perform one overtaking at proper time. For test case 3 and 4, where there are two opponent vehicles on right track, the agent performs overtaking two times. Noted in figure 4.12c that the agent first tried to overtake but end up being blocked by the vehicle on left track, so it selected following policy and merged back to right again. The overall performance with very low critical situation rate shows the agent has successfully learned to overtake at proper time and avoid collision.

No.	Num. of Opponents	Set up ([km/h])	Task Finished	Danger	Collision
1	2	RightFix(92)+LeftFix(80)	Yes	0.45%	No
2	2	RightFix(92)+LeftFix(92)	Yes	0.91%	No
3	3	RightFix(92+80)+LeftFix(92)	Yes	2.08%	No
4	4	RightFix(92+80)+LeftFix(92+80)	Yes	1.82%	No

Table 4.4: Tested setup and results for learning option with learned intra-policy

5 Conclusion

In this thesis, a complete autonomous driving structure for the self-driving vehicle is developed, that could execute continuous actions. Reinforcement learning algorithms are applied for the leaned policy. DQN is used for learning option level policy and DDPG is used for learning intra-policies. Under several traffic and vehicle dynamic constraints, the learning algorithms learns to perform the autonomous driving tasks efficiently. Several experiments are performed to study the proper combination of reward functions.

The results shows that that the reinforcement learning is able to learn the path planning during self-driving tasks. With the designed learning structure, which is a combination of learned path planner and low level controllers, the agent can perform well in the simulated environment to drive on the high way, with several opponent vehicles on both driving track and side track. That is, the agent learns to plan the path for a certain time period, and the planner is able to make behavior changing decisions, such as lane merging, overtaking and following. It is found out that the DQN is very efficient for learning option level of policy, while DDPG for continuous primitive action level of learning is not very easy to achieve a satisfied performance. Therefore, safety controller is still required in this thesis work to guarantee the safety of agent vehicle.

The learned policy is generalize enough to some environment change, but not tested with much randomness of the opponent vehicles due to the working time and limitation of simulation environment. And the algorithm is still far away from real driving data and real world test.

Overall, the concept of using reinforcement learning to plan the path for a certain period has been verified to be reasonable and work at least in the simulation environment with objective level of data.

6
Future Work

6.1 Improvement on Simulation Environment

Though TORCS is a well-developed simulation environment, it is developed for race cars, which makes the track and environment set-up different from urban driving situation. Based on the learning purpose of this thesis, the simulation environment still has many limitations, including:

- The designed track is for racing vehicles, which contains only one lane actually;
- The opponent vehicles can not be easily placed on any position of the track;
- During training the visualization tool cannot be disabled, which slows down the training process;
- The frequency for all the observation data is same, and for some of the perception data it is not high enough.
- The angular resolution for simulated Lidar data is still not enough for accurate perception.

For future development including using real data, probably a simulator more related to real urban scenario should be used to enable more flexibility in environment setup.

6.2 Extension of Training Scenarios

Due the the limitation of working time and simulator as well, the training and testing scenarios do not vary enough in this thesis. That is, to evaluate and test the learning agent, a much bigger group of testing environment with more randomness should be used, including more randomness behavior of opponent vehicles and different tracks.

6.3 Improvement on Learning Algorithms

6.3.1 Observation Data

The input data for learning agent in this thesis should be objective level data. However for TORCS, some of the input data is still raw data from sensors, which could be further processed before being used for learning. For example the Lidar and Ridar data which detects the opponent vehicles and tracks, could be processed with object tracking algorithms. In this way it should be easier for reinforcement learning agent to learn.

Furthermore, the data from simulation is still different from real world data, that errors and drifts have not be taking into consideration in this thesis.

6.3.2 Algorithm Structure

Although the agent managed to learn the expected behavior, the learning result is still not good enough. Therefore, many hard-coded controllers have to be added to guarantee the safety of agent vehicle. Besides, the DDPG part of learning is not guaranteed to be able to learn every time. Deeper understanding of actor-critic algorithm is still needed to improve the learning algorithm, to make a more stable and efficient learning process.

6.3.3 Low Level Controller

The low level controller in this thesis is designed as a simple PID controller. To achieve better performance, more advanced and well tuned controller can be developed, for example the MPC(Model Prediction Control) controller that is often used in automotive industry.

Bibliography

[1] Pierre-Luc Bacon, Jean Harb, and Doina Precup. "The Option-Critic Architecture." In: *AAAI*. 2017, pp. 1726–1734.

[2] Vlad M Cora. "Model-based active learning in hierarchical policies". PhD thesis. University of British Columbia, 2008.

[3] Han-Hsien Huang and Tsaipei Wang. "Learning overtaking and blocking skills in simulated car racing". In: *Computational Intelligence and Games (CIG), 2015 IEEE Conference on*. IEEE. 2015, pp. 439–445.

[4] Andrej Karpathy. *Deep Reinforcement Learning: Pong from pixels*. 2016. URL: `http://karpathy.github.io/2016/05/31/rl/` (visited on 05/31/2016).

[5] Yan-Pan Lau. "Using keras and deep deterministic policy gradient to play torcs". In: *URL https://yanpanlau. github. io/2016/10/11/Torcs-Keras. html.[cit. 2017-04-17]* 50 (2016).

[6] Timothy P Lillicrap et al. "Continuous control with deep reinforcement learning". In: *arXiv preprint arXiv:1509.02971* (2015).

[7] Daniele Loiacono, Luigi Cardamone, and Pier Luca Lanzi. "Simulated car racing championship: Competition software manual". In: *arXiv preprint arXiv:1304.1672* (2013).

[8] Daniele Loiacono et al. "Learning to overtake in torcs using simple reinforcement learning". In: *Evolutionary Computation (CEC), 2010 IEEE Congress on*. IEEE. 2010, pp. 1–8.

[9] Jan Peters and Stefan Schaal. "Policy gradient methods for robotics". In: *Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on*. IEEE. 2006, pp. 2219–2225.

[10] Doina Precup and Richard S Sutton. "Multi-time models for reinforcement learning". In: *Proceedings of the ICML'97 Workshop on Modelling in Reinforcement Learning*. Citeseer. 1997.

[11] Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[12] Shai Shalev-Shwartz, Shaked Shammah, and Amnon Shashua. "Safe, multi-agent, reinforcement learning for autonomous driving". In: *arXiv preprint arXiv:1610.03295* (2016).

[13] David Silver et al. "Deterministic policy gradient algorithms". In: *ICML*. 2014.

[14] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. Vol. 1. 1. MIT press Cambridge, 1998.

[15] Richard S Sutton, Doina Precup, and Satinder Singh. "Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning". In: *Artificial intelligence* 112.1-2 (1999), pp. 181–211.

[16] Hado Van Hasselt, Arthur Guez, and David Silver. “Deep Reinforcement Learning with Double Q-Learning.” In: *AAAI.* Vol. 2. Phoenix, AZ. 2016, p. 5.

[17] Christopher JCH Watkins and Peter Dayan. “Q-learning”. In: *Machine learning* 8.3-4 (1992), pp. 279–292.

[18] Ronald J Williams. “Simple statistical gradient-following algorithms for connectionist reinforcement learning”. In: *Machine learning* 8.3-4 (1992), pp. 229–256.

[19] Xiaopeng Zong et al. “Obstacle Avoidance for Self-Driving Vehicle with Reinforcement Learning”. In: *SAE International Journal of Passenger Cars-Electronic and Electrical Systems* 11.2017-01-1960 (2017).