Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Automation and Applied Informatics

Kamel Mina Milad Thabit

Steering Model for Autonomous Driving System Using Deep Reinforcement Learning

Diploma Thesis Design

SUPERVISOR

László GRAD-GYENGE

BUDAPEST, 2021

TASK DESCRIPTION

HEAD OF DEPARTMENT

MSc Thesis Task Description

Kamel Mina Milad Thabit
candidate for MSc degree in Computer Engineering

Steering Model for Autonomous Driving System using Deep Reinforcement Learning

An important part of the Autonomous Driving System (ADS) is the steering system which supposed to emulate the behavior of human drivers as a self-driving car controller. This eliminates the need for human engineers to anticipate what is important in an image and to foresee all the necessary rules for safe driving. The most mature machine learning framework that can be put forward to do such task is Deep Reinforcement Learning (DRL) due to its ability to work and interact with virtual simulation environment.

In thesis thesis work, a DRL model is designed based on asynchronous advantage actor-critic (A3C) algorithm. The asynchronous nature of the method enables running multiple simulation threads in parallel which is important given the high sample complexity of deep reinforcement learning. Also, we try to build our model using Deep Deterministic Policy Gradients (DDPG) algorithm to solve the problem of continuous action space. Finally, we try to compare between the two algorithms to see which is better in continuous action control. The objective of the Steering Model is to perform complex tasks such as; lane keeping, lane changing and overtaking.

Tasks to be performed by the student includes:

- Train the agent using the different models and parameters for even more episodes based on the computing power available
- Build, train and test DRL model using different algorithms such as A3C and DDPG that are better in continuous action control.
- Design and implement the functions of the steering Model (lane keeping, lane changing and overtaking).
- Implement suitable reward function that best fit our model and give better performance.
- Add more actions for the agent to impact the environment, like reverse, and brake.
- Evaluating the model performance on unseen drive scenarios.

Supervisor at the department: László Grad-Gyenge, Research Assistant

Budapest, February 2021

Dr. Hassan Charaf
professor
head of department

Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Automation and Applied Informatics

H-1117 Budapest, Magyar Tudósok krt 2. Q.B.207
Tel: +3614632870, Fax: +3614632871
www.aut.bme.hu

Table of Contents

List of Figures

List of Tables

STUDENT DECLARATION

I, **Kamel Mina Milad Thabit**, the undersigned, hereby declare that the present MSc thesis work has been prepared by myself and without any unauthorized help or assistance. Only the specified sources (references, tools, etc.) were used. All parts taken from other sources word by word, or after rephrasing but with identical meaning, were unambiguously identified with explicit reference to the sources utilized.

I authorize the Faculty of Electrical Engineering and Informatics of the Budapest University of Technology and Economics to publish the principal data of the thesis work (author's name, title, abstracts in English and in a second language, year of preparation, supervisor's name, etc.) in a searchable, public, electronic, and online database and to publish the full text of the thesis work on the internal network of the university (this may include access by authenticated outside users). I declare that the submitted hardcopy of the thesis work and its electronic version are identical.

Full text of thesis works classified upon the decision of the Dean will be published after a period of three years.

Budapest, 27 May 2021

...…….
Kamel Mina Milad Thabit

Summary

Self-driving cars are one of the most prominent research areas in our world nowadays. An important part of the Autonomous Driving System (ADS) is the steering system which designed to mimic the behavior of human drivers as a self-driving car controller. This eliminates the need for human engineers to predict what is relevant in an image and to anticipate all the required rules for safe driving. Reinforcement Learning (RL) is a general-purpose framework that is nowadays improving rapidly with the new ideas from deep learning, it is used to learn agents without datasets as in supervised learning, instead by using trial and error, it can teach agents through the environment interaction, and learning from their mistakes. RL is limited for low dimensional input space and only applicable for a small number of states. The evolution of neural networks makes RL more popular in many applications especially in autonomous driving systems as it can process more complex tasks through its input, that is known as Deep Reinforcement Learning (DRL).

In this thesis work, we will try to apply reinforcement learning to train our car in TORCS simulator which is considered as one of the most popular simulation platforms for racing that could be used for training autonomous cars. There are many algorithms of RL that could be applied in autonomous driving like Q-learning, DQN (Deep Q-learning), A3C (asynchronous advantage actor critic). First, we selected Q-learning algorithm as a discrete action algorithm to train our agent, it uses Q-table to store the states and actions Q-Values. As expected, results show that Q-learning suffers from taking discrete actions so, it is not applicable in autonomous driving that need to continuous actions. Secondly, we applied the Deep Deterministic Policy Gradient (DDPG) Algorithm as a continuous action algorithm to solve the problems of Q-learning. DDPG is a reinforcement learning technique that combines both Q-learning and Policy gradients that can learn policies in high-dimensional, continuous action space. It was proven that DDPG has smoother and better performance than Q-learning. We modified the used reward function to get better reward values. The modified model improves the stability and learning time of TORCS. The average reward is increased and the smoothness is greatly improved with the modified reward model.

Chapter 1 : Introduction

Self driving cars have become popular and one of the most important and promising fields of our time. It has been a challenge and inspiration for researchers and engineers throughout the world for decades. It doesn't require human input in order to avoid road collisions and must be aware of the surrounding environment using number of sensors built on the vehicle. Driving a vehicle is a task that demands a high level of expertise, attention and skills from a human driver. With the development in computer vision and neural networks, the progress of autonomous driving cars become faster and intensive research[1]. The car can interact with its surrounding environment through various sensors such as Camera, Lasers, and Radars that installed on the car. These sensors provide the vehicle sufficient information like: the other vehicles on the same track, traffic signs and lights, lane markings, pedestrians, etc.

There are various paradigms in machine learning such as supervised and unsupervised learning paradigms. Supervised learning makes use of labeled input and output data, it "learns" from the training dataset until it can predict the correct label like the relation between the teacher and student.While an unsupervised learning algorithm is used to analyze and cluster unlabeled dataset,it doesn't need to label data. But both of them require to datasets for learning[2]. Among the various paradigms in machine learning, a very notable one is reinforcement learning that doesn't make use of labeled data. Recently, Reinforcement Learning (RL) has been considered as a promising technique to learn driving policy in autonomous vehicle researches.

RL has been designed some powerful algorithms and frameworks to solve real-time issues. The main idea is that similar to a biological agent, an artificial agent, that interacts with its own environment from where can take the states it needs for exploration and return different actions to the environment. Using the experience accumulated, in the form of cumulative rewards, the artificial agent learns how to optimize some actions until it reaches to the target. In the case of RL, the agent tries different actions in order to understand which of them generates the most reward[3]. The Deep Neural Network (DNN) is integrated with Reinforcement Learning that expands the RL applications in various sequential decision-making tasks, this is known as Deep Reinforcement Learning (DRL). DRL becomes popular in autonomous driving systems as it can receive high dimensional state-space and output smooth and continuous actions for its environment.

In recent years (DRL) methods have been utilized in many fields such as autonomous driving[4], games such as Atari games[5] and robotics[6]. DRL has been designed some powerful algorithms and frameworks to solve real-time issues. There have also been some improvements for continuous control Reinforcement Learning methods [7], [8].

There are many simulators used in different scenarios and times which are very powerful in the field of autonomous car driving. The two popular simulators that mostly used in training our agent are CARLA and TORCS simulators. CARLA is an open-source autonomous driving simulator which gives a wide variety of working environment to train a car to drive in urban environment. it gives many objects in the environment such as traffic rules, pedestrian, and other vehicles of different types [9]. The CARLA simulator is built on the client-server architecture. Through socket with the help of API the agent can connect to the server to run the simulation for training. The client sends commands receiving sensor reading as acknowledgement. It is more complex simulator for training as it requires higher computational capabilities. On the other hand, TORCS, The Open Racing Car Simulator, is open, flexible and has a portable interface for AI development [10]. It is multi-platform for car racing simulation that is highly portable. It is used as ordinary car racing game, as AI racing game and as research platform. It is much simpler simulator, it only needs SCR (Simulator Car Racing) plug-in that that provides various game modes, several diverse tracks and car models. TORCS simulator supports the training, testing and validation of the autonomous urban driving system. In this work to train a car for autonomous driving, the TORCS simulator has been used.

In our thesis work, will focus on the application of DRL in the domain of autonomous driving in TORCS simulator. There are different scenarios and algorithms of RL that has been applied in this domain such as Discrete Action and Continuous Action algorithms. We will try to compare between the performances of both of them in order to know the better one based on the vehicle motion on track. We used Q-learning as a discrete action algorithm, and Deep Deterministic Actor Critic (DDPG) as a continuous action algorithm. For the DDPG algorithm we used different reward functions that used to give rewards back to our car after taking possible action. This thesis will provide the results of each algorithm in both figures and videos so that it is easily for the reader to criticize each algorithm. We also provided the videos in both training and testing phases for the two different reward models to see the better performance of the modified reward model.

1.1 Thesis Objectives

The proposed work aims to investigate Reinforcement Learning (RL) methods for autonomous vehicle control. We faced many problems because the Autonomous Driving problem is a very critical and difficult problem. This work depends on Machine Learning field especially Reinforcement Learning category that is based on learning from mistakes during the interaction with the environment.

The thesis will investigate two Reinforcement models called Q-learning that based on Q-Table and Deep Deterministic Policy Gradient (DDPG) that based on actor-critic networks. The two RL-models will also be compared to see which algorithm is better in autonomous driving systems.

Training, simulation, and evaluation will be performed in TORCS environment that replicates the real world. We trained our model on track (CG Speedway number 1) that is a simple track for faster training and contains both straight and curved parts. We tested our models on another track in order to validate our algorithms. We also showed the figures of reward function that is the most important method for better analyzing of our algorithms.

1.2 Thesis outline

This thesis work is divided into six chapters including the Introduction chapter. The following chapters are organized as follows:

Chapter 2 introduces the latest papers and work related in applying RL algorithms in autonomous driving vehicles. It reviews the different methods to learn the car to drive in different environments and under different conditions.

Chapter 3 describes TORCS Simulator which allows us to apply our RL algorithms through its built-in sensors that provide us with the sufficient information about the surrounded environment that helps us to define both states and actions which are essential for solving the MDP problem. It also describes SCR-Plugin that provides various game modes, several diverse tracks and car models also, allow communication between client and server through UDP messages.

Chapter 4 defines the concept of Reinforcement learning (RL) and the History of RL starting from Markov Decision Problem (MDP) to the actor-critic networks. We introduced

policy and value iteration and describe Discrete action algorithms such as Q-learning and Deep Q learning that uses neural networks. The architecture, implementation and Pseudo-code of these algorithm are presented. We also introduced Continuous action algorithms such as A3C and DDPG that are better for Autonomous driving systems as they lead to continuous and smooth actions.

Chapter 5 Introduces Q-Learning as a discrete action algorithm. It defines the process of Q-learning and how we applied Reinforcement Learning on TORCS Simulator through describing the Game Loop in TORCS and the interface between the client and the server for taking actions from the states sent. Finally, we will show some experimental results through the constructed Q-Table and training video that can test the presented algorithm and validate its performance in autonomous driving systems.

Chapter 6 describes DDPG as a continuous action algorithm. It presents the architecture of the algorithm through is actor-critic networks and experience replay memory, it summarizes the steps of the algorithm implementation. Also, it explains the implemented code through the depth of the actor-critic networks and the design of the reward function that makes the car driving faster by modifying the used reward model and compare between the two functions.it also defines ϵ greedy policy as an Exploration Strategy to help the car to explore more and take random action at the beginning of the training. Stochastic braking is also presented to learn the car how to brake. Finally, we show the results of both training and testing at different TORCS tracks for the two reward models. We can also test our mode though the figures of reward function at training and testing phases besides to the steering actions for each model.

Chapter 7 illustrates the thesis conclusion and the future works in the field of Autonomous Driving.

Chapter 2 : Related work

In this chapter, we will review the previous work and experiments that made by the researchers in the field of autonomous cars using different RL algorithms. They used different methods, simulators, and reward functions for better training .

In [11], the authors adapt an actor-critic, model-free algorithm called Deep DPG (DDPG) based on the deterministic policy gradient that is able to run over continuous action spaces. It can solve more than 20 simulated physics challenges, including classic issues such as cart pole swing-up, legged locomotion, dexterous manipulation and car driving by using the same learning algorithm, network architecture and hyper-parameters. The authors show further that the algorithm will learn end-to-end policy for many of these tasks: directly from raw pixel inputs. The work has used significantly lower levels of expertise, a factor of 20 fewer steps than was used by than DQN's learning to identify solutions in the Atari domain. This means DDPG can solve much harder problems than those discussed here in view of more simulation time.

The authors in [12] first use supervised learning to measure the depth of monocular images. The learning algorithm can be trained either on real-world camera images labeled with distance from the ground to near obstacles, or on a training set consisting of synthetic graphics images. The resulting algorithm is able to correctly evaluate the relative depth of obstacles in a scene using monocular vision indicators. Also, in settings with diverse conditions, with complex perceptions, the Graphical Simulator tests demonstrate that model-based RL holds great promise. The proposed work shows that monocular (1D) depth estimates can be used in unstructured outdoor environments with supervised learning.

The paper [13] proposes an approach of training an autonomous vehicle agent based on the use of a Q learning algorithm to learn how to handle rounds in the CARLA simulator in a proper way. The authors design a tangible learning technique in a simulation environment for sequential and automatic decision-making of autonomous vehicles. In order for a roundabout to be navigated using the Q-learning algorithm in a simulation environment safely, the Markov decision-making (MDP) is used for planning the study of the behavior of an autonomous vehicle. Set of naturalistic driving knowledge and a machine learning model algorithm have been used for learning the making of decisions. The designed learning system will learn enough to perform optimum activities and decide if a positive or negative action has been taken by improving it with its specified rewards policy.

In [14], the research demonstrates a deep Q-network that can autonomously drive in TORCS over different environments. It uses reward function that maximizing longitudinal velocity while minimizing transverse velocity and divergence from the track center is used to train the agent and test the model over two validation tracks. The authors expect LSTM networks to be capable of learning superior policies by modeling long-term state dependencies besides to a memory on a particular track to improve performance on subsequent turns and laps. They propose to use LIDAR sensors along with an inertial measurement unit for obstacle distance and velocity measurements. In addition, a coordinate transformation can be applied so that real-world sensor measurements are comparable to simulated ones from TORCS.

Thesis [15] aims to show that it is possible to train an agent with Q-learning to drive around a track in TORCS without crashing, with little prior knowledge and under the conditions of the warm-up phase of the SCR championship which is a plugin to The Open Racing Car Simulator. The researcher describes the Simulated Car Racing Championship including the software framework and some interesting competitors. They chose to loosen the time constraint of the championship's warm-up phase due to the restriction of giving the agent little prior knowledge more interesting than a strictly limited training time.

In [16], the authors proposed a novel reinforcement learning framework with image semantic segmentation network to make the whole paradigm adaptable to reality. Its efficiency is constrained as a result of segmentation. If the segmentation strategies improve, the model's capability would improve. The agent is trained in TORCS which is a car racing simulator. They suggest combining their proposed model with supervised models on real- world data to achieve better results.

Paper [17] introduced a framework and several learning strategies for end-to-end driving without any mediated perception (object recognition, scene understanding). In comparison with prior study, the authors were entirely self-supervised in learning full control (lateral and longitudinal) like hand brake for drifts. The recently proposed reward and learning strategies aid in faster convergence and more stable drive using only RGB image from a forward-facing camera. An Asynchronous Advantage Actor Critic (A3C) framework is employed to learn the car control in a physically and graphically realistic car control game, with the agents running simultaneously on tracks with a variety of road structures (turns, mountains), graphics (seasons, position). The comparative evaluation has shown the value of the learning strategy as we perform over existing approaches despite the significantly more challenging task and the longer training tracks.

The author in paper [18] introduces DRL system for lane keeping assist depending on different categories for the used algorithms in TORCS simulator. The space of possible actions may be discrete or continuous based on the problem domain. The authors use Deep Q-Network Algorithm (DQN) for

the discrete actions control and Deep Deterministic Actor Critic Algorithm (DDAC) for the continuous actions control. Simulation results demonstrate autonomous maneuvering learning in a situation of simple interaction with other vehicles and complex road curvatures. The authors introduced a new area of research which studies the effect of the limited conditions (termination conditions) on the convergence time of learning for the same algorithm. They concluded that the more termination conditions we applied, the slower convergence time to learn.

In [19], the author proposes an improved Deep Deterministic Policy Gradients (DDPG) algorithm to solve the problem of continuous action space, so that the continuous steering angle and acceleration can be obtained as the action space must be continuous which cannot be dealt with by traditional Q-learning. The authors also design a more reasonable path for obstacle avoidance according to the vehicle constraints include inside and outside. The obstacles are divided into two categories which include static obstacles and moving obstacles. Finally, the different sensor data are combined into the vehicle to meet the need for the algorithm details, including the vehicle state and environmental conditions. The algorithm is tested on an open-source vehicle simulator for racing called TORCS. The result demonstrates the effectiveness and robustness of the method.

The authors in [20] mainly focus on continuous steering control as it is impossible to switch among different discrete steering values at short intervals in reality, they first quantify the degree of continuity by formulating the steering smoothness, then they propose an extra reward penalty. Experiments are carried based on DDPG and A3C in TORCS and CARLA simulators. The comparative evaluations in both experiments prove the effectiveness of the newly proposed penalty. Results show that the proposed penalty improves the smoothness of the steering actions in both algorithms.

In [21], the authors presented a distributed approach to perform A3C on TORCS simulator. A popular approach to increase the exploration in reinforcement learning is to implement A3C algorithm, but it requires a multi-core high-performance CPU which can multi-thread environment processes. Experiments are carried using the distributed approach on single-agent and two-agent with dual core Intel CPUs. Distributed approach with a large

number of agents allows the agent to explore further and decreases the training time compared to a single agent running on a single system. The distributed approach has no limits with the number of agents or workers. As the no of workers increase the exploration of the environment increases, making the agent less likely to get stuck in local minimums. But it has problems in latency due to IO operations and communications between the workers. This approach is suitable for clusters with low performance single or dual core CPUs, but it can be replaced by a single high performance multithreaded system.

Finally, the author in [22]applied simple reinforcement learning, namely Q-learning in order to develop an advanced over-taking policy which we integrated into a behavior-based architecture implementing a complete driver for TORCS. The architecture is developed using the Behavior Analysis and Training (BAT) methodology. They focused on two major behavioral overtaking: (i) the overtaking of a fast opponent either on a straight stretch or on a wide curve; (ii) the overtaking on a narrow bend, which usually demands a more advanced braking strategy. Q-learning can produce competitive overtaking behaviors which can outperform the strategy employed by one of the best drivers available in TORCS distribution. It can also effectively learn complex patterns and outperform programmed NPCs.

Chapter 3 : TORCS simulator

In order to train our agent, it is important to use an environment in which the agent can learn and acquire information needed for training in reinforcement learning. Simulators are the recent developments while training a car agent how to run in an environment. They are very efficient to collect huge amount of data through its different built-in sensors such as cameras, LIDAR, and RADAR sensors. Simulators provide real environments that contains town maps, racing tracks, urban and rural environments. In addition, they provide many objects such as pedestrians, bicycles and others driving vehicles of different shapes and types. Simulators can help cars in learning easily to detect these kind of objects, so they become very popular to train autonomous cars how to drive in different environments. There are many different simulators which are very powerful in autonomous car driving. CARLA and TORCS simulators are the two famous simulators for training the autonomous cars.

In this chapter, we will discuss in detail the TORCS simulator, the reasons for selecting TORCS among the other simulators to train our agent, and how we can apply Reinforcement Learning algorithms on it using SCR plug-in. We will describe the available sensors that used to feed our neural network for training process. Finally, we will show available control actions that may our agent take in its environment.

3.1 TORCS

TORCS: The open Racing Car Simulator is an open-source 3D racing simulation game. It is widely used to train a car how-to run-in racing tracks and similar kind of environments. It is used as a highly portable multi-platform car racing simulation [10]. It is considered as a game engine for a car that is designed as a racing game. Figure 3.1 is a screenshot of TORCS environment that shows the agent under training and other cars of the racing.

3.1.1 Why TORCS?

- It is possible to visualize and examine the learning process of neural networks instead of just looking at the final results.
- It is easy to see that the neural network falls into the local minimum.
- TORCS can help us simulate and understand machine learning technique in automated driving, which is important for self-driving car technologies.

- The AI can learn how to drive.

Figure 3.1 TORCS Environment

3.1.2 SCR-Plugin

Simulator Car Racing (SCR) is a highly customizable plug-in added on TORCS that provides a sophisticated physics engine, 3D Graphics, various game modes, several diverse tracks, and car models. SCR plug-in ensures that all cars in TORCS have the access to all information in either the environment or the tracks. The architecture of TORCS-SCR is composed of two main parts: Server and Client as shown below.

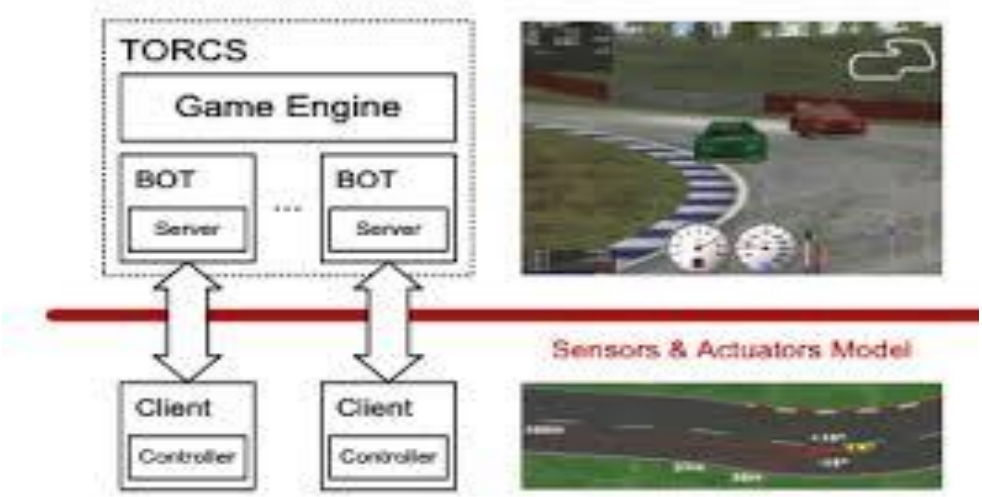

Figure 3.2 TORCS-SCR Architecture [15]

The server acts as a proxy for the environment and the client controls a single car. The controllers operate as external programs and communicate via UDP connections with the custom TORCS version. The server sends the client the available sensory input such as speed, track position, and angle. In return, it receives the optimal output of the actuators. This isolates the controller from its surroundings so that it can be handled as a self-contained agent.

3.1.3 Gym TORCS

Gym-TORCS [23] is a reinforcement learning environment in the TORCS domain. It is made to have an interface which matched Open-AI which is a toolkit for developing and comparing reinforcement learning algorithms as Gym does not have the environment set for TORCS. The process begins from building the environment, defining rewards, and then training the agent through Reinforcement Learning. There are three steps to have this agent running. Server for TORCS, Client for TORCS, and an environment, built like Gym environments, that provides observations and rewards based on the agent.

3.1.4 TORCS Sensors

There are 18 different types of sensor inputs in TORCS. For some values, we normalize and then feed them into the neural network. I got useful input after trial and error:

nsor name	Range (Unit)	Description
ob.angle	[-π,+π]	Angle between car direction and road axis direction.
Gear	-1,0,1,2,3,4,5,6	Current Gear: -1: Reverse 0: neutral 1-6: Various Gear Values
ob.track	(0, 200)(meters)	A vector of 19 rangefinder sensors, each sensor returns the distance between the car and the road edge within 200 meters.
ob.trackPos	(-∞,+∞)	The distance between the car and the road axis. This value is normalized by the road width: 0 means the car is on the center axis, and greater than 1 or less than -1 means that the car has run off the road.

ob.speedX	(-∞,+∞)(km/h)	Vehicle velocity along the longitudinal axis of the vehicle (good velocity)
ob.speedY	(-∞,+∞)(km/h)	Vehicle speed along the transverse axis of the vehicle.
ob.speedZ	(-∞,+∞)(km/h)	Vehicle speed along the Z-axis of the vehicle
ob.wheel SpinVel	(0,+∞)(rad/s)	A vector of 4 sensors, representing the rotation speed of the wheel.
ob.rpm	(0,+∞)(rpm)	RPM of car engine

Table 3.1 TORCS Sensors [34]

3.1.5 TORCS Control Actions

The most common control actions, that we can use in training our agent, are shown in the following table:

Action	Description
Accel	Virtual gas pedal 0: No gas 1 : full gas
Brake	Virtual brake pedal 0: No brake 1: full brake
Gear	Gear value which can be controlled based on the RPM of the car.
Steer	Steering value: -1: full left +1: full right
Meta	This is meta-control command: 0:do nothing 1:ask competition server to restart the race

Table 3.2 TORCS Control Actions [34]

Chapter 4 :Reinforcement Learning Algorithms on TORCS

This chapter describes the foundation of reinforcement learning and some basic algorithms that we could apply in autonomous driving systems in TORCS simulator. We first introduce Markov Decision Process (MDP), policy iteration, and value iteration. Then we go to describe the discrete action algorithms such as Q-learning and Deep-Q Network (DQN) besides to Continuous action algorithm that could be used in our autonomous vehicles such as A3C and DDPG.

4.1 Traditional Reinforcement Learning

Reinforcement learning (RL)[3] is the branch of machine learning that deals with solving sequential decision-making processes. In a general RL model, an RL problem consists of an agent interacts with the environment to learn how to behave in an environment without having any prior knowledge. At every time step, the agent observes the environment state then the agent takes an action based on the observation and its policy. After taking an action, RL agent receives a new observation and reward from the environment about the performance of its action as shown in Figure 4.1. Using this reward, it iteratively updates its action policy to reach to an optimum control policy.

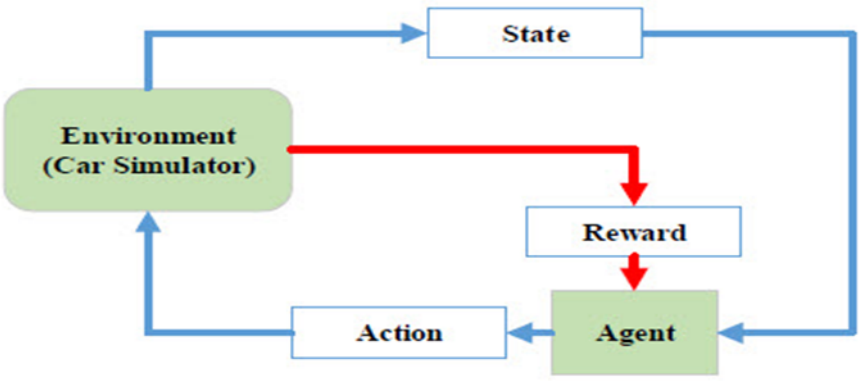

Figure 4.1 Reinforcement learning process [20]

A DRL agent (car) aims to discover optimal action policy through trial and error according to feedback of environment. After carrying out an action and entering a new state, the agent receives a reward from the environment based on how good or bad the action is. The goal of RL is to maximize the discounted accumulative reward.

4.1.1 Markov Decision Process (MDP)

The Markov decision processes (MDPs)[24] is a mathematical framework that describes any RL fully observed environment. It is most commonly used in optimization problems solved by Reinforcement Learning. Markov process has the property that the next state only has relation with the current state and has no relation with the earlier states. Hence the future outcomes are independent of the past given the present state.

MDP is formulated as a tuple consists of 5 main parameters

- A set of states, s ∈ S, where S denotes the state space.
- A set of actions, a ∈ A, where A denotes the action space.
- A transition probability function, P (s'| s, a), where s, s' ∈ S and a ∈ A . it represents the probability of observing state s' after executing action (a) at state (s).
- A reward function, R (s, a, s'), where R(s, a, s') = E[r_t| S_t =s, A_t = a,S_{t+1} = s'].
- Discount factor, denoted by γ, γ ∈ [0, 1]

The main goal of DRL is to find an optimal policy π^* that achieves the maximum expected reward:

$$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k} \tag{4.1}$$

Where γ is the discount factor that prevents the total reward from increasing to infinity. Choosing a greater γ value between 0 and 1 indicates that agent's actions are more dependent on future reward. On the other hand, a smaller γ value results in actions that mostly concerned with the instantaneous reward r_t.

When executing an action, we can't precisely know the future states. So, we must take all the possible scenario into account and expect the long-term returns based on the state transition probability. We would therefore introduce the value-iteration and policy-iteration algorithms which are two fundamental methods for solving MDPs. Both value-iteration and policy-iteration assume that the agent is familiar with the MDP model of the world (i.e. the agent knows both the state-transition and reward functions). Consequently, they can be used by the agent to (offline) plan its actions provided information about the environment before interacting with it.

The value function expresses how good is a state for an agent to be in. It is equal to predicted total reward for an agent starting from state (s). The value function is determined by the policy by which the agent chooses which actions to execute. Value function can represent the expectation of the long-term returns, which are classified into two categories:

(1) state value function $V^{\pi}(s)$: It represents the maximum expected total reward from the current state (s) under the policy π.

$$V^{\pi}(s) = E[\sum_{k=0}^{\infty} \gamma^{k} r_{t+k} | S_t = s] \tag{4.2}$$

There is an optimal value function of all possible value functions that has a higher value than other functions for all states.

$$V^{*}(s) = \max_{\pi}[V^{\pi}(S)] \tag{4.3}$$

The optimal policy **π*** is the policy that corresponds to optimal value function:

$$\pi^{*} = arg \max_{\pi}[V^{\pi}(S)] \tag{4.4}$$

(2) state-action value function $Q^{\pi}(s, a)$: It represents the expected total reward received by an agent after executing an action (a) at the current state (s).

We can define the optimal policy as follows:

$$V^{*}(s) = \max_{a}[Q^{*}(s, a)] \tag{4.5}$$

or

$$\pi^{*}(s) = arg \max_{a}[Q^{*}(s, a)] \tag{4.6}$$

Using the above optimal set of equations, it follows that these two can be expressed in the recursive form yielding the Bellman Optimality Equations (Eq. 3.7).

$$V^*(S) = \max_a \left[\mathrm{R(s,a)} + \gamma \sum_{s'} \mathrm{P}\,(s' | \,\mathrm{s,a}) V^*(s') \right]$$

$$Q^*(s,a) = \mathrm{R(s,a)} + \gamma \sum_{s'} \mathrm{P}\,(s' | \,\mathrm{s,a}) V^*(s') \tag{4.7}$$

Bellman equation is a super-important equation optimization and forms the basis for most reinforcement learning algorithms.

4.1.2 Value Iteration

Value iteration [24] computes the optimal state value function by iteratively improving the estimate of V(s). For each state, it calculates the state value V(s) after executing every possible action and chooses the maximum value as the state value of this state. This process is repeated until all the state values converge. After all the state values converge, an action which has the maximum state- action value is chosen for any state. It can be explained in the algorithm presented in Algorithm 4.1.

Initialize $V(s)$ to arbitrary values
Repeat
  For all $s \in S$
    For all $a \in \mathcal{A}$
      $Q(s,a) \leftarrow E[r|s,a] + \gamma \sum_{s' \in S} P(s'|s,a) V(s')$
    $V(s) \leftarrow \max_a Q(s,a)$
Until $V(s)$ converge

Algorithm 4.1 Pseudo code for value-iteration algorithm [30]

4.1.3 Policy Iteration

Since the agent only concerned with the finding the optimal policy, the optimal policy may often converge before the value function. Instead of using the value iteration to find the optimal value function, policy iteration directly manipulates the agent's policy. Rather than repeatedly improving the value-function estimate, it will re-define the policy at each step and compute the value based on this new policy until the policy converges. Policy iteration is also ensured to converge to the optimal policy, and it often takes fewer iterations to converge than the value-iteration algorithm. Value iteration selects the action according to the state value,

while policy iteration selects the action according to the current policy. It can be explained in the algorithm presented in Algorithm 4.2.

Initialize a policy π' arbitrarily
Repeat
    $\pi \leftarrow \pi'$
    Compute the values using π by
        solving the linear equations
            $V^{\pi}(s) = E[r|s, \pi(s)] + \gamma \sum_{s' \in S} P(s'|s, \pi(s)) V^{\pi}(s')$
    Improve the policy at each state
        $\pi'(s) \leftarrow \arg\max_a (E[r|s, a] + \gamma \sum_{s' \in S} P(s'|s, a) V^{\pi}(s'))$
Until $\pi = \pi'$

Algorithm 4.2 Pseudo code for policy-iteration algorithm [30]

4.1.4 Q-Learning

Value-iteration and policy-iteration algorithms require the state transition probability of the environment which might not always be available in many practical problems. Thus, model-free methods are proposed to solve these problems. One of the most famous of these model-free algorithms is Q-learning. Q-learning [25] is a value-based Reinforcement Learning algorithm that is used to find the optimal action-selection policy. It is a model-free learning environment that can be used in a situation where the agent initially knows only that are the possible states and actions but doesn't know the state-transition and reward probability functions.

The basic idea of Q-learning is to keep a lookup table which rows represents the states and columns represents the actions, the initial values of the table are randomly chosen. In order to learn the optimal Q-value function, the Q-learning algorithm employs the Bellman equation for the Q-value function:

$$New\ Q(s,a) = Q(s,a) + \alpha[R(s,a) + \gamma\ maxQ'(s',a') - Q(s,a)] \quad (4.8)$$

where $\boldsymbol{\alpha}$ is the learning rate. The Q (s, a) table is randomly initialized. The agent starts to interact with the environment, and upon each interaction the agent will observe the reward of its action R(s,a) and the next environment state(s'). By using Bellman Equation in (4.8), the agent can update the estimated Q-values.

We will use Q-learning algorithm to be applied as a discrete action algorithm in autonomous driving systems. In chapter 4 we will describe the Q-leaning process in detail and how to implement the algorithm in TORCS simulator.

4.1.5 SARSA

SARSA is an abbreviation of (state, action, reward, state, and action) tuples that used to evaluate the optimal state value function < s, a, r, s' , a' >. It is very similar to Q-learning. Both of these two methods update the Q-value of one state-action pair in the table after observing a transition. SARSA uses the real action of the next state to evaluate the value, while Q-learning updates the Q estimate using the maximum Q-value independently of the action chosen. Therefore, SARSA is an on-policy method and Q-learning is an off-policy method.

The update of the state-action value of SARSA is expressed:

$$New\ Q(s,a) = Q(s,a) + \alpha[R(s,a) + \gamma Q'(s',a') - Q(s,a)] \tag{4.9}$$

The action (a) is selected based on the **ε**-greedy policy that used to solve the exploration and exploitation problem in reinforcement learning. With small probability **ϵ**, the agent will pick a random action to encourage exploration or with probability (1-**ϵ**) the agent will exploit the current optimal policy. With the training process, **ϵ** value can be decreased overtime as the agent becomes more confident with its estimate of Q-values.

4.2 Deep Reinforcement Learning

The traditional RL algorithms learns the action-value function Q(s,a), which measure how good to take an action a given a state (s). All the values are stored in tables. These traditional algorithms suffer from high dimensional state and action. If the combinations of states and actions are too large, it is impossible to learn such Q-table. The amount of memory used to store and update that table would increase as the number of states increases.

To overcome these problems, the role of Neural networks is coming to override the Q-tables functions. Instead of using tables, Deep Neural Networks (DNN) are used to estimate the action to take (policy based) or to estimate the value of a state (value based) that is why it is called "Deep". We are going to introduce some deep RL algorithms that can be applied in training our autonomous car such as DQN, DDPG, and A3C algorithms.

4.2.1 Deep Convolution Q learning (DQN)

Deep Convolutional Q Learning [5] is basically Q-learning but with deep neural networks. It takes the images as input and the output will be different predicted Q(s,a) values for each action possible in the given state as shown in Figure 4.2. We choose the biggest Q-value of this vector to find our best action. DQN employs a Neural Network to calculate the Q-value function. The neural network aims to estimate the value corresponding to each action.

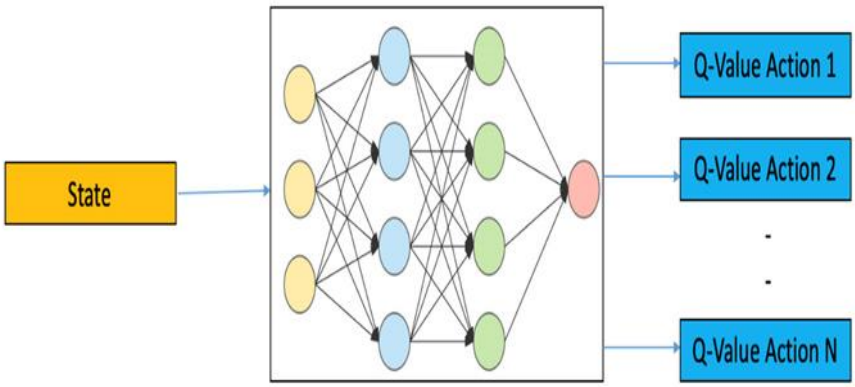

Figure 4.2 Deep Q Learning

4.2.1.1 DQN Architecture

The architecture of DQN network can be shown in Figure 4.3. The following techniques are essential for training DQN:

- **Experience Replay**

Training samples in RL environment is not very stable and less data-efficient. It makes the network harder to converge. So, by using experience replay, it is possible to reduce the convergence time. Rather than performing Q-learning on state/action pairs, the network stores the data discovered for [state, action, reward, next state] in a replay memory. While training, the network uses random mini-batches instead of the recent transition. This breaks the similarity of training samples and avoids the network to reach a local minimum.

- **Fixed Target Network**

We only have one network that calculates the predicted value and the target value that may lead to divergence and the network can become destabilized by falling into feedback loops between the target and predicted Q-values. So, instead of using one neural network for learning, we can use two networks one to calculate the predicted Q-Values and another one called target network to estimate the target. The target Q Network has the same structure as the main network but, the target network's weights are fixed and only periodically or slowly updated to the primary Q-networks values. This results in more stable training since it remains the target function fixed.

- **Exploration-Exploitation**

At the beginning of the training, we want the agent to explore more and takes random actions from the action-space with a certain probability ε (epsilon). This is called **exploration** as the agent tries to gather more information and explores possible new paths. The value of ε gets decreased over training time as the Q-function converges, it returns more consistent Q-values. We let the agent to take the best possible action according to ε-Greedy policy that allow the agent to the take the "greedy" action with the highest Q-value. This is called **exploitation**. So, when ε = 1 that means the system makes random actions to explore but when ε = 0 that means the model exploits the greedy action.

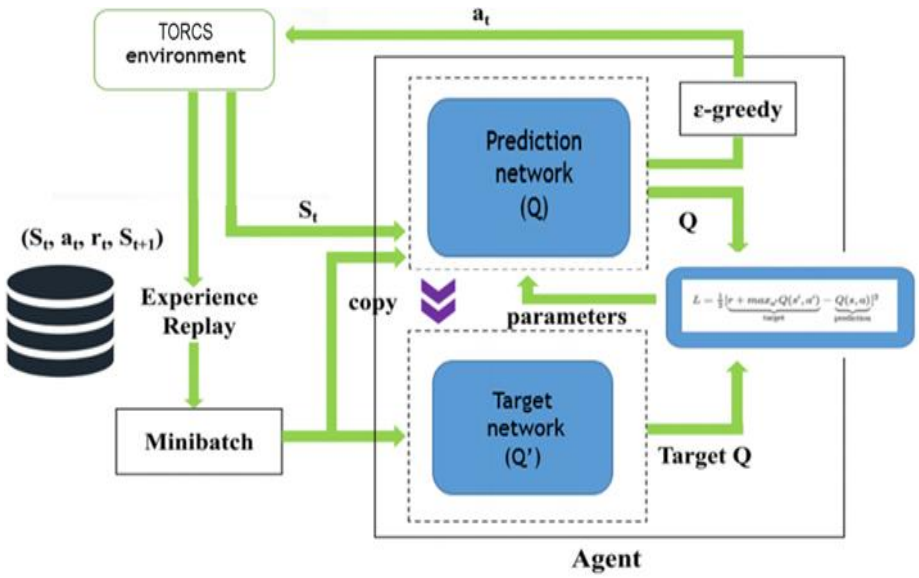

Figure 4.3 DQN Architecture

4.2.1.2 DQN Implementation

The main procedures of DQN are as follows:

1.Initialize replay buffer D

2. Pre-process and the environment and feed state (*s*) to DQN, which will return the *Q* values of all possible actions in the state.

3. Choose an action based on the epsilon-greedy policy: with the probability epsilon. choose an action which has a maximum *Q* value, such as:

$$a = \text{argmax}\,(Q\,(s, a, \theta)) \tag{4.10}$$

4.Execute the action at to the environment, and obtain the next state st+1 and reward rt.

5. Store transition in replay buffer as <s, a, r, s'>.

6. Next, take some random batches of transitions from the replay buffer and compute the loss using the following formula:

$$LOSS = (r + \gamma\, maxQ(s', a'; \theta') - Q(s, a; \theta'))^2 \tag{4.11}$$

7. Use gradient descent with respect to network parameters (θ) in order to minimize this loss.

8. Copy our real network weights to the target network weights for every *k* steps.

9. Repeat the previous steps for *M* number of episodes.

The following pseudo-algorithm implements the Deep-Q Learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise state s_t
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(s_t, a; \theta)$
    Execute action a_t and observe reward r_t and state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in $\mathcal{D}$
    Set $s_{t+1} = s_t$
    Sample random minibatch of transitions (s_t, a_t, r_t, s_{t+1}) from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } s_{t+1} \\ r_j + \gamma \max_{a'} Q(s_{t+1}, a'; \theta) & \text{for non-terminal } s_{t+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(s_t, a_j; \theta))^2$
  end for
end for

Algorithm 4.3 Pseudo-Algorithm for Deep-Q Learning with Experience Replay [31]

4.2.2 Deep Deterministic Policy Gradient (DDPG)

DDPG is an extension of the DQN algorithm that can acquire control policies for continuous action spaces. Although DQN achieved enormous success in higher dimensional problems such as the Atari game, the action space remains discrete. For higher dimensional sates, it will be difficult to discretize such the action space [26]. Unlike the DQN that takes only the states at its input, DDPG requires both state and action can estimate the Q-values. You can see the difference between them at Figure 4.4.

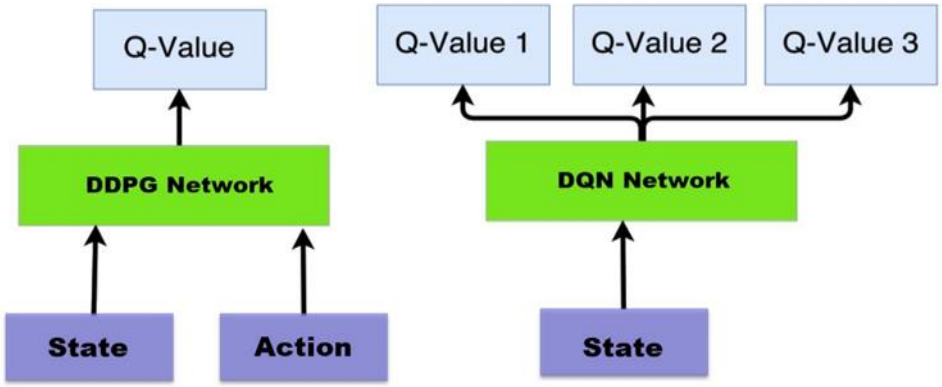

Figure 4.4 Difference between DQN& DDPG

DDPG consists of Actor and Critic networks, it learns both a policy and value function (Q function). Actor network is used to calculate the Q-values for each action, while the critic is used to Criticize the Q-values that are calculated using actor networks by calculating the error difference between the expected and actual Q-values. Like DQN, DDPG makes use of experience replay buffers and a stationary target network to stabilize training. The difference between them is that the critic used in DDPG takes as input both the states and actions. Also, the critic does not output a Q-value for every action (otherwise, there would be infinitely many outputs!), but instead the architecture has only one neuron that outputs values for each state-action pair.

We will use DDPG algorithm to be applied as a continuous action algorithm in autonomous driving systems. We will describe the DDPG in detail in chapter 5, the actor-critic architecture of the algorithm, and how to implement it in TORCS simulator. Also, we will describe different models and techniques that we could use in order to enhance the performance of the algorithm that is better in continuous action control compared to Q-learning algorithm.

4.2.3 Asynchronous Advantage Actor Critic (A3C)

A3C algorithm [21] is currently one of the most powerful algorithms in deep reinforcement learning it is similar to DQN where an actor-critic network is used to train our agent but the difference is that instead of training single agent in single DQN network, A3C can train multiple agents in parallel at different environments in different ways so that they can share their experience with one another and help each other out while exploring their environment in different ways as shown in Figure 4.5.

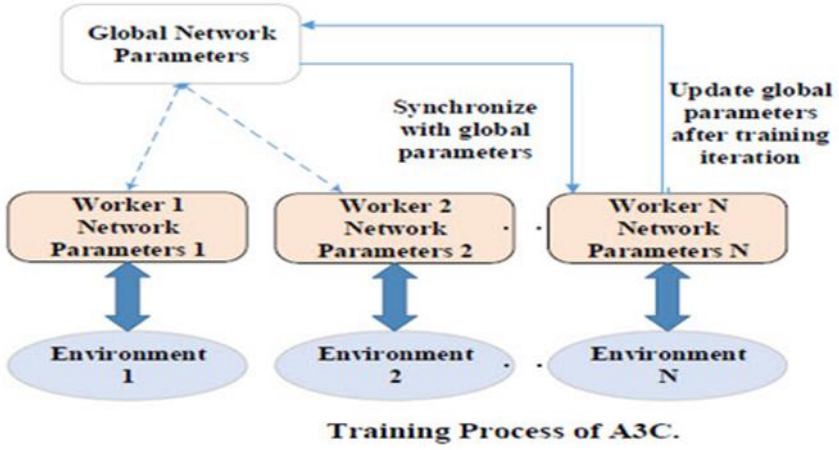

Figure 4.5 The A3C process [20]

4.2.3.1 A3C Architecture

The following concepts describe the meaning of A3C algorithm

- **Actor-critic**

The algorithm predicts both the value function V(s) (critic) as well as the optimal policy function (Actor). The neural network has two outputs: the critic measures how good the action taken is (value-based) V(s) and the actor outputs a set of action probabilities the agent can take(policy-based) Q (s, a). The agent updates the policy (the actor) based on the value estimate (the critic) so that the actor can output better actions that result in a higher reward.

- **Asynchronous**

In contrast to DQN and DDPG which use a single agent and a single environment, A3C algorithm is an asynchronous algorithm where multiple worker agents are trained in parallel, each with their own environment asynchronously. Each agent or worker is controlled by a

global network. As each agent learns more information, it contributes to the overall knowledge of the global network. There is one global network now shared between all agents as the critic will help the Q-values of each agent improve. As a result, the overall experience available for training becomes more diverse.

- **Advantage**

The Advantage is how the Critic informs the actor network if the expected Q-values from the ANN are good or bad. It computes the policy loss.

$$A = Q(s, a) - V(s) \quad \text{(advantage equation)} \tag{4.12}$$

Essentially, the advantage subtracts the known value of a state [V(s)] from the expected selected action from the Actor [Q (s, a)]. The critic is aware of the value of state, but it doesn't know how much better the Q-value chosen is relative to the current value of state. This is where the advantage comes in. The higher the advantage, the more likely the agents are to perform such actions. The policy loss helps to improve the agent's behavior by making them do more of the positive actions rather than the negative impacting ones.

- If the Actor chooses a good action for the agent, then the Q (s, a) > V(s)

- If not, then the policy loss is back propagated across the Global Network, with the weights adjusted in order to maximize Advantage and improve the selected action for the future.

4.2.3.2 A3C Implementation

To explain training process of each worker agent in A3C algorithm, the following steps summarize the training process in TORCS simulator as shown in Figure 4.6.

1.Worker reset to global network: a number of agents or workers are instantiated to start training.

2.Worker interacts with environment: each worker interacts themselves in their own environment using a copy of the global network.

3.Worker calculates value and policy loss:

During training, the worker picks the action with the highest probability given by the output of the discrete policy layer, then the worker executes the action while the environment returns the next state and the reward. The value loss is calculated based on the TD error which

is the squared difference between the target value and the estimated value. The policy loss is computed based on the logarithm of the taken actions multiplied by their probabilities and multiplied by the advantages.

4.Worker get gradients from losses:

The gradient of the loss function with respect to the neural networks weights is calculated and applied using an optimizer such as the Adaptive Moment Estimation (Adam) optimizer.

5.Worker updates global network with gradients

Finally, each worker updates its information with the global network and starts again the training process steps until convergence.

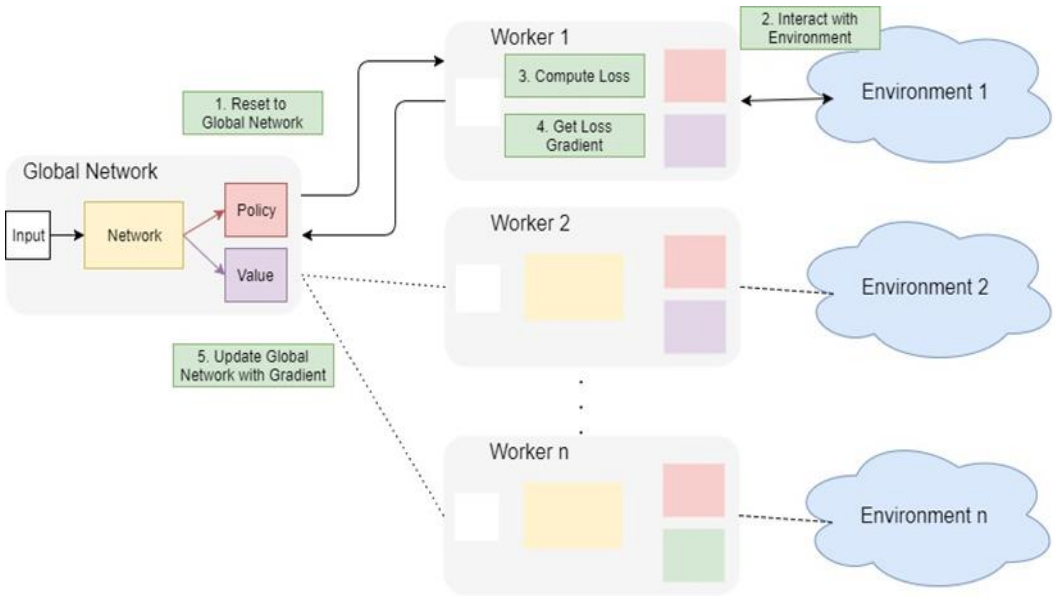

Figure 4.6 A3C implementation in TORCS

Chapter 5 : Q-Learning as a Discrete Action Algorithm

After we discussed the learning algorithms that we could use to learn our self-driving car on TORCS, we will apply our first RL algorithm. We selected Q-learning as a discrete action algorithm. In this chapter, we will discuss the concept of Q-learning and how to implement it to learn our agent. We will show the results of our implementation and discuss if that algorithm is applicable in autonomous driving cars or not.

5.1 Q-Learning concept

Q-learning [25] is a model-free RL algorithm that can be used to learn the value of an action in a given state. It does not require a model of the environment, so it is called "model-free" algorithm and it can learn the policy which can inform the agent that what actions should be taken for maximizing the reward under what conditions. It is only applicable for a small amount of states [5].

Q-Learning is a RL method that uses the Bellman's equation (5.1) to update its Q-values. The Q-value [Q(s,a)] determines the value of the agent being in a certain state and taking a certain action to go to that state.

$$New\ Q(s,a) = Q(s,a) + \alpha[R(s,a) + \gamma\ maxQ'(s',a') - Q(s,a)] \quad (5.1)$$

Where:

- R(s,a): reward for taking action at the given state.

- α is the learning rate ($0<\alpha\leq1$) - which defines to what extend new information overrides old information.

- γ is the discount factor ($0\leq\gamma\leq1$) that specifies how much importance we want to give to future rewards. The long-term effective award is given to the higher discount factor (close to 1), Whereas a discount factor of 0 allows our agent to consider only immediate reward, hence making it greedy.

The Bellman equation connects states and, as a result, action value functions. It allows us to get through the environment and to calculate the optimum values that give us the best policy in turn. Q values in their simplest form are a matrix of states in rows and actions in

columns. We randomly initialize the Q-matrix, the agent begins to communicate with its environment, and measures the reward for each action. The observed Q values are then calculated and the matrix is updated.

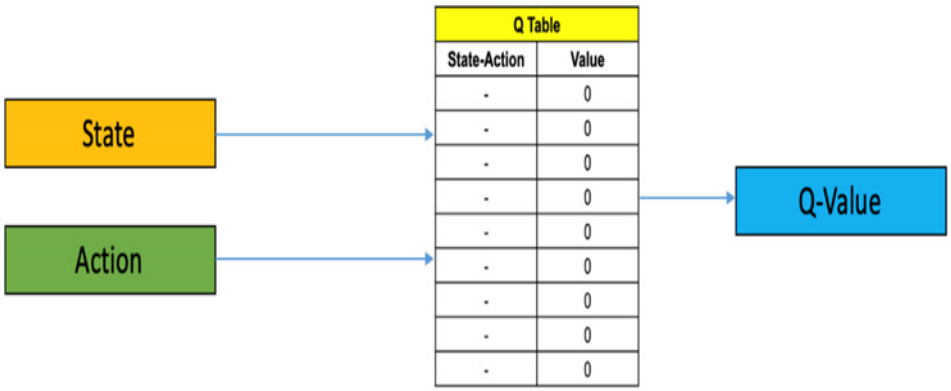

Figure 5.1 Q-Learning

The Q-value of a state-action pair is the sum of the immediate reward and the discounted future reward. We store the Q-values for each state and action through a Q-table that is created during the Q-learning process. This means that we construct only one Q-table which serves as a mapping for the Q-Values of the states and actions. These Values are updated depending on the reward value came from the environment for the current state and action. It is obvious that this algorithm depends on the optimal future value that will ensure taking the optimal or best action and achieve the optimal policy for the problem by an iterative method across the episodes applied.

Initially, the Q-table is constructed as zero values for the discretized number of states and number of actions. If we know the state and the action, we will know which Q-value needed to be updated depending on the previous equation. Applying this algorithm iteratively will lead to construct the Q-table which can be taken as an input and follow it in the testing phase for the autonomous car.

The Q-learning algorithm Process

The process of the Q-learning algorithm is shown in Figure 5.2 and can be summarized in the following steps [35]:

Step 1: Initialize Q-Table

We first build a Q-table (m*n) where m rows are the number of states and n columns are number of actions. We initialize the Q-table at zero values.

Step 2: Iteration

Steps 3–5 will be repeated until we exceed the maximum number of episodes or manually stop the training.

Step 3: Choose and perform an action

We will choose an action (a) in the current state (s) based on the Q-Table. At the beginning, the Q-Values equal 0, so we will use concept of exploration and exploitation that allow our agent to do a lot of exploration, by randomly choosing our actions. We set epsilon value at 1 because we don't know anything about the values in Q-table. As the agent explores the environment, the epsilon value decreases and the agent starts to exploit the environment that allows our agent to best action depending on what was learnt before, so it becomes more confident at estimating Q-values.

Steps 4–5: Evaluate

Perform the action (a), observe the outcome state (s') and reward r, and update the function Q (s, a). We take the action (a) that we selected in step 3, and then performing this action returns us a new state (s') and a reward r. To update Q (s, a) we use the following Bellman equation as we stated before.

$$New\ Q(s,a) = Q(s,a) + \alpha[R(s,a) + \gamma\ maxQ'(s',a') - Q(s,a)]$$

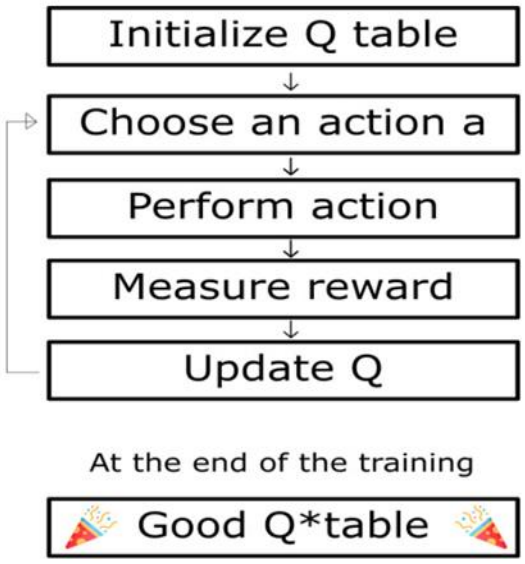

Figure 5.2 The Q-learning algorithm Process [35]

Initialize $Q(s, a)$ arbitrarily
Repeat (for each episode):
  Initialize s
  Repeat (for each step of episode):
    Choose a from s using policy derived from Q (e.g., ε-greedy)
    Take action a, observe r, s'
    $Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right]$
    $s \leftarrow s'$;
  until s is terminal

Algorithm 5.1 The Q learning algorithm's pseudo-code [35]

5.2 TORCS as MDP

In order to apply Q-learning algorithm in TORCS, we need to discuss The MDP components to make our algorithm as model-free RL algorithm instead of the transition model that contains many transitions and is highly complex. MDP consists of the following components [15]:

5.2.1 States

Sensor readings from the Server are very useful for controlling the car as they represent car states. The states are the car speed along the track, its position on the track, the angle with respect to the track axis, and five distance sensors that determine the distance to the track's edge. To account for noise, the 20 inputs are measured as an average over sensors 6, 7, 8, 10, 11, 12, respectively, rather than directly from sensors 7 and 11.

The most important sensor readings are shown in the following Table 5.1:

Sensor	State Description
Speed X	Longitudinal speed in the car axis
Angle	Angle between the car direction and the track axis
Trackpos	Distance between the car and the track axis
Track	Distance sensor at −40◦, −20◦, 0◦, 20◦, 40◦ Distance between the car and track [5, 7, 9, 11, 13]

Table 5.1 TORCS Main States

5.2.2 Actions

There are five action dimensions available in TORCS accelerate, brake, gear, meta, and steer. Since brake is the opposite of accelerate, so they will be considered as one parameter and Meta is considered as the Restart in case of failure. The following Table 5.2 will show TORCS actions:

Action	Range	Description
Accel	[0,1]	Virtual gas pedal (0 means no gas, 1 full gas)
Brake	[0,1]	Virtual brake pedal (0 means no brake, 1 full brake)
Gear	-1,0,1,2,3,4,5,6	Gear value which can be controlled based on the RPM of the car.
Steer	[-1,1]	Steering value: -1 and +1 means respectively full right and left
Meta	0 or 1	This is meta-control command: 0 do nothing, 1 ask competition server to restart the race

Table 5.2 TORCS Main Actions

5.2.3 Rewards

We want to control the car to drive autonomously. Due to random choice of actions taken, there are good actions and bad actions, so we want to prevent the car from:

- Going out of the track
- Stopping in a stuck position
- Making bad actions

The reward function will be calculated according to the following scenarios:

1. Car make good lane keeping:

   The reward will be positive and high if the distance was long and in general it has a continuous range from [-1,1].

2. Car is in a stuck position:

   It is unwanted blocking state, so the reward will be negative and equal to -1 to prevent doing this another time.

3. The car goes out of the track

   It is a forbidden scenario so; agent sends a Meta action to restart the race. We set the reward to be negative and equal to -1.

5.3 Q-Learning implementation

We will discuss how to implement the concepts of states, actions, and reward that mentioned above to connect our agent as a client to the TORCS simulator as a server. Our code consists of four parts:

5.3.1 Client

It is the code that establishes a communication channel with the server host using a special port and socket, then calls the driver module which is our game loop, and uses the msgParser to send UDP messages to the server.

5.3.2 msgParser

It has a server-client communication for message builder and receiver. It generates the control actions' messages, then sends them to the server and receives UDP messages from the car state server.

5.3.3 Driver

The Driver function processes the input and computes the output to the server. it consists of two parts: one that checks whether the agent is stuck or outside the road and one that handles action selection and learning.

- **CheckStuck**

This function determines whether the car is stuck or not. If the angle of the car exceeds 45 degrees, it is considered stuck. The episode ends if it remains stuck for more than 25 game ticks and the traveled distance is less than 0.01m. The stuck timer is reset to zero if the agent is not stuck.

- **Learning Interface**

It is a specific interface used to connect the learning algorithm with the driver. The primary role of the learning interface is to pick actions and call the update function of the learning algorithm. The elements of the Learning interface are: GetState, ActionSelection, RewardFunction and QtableUpdate.

5.3.4 Learning Interface steps

5.3.4.1 GetState

First, we discretize the distance sensor and speed values as follows:

- speedList = [0,10,20,30,40,50,60,70,80,90,100,110,120,130,140,150]
- distList = [-1,0,5,10,20,30,40,50,60,70,80,90,100,120,150,200]

We have carefully chosen these values to cover all possible states and we expressed each of them in 4 bits of binary form as they are 16 discretized values. We have used only 1 sensor out of 5 sensors in which we take the maximum so that we can minimize the number of bits representing the state, so we represent all 5 sensors in the 20 bits that make the states equal to 220, which is not possible, and we would need to represent this in 20 bits. Thus, we used an additional 3 bits to separate the maximum of 5 sensors, giving us 7 bits to describe the value of the sensor.

Eventually, we have a total of 11 bits for the states which corresponds to 2048 possible state e.g., Distance sensor = 200, Maximum sensor No. is 9, Speed = 90, this is equivalent to 1001 010 1111.

5.3.4.2 ActionSelection

To apply Q-Learning, we need to discretize steering angle values as we will see in the following table, but actually steering must be compatible with both accelerate and brake, so they are strongly related with each other. Initially, we had [-1, -0.5, -0.1, 0, 0.1, 0.5, 1] values for steering angle, but in order to avoid sharp edges steering curves, we removed both of (-1) and (+1). In addition to these values, we may have accelerated or brake with can be modeled as (+1 or -1), so we have the following combinations:

Steer	**Accelerate (1)**	**Neutral (0)**	**Brake (-1)**
0.5(left)	0	1	2
0.1(left)	3	4	5
0	6	7	8
-0.1(right)	9	10	11
-0.5(right)	12	13	15

Table 5.3 TORCS designed actions for Steering, Acceleration, and Brake [15]

The action is chosen at random from a range of 0 to 1. If the number is less than η, the agent takes action based on its heuristic. Otherwise, if the random number is less than the sum of η, it performs a random action. This makes it easier to use of the same random number to choose a random action with probability. If neither condition is met, the agent takes a greedy action (max action) based on its Q-table.

The heuristic action serves as a guide rather than a teacher. As a result, the random exploration is needed to learn how to improve the heuristic policy. Given the current state, the maximum action is the action with the highest Q-value in the Q-table.

5.3.4.3 RewardFunction

We have 3 different scenarios:

- If the car is stuck (traveled distance < 0, 01m)

It receives -2 reward and sends a meta action to restart the episode. Since it is an unacceptable action, therefore we ensure that it never happens again.

- If the car is out of track (abs (track position) > 1)

It receives -1 reward that serving as an extra penalty for behavior and ends the episode.

- If the car is neither stuck or out of track

It receives a reward based on track position, angle, and travelled distance with a maximum value of 1.

5.3.4.3 QtableUpdate

First, we search to see if the current state is still in the table; if not, we establish it in the Q-table, which is initially set to 0 for all actions. Then, we can update the Q-values of the previous state using the current state, the previous state, action, and reward. Many actions may have the same value for example, when exploring a new state and all values are unknown (and all have default value 0). The agent then must make a decision based on something that is not dependent on the value. It could choose an action based on some heuristic, or simply the first action that appeared.

5.4 Evaluation and Results

We trained our model on track (CG Speedway number 1) as shown in Figure 5.3 as it is a simple track for faster training and contains both: straight and curved parts.

Figure 5.3 CG Speedway Track

We trained the model for more than 1000 episodes and our Q-table is constructed as shown in Figure 5.4 that consists of the 2^{11} = 2048 possible states and the corresponding Q-values that represent the actions taken for each state.

States	Q-values (actions)													
(0, 0, 1, 1)(0, 0, 1)(0, 0, 0, 1)	-0.67404	-0.69444	-0.77654	-0.44863	-0.42214	-0.64937	-0.84675	-0.63835	-0.60312	-0.58882	-0.59611	-0.59505	-0.62919	-0.58199	-0.80146
(0, 0, 1, 1)(0, 0, 1)(0, 0, 1, 0)	-0.00838	-0.24059	-0.02017	0.011282	-0.00509	-0.52398	-0.22262	-0.00993	-0.25792	-0.26389	-0.00783	-0.0052	-0.00714	-0.22912	0.002783
(0, 0, 1, 1)(0, 0, 1)(0, 0, 1, 1)	-0.10359	-0.22216	0.003587	0.116848	0.003575	0.006233	0.007071	0.006254	0	0	0.011647	-0.00508	0.015846	0.01992	-0.06577
(0, 0, 1, 1)(0, 0, 1)(0, 1, 0, 0)	0.053693	0.023922	0.006021	0.029246	0.035546	0.060648	0.041568	0.021634	0.022552	0.04996	0.596724	0.016952	0.005257	0.022279	0.249002
(0, 0, 1, 1)(0, 0, 1)(0, 1, 0, 1)	0.019081	-0.17867	0.015793	0.008344	-0.25398	0.012088	0.012978	-0.25596	0.004796	0.009358	0.014836	-0.05785	0.028076	0.014824	-0.16623
(0, 0, 1, 1)(0, 0, 1)(0, 1, 1, 0)	-0.08631	-0.00807	0.041824	0.004815	0.002524	-0.29491	-0.32831	0.001697	0.005638	-0.3475	0.002225	0.004892	-0.30114	0	-0.24632
(0, 0, 1, 1)(0, 0, 1)(0, 1, 1, 1)	0.007908	0.020804	-0.07809	0	0.004928	0.019332	0.01758	0	0.00643	0.002375	0	0.019712	0.00207	0.014503	0.192673
(0, 0, 1, 1)(0, 0, 1)(1, 0, 0, 0)	0.019776	0	0.011055	0.015419	0.020438	0.005038	0.008018	0.020005	0.016004	0	0.009691	0.016143	0.444972	0.007605	0.109123
(0, 0, 1, 1)(0, 0, 1)(1, 0, 0, 1)	-0.00636	0.02717	0.004822	0.056751	0.005412	0.042405	0.01276	0.025806	0.010575	0	0.007296	0.02798	0	0.023258	0.339508
(0, 0, 1, 1)(0, 0, 1)(1, 0, 1, 0)	-0.15007	0.025167	0.009537	0.032096	0.021512	0.055008	0.019781	0.019133	0.041513	0.022826	0.004843	0.064674	0.021574	0.025288	0.605186
(0, 0, 1, 1)(0, 0, 1)(1, 0, 1, 1)	0.07331	0	-0.02073	0.048644	0.125117	0.008276	0.035375	0.07367	0.077467	0.039931	0.018832	0.12398	1.597939	0.058168	0.172331
(0, 0, 1, 1)(0, 0, 1)(1, 1, 0, 0)	0.039899	2.389088	0.130428	0.162632	0.079519	0.117546	0.140498	0.178135	-0.00986	-0.11551	0.197597	0.258257	-0.20831	0.129912	0.491602
(0, 0, 1, 1)(0, 0, 1)(1, 1, 0, 1)	0.056891	-0.07791	-0.05811	-0.2446	-0.17122	-0.03816	0.075003	0.080863	-0.06519	-0.14387	0.019701	0.107436	-0.16115	0.016784	0.933529
(0, 0, 1, 1)(0, 0, 1)(1, 1, 1, 0)	6.31E-02	0.031624	-0.08682	0.004382	-0.08164	0.044928	0.112004	-0.27723	-0.01839	0.056174	-0.00388	0.089146	-0.0721	-0.08928	0.673013
(0, 0, 1, 1)(0, 0, 1)(1, 1, 1, 1)	-0.03552	0.112125	-0.25483	-0.05285	0.081684	-0.03687	-0.00281	0.042397	0.043606	0.116713	0.059682	0.109314	0.033488	0.717722	-0.15887

Figure 5.4 Constructed Q-Table

From the training video at [36], it is obvious that Q-learning Algorithm suffers from taking discrete actions which effects on the car motion. This is very obvious from the Q-Learning path in Figure 5.5 that shows non smooth curve in both straight and curve parts of the track. If the combinations of states and actions are too large, it is impossible to learn such Q-table. Q-learning agent does not have the ability to estimate value for unseen states as it suffers from high dimensional state and action.

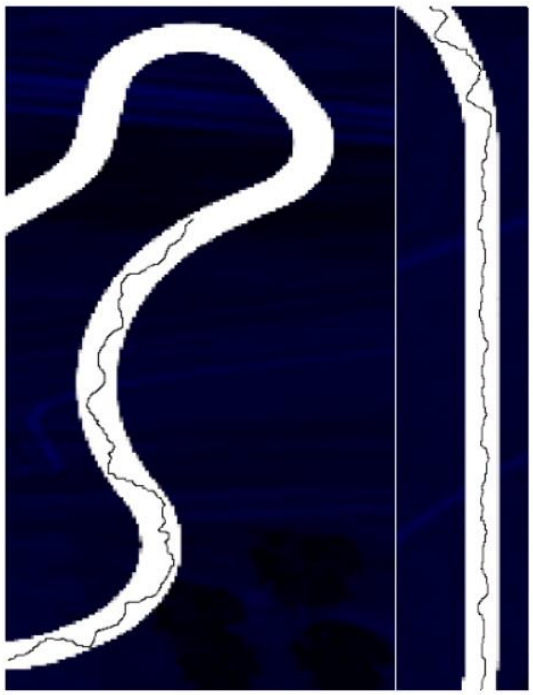

Figure 5.5 Q-Learning Algorithm Performance

Chapter 6 : DDPG as a Continuous Action Algorithm

Deep-Q-Network (DQN) algorithms solves problems with high dimensional observation space, but the problem is it can only handle discrete and low-dimensional action space. DQN cannot be applied to continuous domains as it relies on finding the action that maximizes action-value function. DQN can simply discretize the action space to be able to handle action space, but it has many limitations due to higher dimension action spaces. Such a large action space makes it difficult to explore efficiently. Discretization of action spaces throws away information of the action domain.

The Deep Deterministic Policy Gradient (DDPG) is one of the main algorithms for the Continuous Action Algorithms that tries to solve these problems. It is a model-free off-policy actor- critic algorithm using deep function approximators that can learn policies in high-dimensional, continuous action space.

In this chapter, we will deal with DDPG algorithm for continuous action algorithm, discuss in detail the architecture of DDPG, and how to implement it in TORCS. We will discuss the code that was implemented in keras and tensor libraries, the design of reward functions, and stochastic braking. Finally, we will plot the results obtained from training our code to show the performance of the algorithm.

6.1 DDPG Architecture

6.1.1 Actor-Critic Networks

DDPG [27] is a RL technique that takes the advantages of both Q-learning and policy gradients. DDPG being an actor-critic technique consists of two models: Actor and Critic as shown in Figure 6.1. Instead of using probability distribution over actions, the actor network takes the state as input and outputs the exact action (continuous), so it is the policy network. The critic is a Q-value network that takes both state and action as input and outputs the Q-values. DDPG is used in the continuous action control and the “deterministic” word in DDPG refers to the fact that the actor computes the action directly instead of a probability distribution over actions.

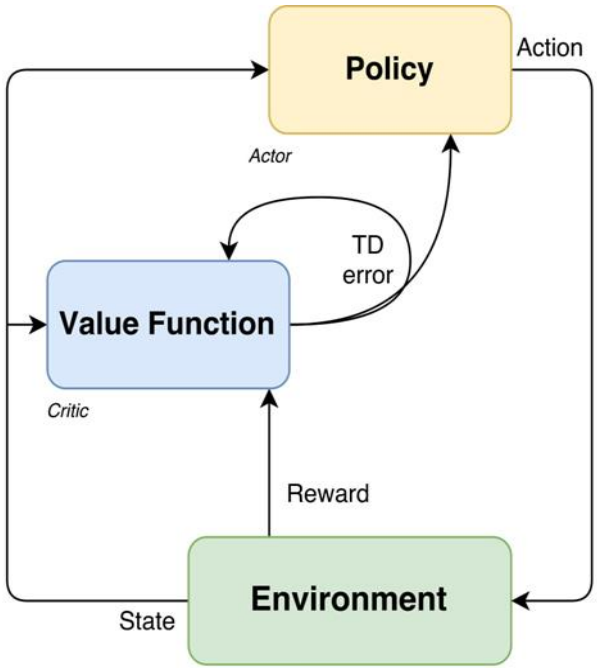

Figure 6.1 Actor-critic Networks

6.1.2 Target Update

The main network has to calculate the Q-values and the future Q-values at the same time, that lead to problems in stability. So, instead of using one network, DDPG also includes target critic and actor networks to calculate future Q-value to increase stability. The target networks are delayed networks compared to main networks and their weights are updated periodically based on the main networks [26]. In DQN the target network copies the weights of the main network periodically, this is known as a "hard update". DDPG don't copy all the weights but perform a "soft update" where only a fraction of main weights is transferred in the following manner:

$$\theta^{\mu}_{targ} \text{ <---------- } \tau\, \theta^{\mu}_{targ} + (1-\tau)\theta^{\mu} \tag{6.1}$$

$$\theta^{Q}_{targ} \text{ <---------- } \tau\, \theta^{Q}_{targ} + (1-\tau)\theta^{Q} \tag{6.2}$$

τ is a parameter that is typically chosen to be close to 1 (eg. 0.999)

6.1.3 Loss Functions

To train actor and critic networks, we need to calculate the loss function of both actor-critic networks as in the following:

- **Actor (policy network) loss:** The sum of Q-values for the states that we want to be maximized.

$$J_\mu = \frac{1}{N} \sum_{i=1}^{N} Q(s_i, \mu(s_i)) \tag{6.3}$$

- **Critic loss:** is a simple TD-error that target networks use to compute the future Q-values. We need to minimize this loss.

$$J_Q = \frac{1}{N} \sum_{i=1}^{N} (r_i + \gamma Q_{targ}(s_i', \mu_{targ}(s_i') - Q(s_i, \mu(s_i))^2 \tag{6.4}$$

6.1.4 Replay Buffer

DDPG also uses a replay buffer so that it can store sample experience that is used to update neural network parameters. The experience information (state, action, reward, next_state) are saved in memory. Then, we can sample random mini-batches of experience from the replay buffer when we need to update the actor and critic networks.

6.1.5 Exploration

In Reinforcement learning, exploration is required to allow the agent to explore more at the beginning of the training process. For discrete action spaces, exploration is done using probabilistically for selecting a random action. For continuous action spaces, exploration is done by adding noise to the action itself.

$$\mu'(s_t) = \mu(\left(s_t \middle| \theta_t^\mu\right) + N) \tag{6.5}$$

Algorithm 1 DDPG algorithm

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

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

    Update the target networks:

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

  end for
end for

Algorithm 6.1 DDPG algorithm's pseudo-code[26]

6.2 DDPG Implementation

The implementation of DDPG algorithm is shown in Figure 6.2 and can be written the following steps:

1) The actor chooses one action according to the behavior strategy a_t:

$$a_t = \mu(s_t|\theta^{\mu}) + N_t \tag{6.6}$$

The behavior strategy is based on the current online strategy μ and the random process of random UO noise generation, sampled from this random process a_t.

2) Execute a_t, return reward r_t and new state s_{t+1}

3) The actor saves this state transition process$\{ s_t, a_t, r_t, s_{t+1}\}$ into the replay memory buffer R as a dataset for training the online network.

4) From replay memory buffer R Medium, random sampling NA transition data is used as a mini-batch training data for the online strategy network and the online Q network. We use $\{ s_t, a_t, r_t, s_{t+1}\}$ to represent a single transition data in the mini-batch.

5) Calculate the gradient of the online Q network:

The network loss function is defined as MSE: mean squared error:

$$\mathrm{L} = \frac{1}{N}\sum_i \left(y_i - Q(s_i, (a_i|\theta^Q)\right)^2 \tag{6.7}$$

where y_i can be seen as a "label":

$$y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\, \theta^{Q'}) \tag{6.8}$$

6) Update online Q: Use Adam optimizer to update θ^Q

7) Calculate the policy gradient of the policy network: a function representing the performance objective J against θ^Q The gradient.

8) Update online strategy network: use Adam optimizer to update θ^{μ}

9) Soft update target network μ ′ with Q ′ by using the running average method to soft update the parameters of the online network to the parameters of the target network:

$$\theta^{\mu}_{targ} \text{ <---------- } \tau\,\theta^{\mu}_{targ} + (1-\tau)\theta^{\mu}$$

$$\theta^{Q}_{targ} \text{ <---------- } \tau\,\theta^{Q}_{targ} + (1-\tau)\theta^{Q} \tag{6.9}$$

τ is a parameter that is typically chosen to be close to 1 (eg. 0.999)

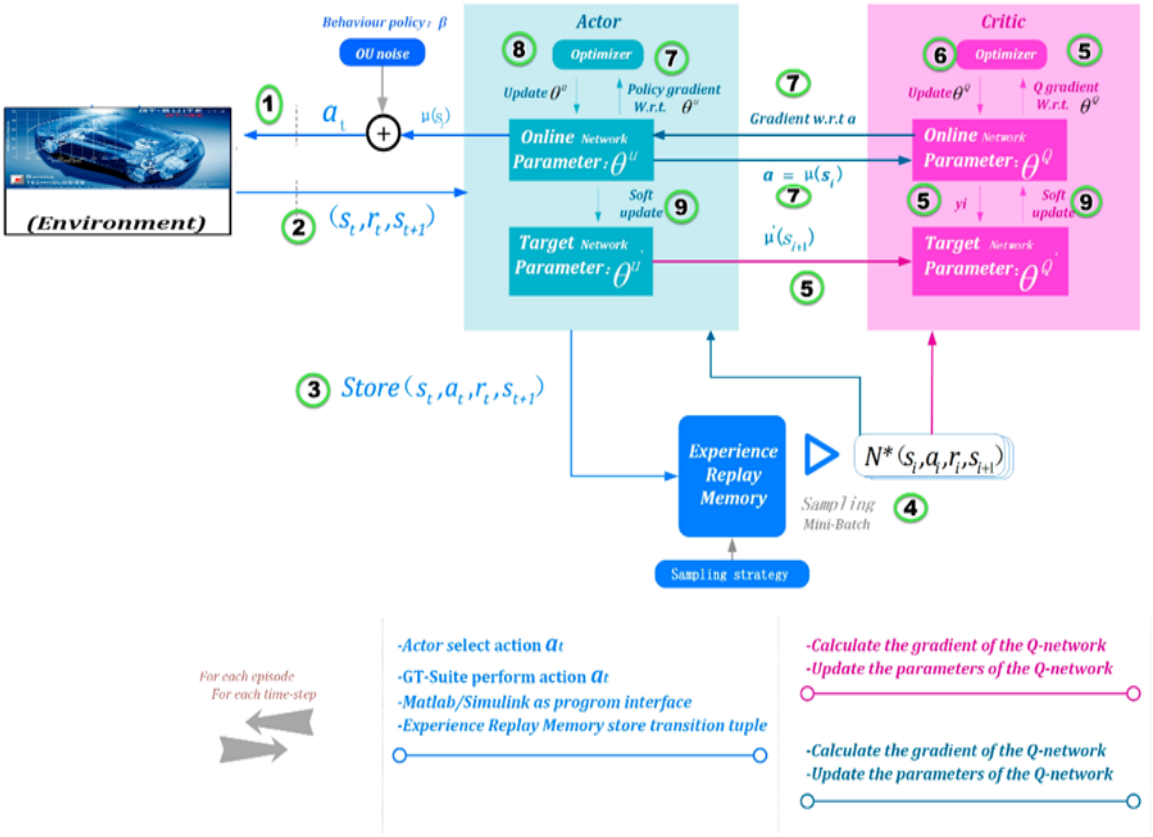

Figure 6.2 DDPG implementation framework

6.3 DDPG Code Explanation

After sensors receive input in the form of an array, the sensors input will be fed into our neural network, and then the network will output 3 real numbers (the value of steering, acceleration and braking). The network will be trained many times, using DDPG algorithm to maximize future expected returns.

6.3.1 Actor & Critic Network

- **Actor Network**

The actor network takes states from the environment. We used 2 hidden layers with 300 and 600 hidden units that use relu activation function at its output for both. Actor Network parameters are updated from the gradients computed from the Critic Network. The output consists of 3 continuous actions:

1. Steering, which is a single unit with tanh activation function (output -1 means maximum right turn, +1 means maximum left turn)
2. Acceleration, which is a single unit with sigmoid activation function (output 0 means no acceleration, 1 means full acceleration).
3. Brake, another single unit with sigmoid activation function (output 0 means no braking, 1 means emergency braking).

- **Critic Network**

The critic network takes both the states and action as inputs not only the states and predicts the Q-Value. We used 4 hidden layers:

1. 2 hidden layers with 300 then 600 hidden units for the states, relu then linear activation function, respectively.
2. 1 hidden layer with 600 hidden units for the actions with linear activation function.
3. the last hidden layer has 600 hidden units merging the output of action and state hidden layers together with relu activation function and the output is with linear activation function.

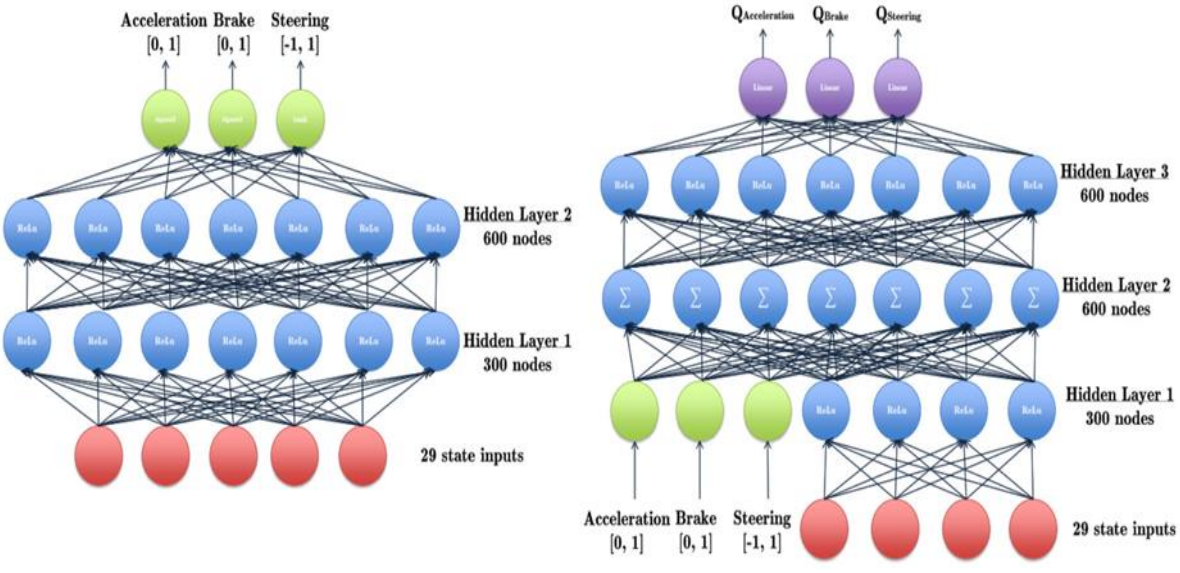

Figure 6.3 Actor (Left) & Critic(Right) Network

6.3.2 Reward design

It is necessary to define a proper reward function in reinforcement learning problems. We want the car to go as fast as possible, as far as possible all the while staying inside the track.

First, I tried to apply the reward $R_t = V_x cos(\theta)$ that is the speed of the car projected to the axis of the road as shown in Figure 6.4, but the training is not very stable as mentioned in [26] as the AI will try to accelerate the gas pedal very hard (to get maximum reward) and it hits the edge and the episode terminated very quickly. Therefore, the neural network stuck in local minimum.

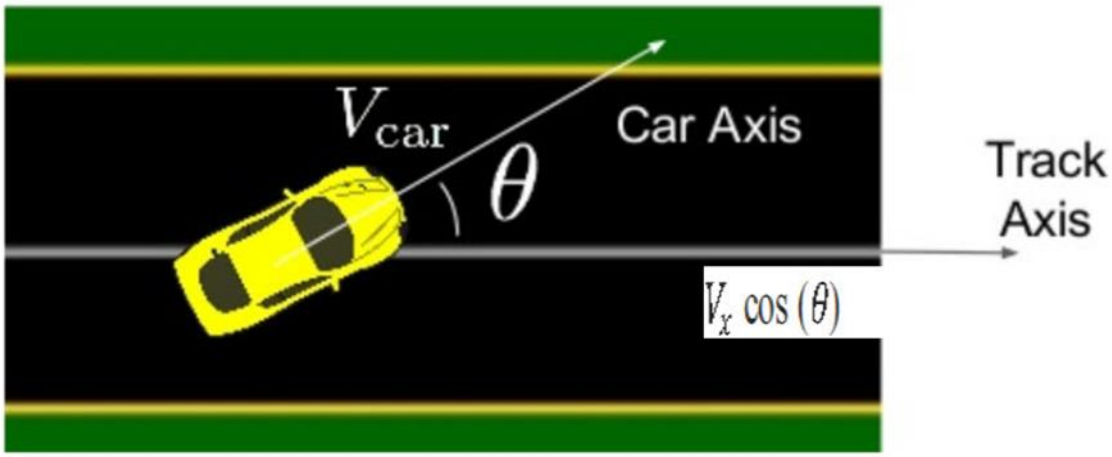

Figure 6.4 Car position with respect to track [28]

So, I tried to find another reward function to avoid the previous problem. I used the one [28]

$$R_t = V_x cos(\theta) - V_x sin(\theta) - V_x |track\ Pos| \tag{6.10}$$

It is a more efficient reward is to keep maximizing longitudinal velocity (first term) but also minimizing the transverse velocity (second term) and we also penalize the AI if it constantly drives very off center of the track (last term). The problem with this reward is that the model has difficulties to learn how to brake because it diminishes it (hence it will tend to accelerate a lot while it's not outside of the track) which is tackled by imposing a 10% brake during exploration.

I modified the previous reward function to be:

$$R_t = V_x cos(\theta) - V_x sin(\theta) - 2 * V_x |track\ Pos| sin(\theta) - V_y cos(\theta) \tag{6.11}$$

Where V_y is the transverse velocity.

6.3.3 Exploration Strategy

We have used ϵ greedy policy [5] in our RL problem like packman and Atari breakout where the agent to try a random action some percentage of the time. This approach does not work very well in TORCS as we have 3 actions (steering, acceleration, brake). If I just randomly select from uniformly distributed actions, some boring combinations will be produced (for example, if the braking value is greater than the acceleration value, the car will not move at all). Therefore, we use the Ornstein-Uhlenbeck process to add noise to actions for helping in exploration. Ornstein-Uhlenbeck process is a stochastic process which has mean-reverting properties.

$$dx_t = \theta(\mu - x_t)dt + \sigma dWt \tag{6.12}$$

Where θ reflects how fast the variable reverts to the mean. μ stands for balance or mean. σ is the degree of fluctuation of the process. The following table shows the recommended values used in the code.

Action	θ	μ	σ
steering	0.6	0.0	0.30
acceleration	1.0	[0.3-0.6]	0.10
brake	1.0	-0.1	0.05

Table 6.1 Parameter values used in OU process [28]

The most important parameter is the acceleration μ. You want the car to have a certain initial speed without falling into a local minimum where the car keeps on the brakes and no longer steps on the accelerator.

6.3.4 Stochastic Braking

Braking is an important driving action, teaching a model to brake is much more difficult than teaching it to steer or accelerate. The reason for this is simple: it has to do with the reward function. Because the reward is proportional to the velocity of the car, slowing it down results in a lower reward. As a result, the AI agent tries to avoid braking as much as possible. Also, it is not a fine decision to experiment with braking and acceleration at the same time during training because the model can learn to brake harder before learning to accelerate well, pushing it to a local minimum (car coming to a halt and no reward is received). One approach is to

employ stochastic braking [26]. During the exploration process, the car should be allowed to brake 10% of the time. The other 90% of the time, we don't let the car brake, allowing it to learn how to accelerate. The model avoids local minima during training by allowing the car to gain some non-zero velocity while also learning how to brake.

6.4 Evaluation and Results

We trained our DDPG algorithm using Tensorflow and Keras deep learning frameworks. Our experiments were made on an Ubuntu 18.04 machine, with 12 cores CPU, 64GB memory and NVIDIA GeForce GTX 1060 GPU. Our code parameters are selected according [28]. The selected optimizer is Adam with learning rates of 0.0001 and 0.001 for the actor and critic respectively, and a batch-size of 32. The size of the replay buffer is 100000 state-action pairs, with a discount factor of $\gamma = 0.99$.Target networks are updated gradually with $\tau = 0.001$ that stabilizes the convergence.

From the training phase, DDPG shows good performance and smooth actions in both straight and curved parts of the track as shown in Figure 6.5. DDPG algorithm learns faster how to complete one lap on that same track compared to Discrete actions algorithms such as Q-learning as mentioned earlier. During the Training phase, DDPG algorithm covers many laps in a smooth way, and actually this performance is extended also to be found the testing phase on different tracks. As a result of learning in shorter time, it is obvious that the number of episodes to build the model will be low.

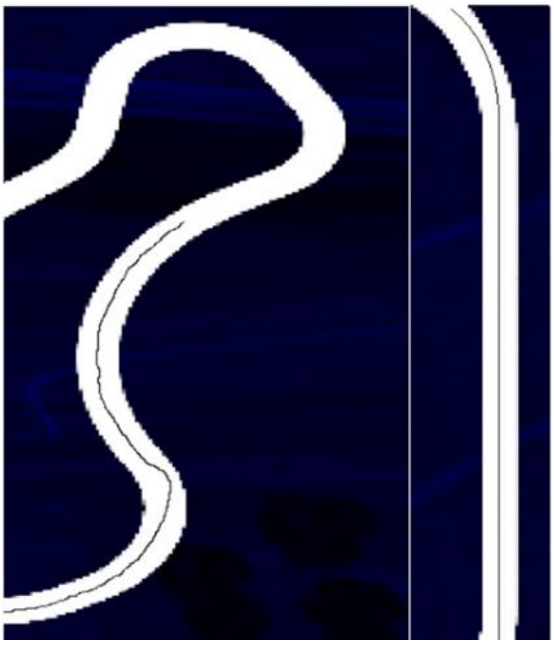

Figure 6.5 DDPG Algorithm Performance in Straight and curved parts

We trained our model on track (CG Speedway number 1) that we used in the previous chapter as shown in Figure 5.3 that is a simple track for faster training. Our training is more than 300K steps that is sufficient for our model training and took about 18 hours for complete training. For validation of our model, we tested it on a Challenging Alpine-1 track (Figure 6.6) for about 5000 steps.

Figure 6.6 Alpine-1 Track

You can see the training and testing of our racing car in the following URL references:

- First Reward Function:

  Training: [37] Testing: [38]

- Modified Reward Function:

  Training: [39] Testing: [40]

We noticed from the videos that at first reward function, our agent faces some problems when trying to turn right or left but with the modified reward function it takes smoother actions that speeds the car and completes the lap in less time.

- **Results using the different Reward Functions**

We plot the reward functions to validate our model. We can see from Figure 6.7 when we used the first reward function to train on the CG Speedway Track that the reward is increased gradually to be 50 in average and after 50K steps it still increases to be 100 in average.

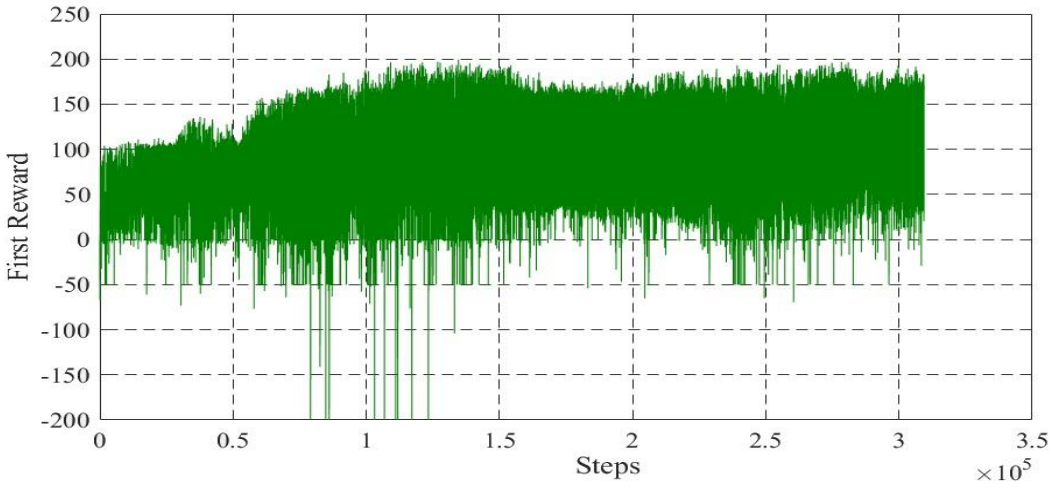

Figure 6.7 Reward obtained in training using first reward function

We want to increase the reward that the car got from the environment so that it can reach the target faster and improves its speed during the racing. We plot our modified reward function as we can see at Figure 6.8. It suffers from many negatives at the beginning of the training due to the added 2 extra penalizing terms to the reward function, but after 100K steps the reward is increased and reached in average to 150 which is higher the first reward function. So, it improves the stability and learning time of TORCS.

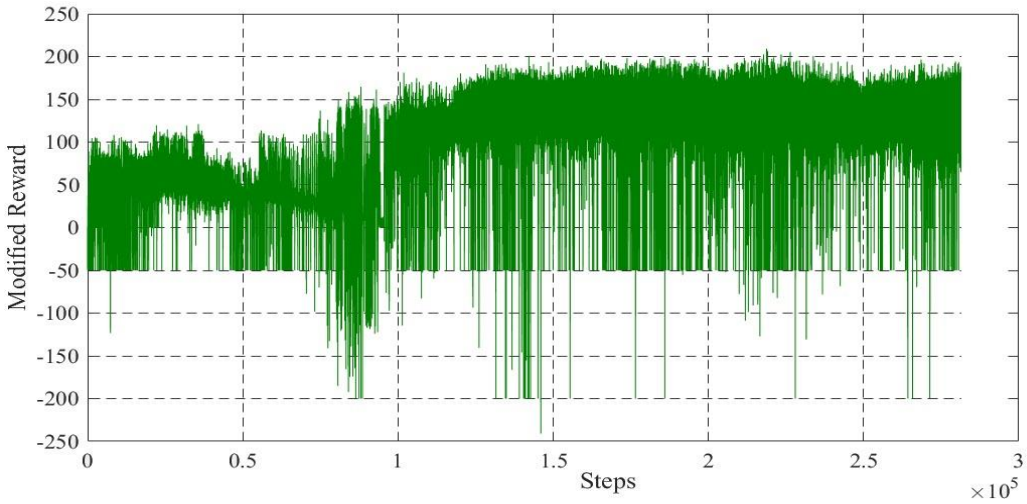

Figure 6.8 Reward obtained in training using modified reward function

The RL agent trained using first reward function was unable to drive on the test track and stopped part way through the testing. The reward of the modified model is comparatively higher than that of first one during testing phase. The valleys in Figure 6.10 are caused by the car trying to steer into the racetrack's flanks, the agent using modified reward takes the right action and swerving back to stay on track thereby increasing the reward. The RL agent trained using first reward function failed to complete an entire lap and hence has fewer steps. On the other hand, our modified model successfully completed an entire lap without crashing.

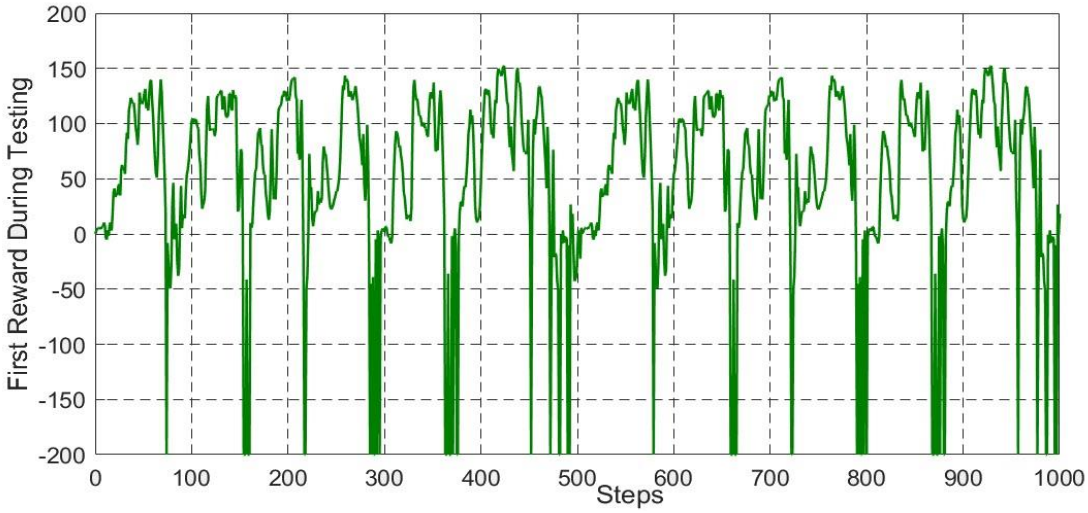

Figure 6.9 Reward obtained in testing using first reward function

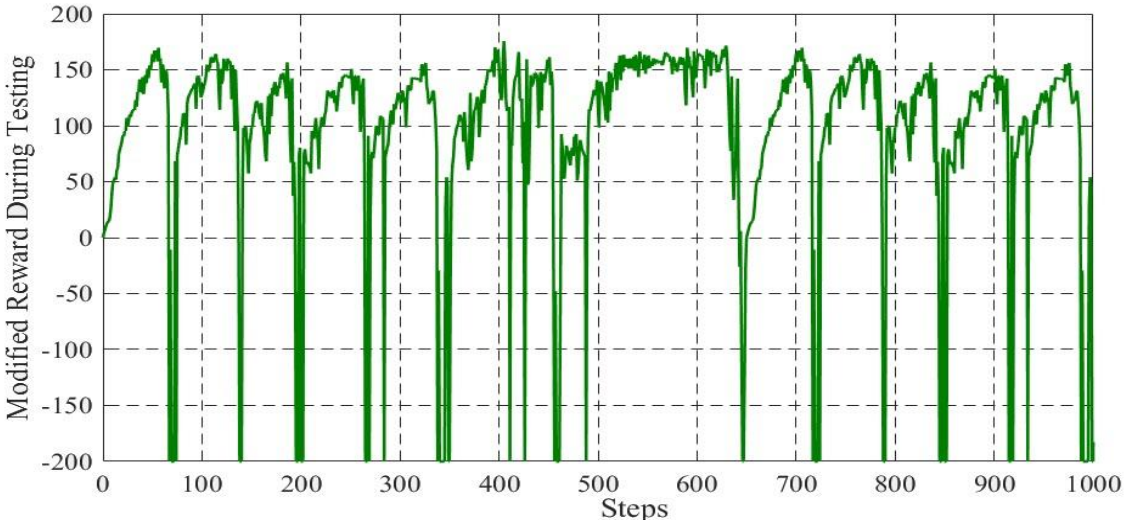

Figure 6.10 Reward obtained in testing using modified reward function

From the figures below, we see that the proposed model greatly improves the smoothness. It learns a more stable driving style not having to change direction very often which can lead to uncontrolled motion and reduced lap timing. On the other hand, the first reward model seems to change the driving direction very often – which is not desirable, the agent's steering fluctuates a lot even on the straight, compared to modified reward model.

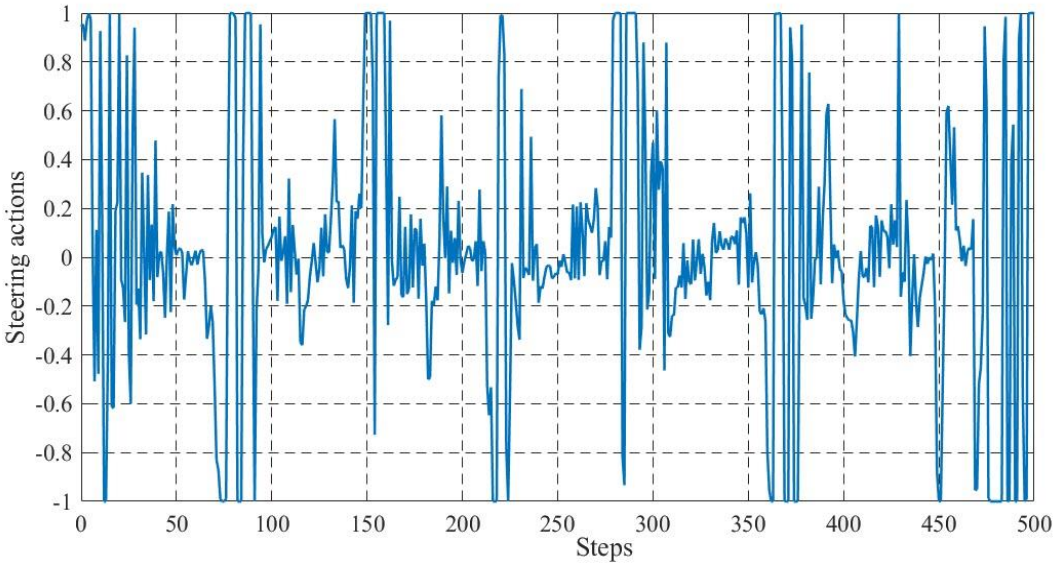

Figure 6.11 Normalized steering angle in testing using first reward function

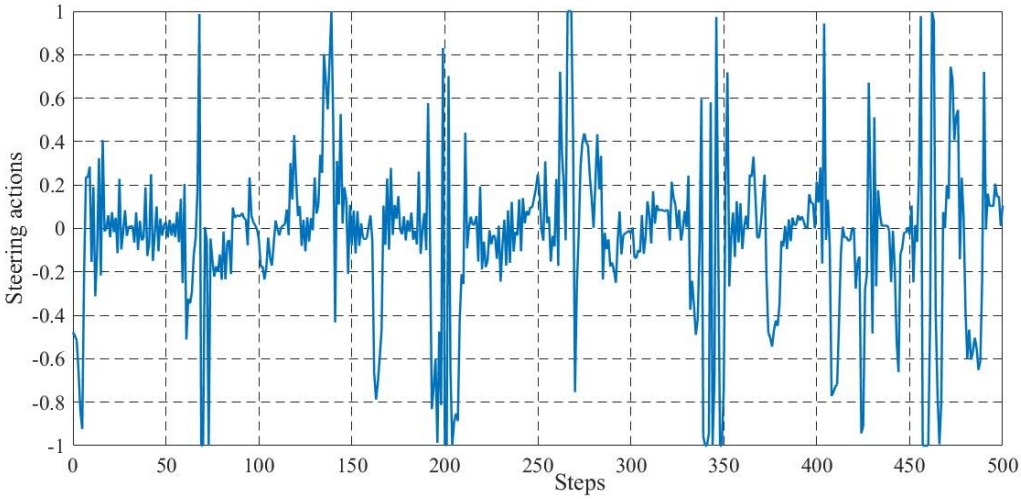

Figure 6.12 Normalized steering angle in testing using modified reward function

Chapter 7 : Conclusion and Future Work

In this chapter, we will deal with two main parts which are the conclusion from this thesis in addition to the future research work which could help in the extension of this thesis.

7.1 Conclusion

This research demonstrates some of RL algorithms that can autonomously drive in TORCS, with robustness over diverse environments including Alpine-1 and A-Speedway TORCS tracks. We can conclude that applying reinforcement learning is an efficient way to train an autonomous driving system. Some of these algorithms is applicable for discrete actions such as Q-learning and DQN algorithms that are not suitable in driving systems. There are RL algorithms that could be used in autonomous driving such as A3C and DDPG that provide continuous and smoother actions than Discrete Action Algorithms. Results showed that DDPG compared to Q-learning algorithm has an excellent performance on both straight and curved part of the track, the car takes continuous actions, so there is a smooth curve also, it is helpful when training cars to avoid obstacles.

We proposed a reward model by modifying the used reward function by adding additional penalty terms. The modified model showed faster and stable learning, making faster turns and performing well to new racetracks, it increased the average reward compared to the used one and improved the smoothness of the actions. We also applied stochastic braking that allow the car to brake 10% of the time during the exploration phase and the remaining 90% of the time we don’t let the car brake which lets it learn how to accelerate. The model avoids local minimum while training by allowing the car to acquire some non-zero velocity as well as helping the RL agent learn how to apply brakes which we believe is part of the essence of driving.

7.2 Future Work

The research done in this thesis can be extended simply as follows:

- Expand the action spaces in Q-Learning algorithm to larger action spaces, i.e., 32 action spaces instead of 16 with more computational power and trained agent.
- Training our DDPG model at different simulators like CARLA simulator, in real environment that has different types of vehicles, infrastructure and pedestrian.
- Test a variety of reward functions, exploration policies, and adaptive gradient descent strategies.
- Training using other Continuous Action Algorithms such as A3C algorithm and compare between them in order to get better results.
- Besides to CNN (Convolutional neural networks), RNN (recurrent neural network) can be implemented to process inputs of any length.
- Using deeper actor and critic networks without causing overfitting.

References

[1] Y. LeCun, Y. Bengio, and G. Hinton, “Deep learning. nature 521 (7553), 436-444,” Google Sch. Google Sch. Cross Ref Cross Ref, 2015.

[2] Y. Chen and Z. He, “Supervised-unsupervised combined neural learning for independent component analysis,” in Proceedings of the 9th International Conference on Neural Information Processing, 2002. ICONIP’02., 2002, vol. 3, pp. 1373–1377.

[3] B. R. Kiran et al., “Deep reinforcement learning for autonomous driving: A survey,” IEEE Trans. Intell. Transp. Syst., 2021.

[4] A. Geiger, P. Lenz, and R. Urtasun, “Are we ready for autonomous driving? the kitti vision benchmark suite,” in 2012 IEEE Conference on Computer Vision and Pattern Recognition, 2012, pp. 3354–3361.

[5] V. Mnih et al., “Playing atari with deep reinforcement learning,” arXiv Prepr. arXiv1312.5602, 2013.

[6] L. Pinto, M. Andrychowicz, P. Welinder, W. Zaremba, and P. Abbeel, “Asymmetric actor critic for image-based robot learning,” arXiv Prepr. arXiv1710.06542, 2017.

[7] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, “Trust region policy optimization,” in International conference on machine learning, 2015, pp. 1889–1897.

[8] Z. Wang et al., “Sample efficient actor-critic with experience replay,” arXiv Prepr. arXiv1611.01224, 2016.

[9] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, “CARLA: An open urban driving simulator,” in Conference on robot learning, 2017, pp. 1–16.

[10] B. Wymann, E. Espié, C. Guionneau, C. Dimitrakakis, R. Coulom, and A. Sumner, “Torcs, the open racing car simulator,” Softw. available http//torcs. sourceforge. net, vol. 4, no. 6, p. 2, 2000.

[11] D. Isele, R. Rahimi, A. Cosgun, K. Subramanian, and K. Fujimura, “Navigating occluded intersections with autonomous vehicles using deep reinforcement learning,” in 2018 IEEE International Conference on Robotics and Automation (ICRA), 2018, pp. 2034–2039.

[12] J. Michels, A. Saxena, and A. Y. Ng, “High speed obstacle avoidance using monocular vision and reinforcement learning,” in Proceedings of the 22nd international conference on Machine learning, 2005, pp. 593–600.

[13] L. García Cuenca, E. Puertas, J. Fernandez Andres, and N. Aliane, “Autonomous driving in roundabout maneuvers using reinforcement learning with Q-learning,” Electronics, vol. 8, no. 12, p. 1536, 2019.

[14] A. Ganesh, J. Charalel, M. Das Sarma, and N. Xu, “Deep Reinforcement Learning for Simulated Autonomous Driving,” 2016.

[15] D. Karavolos, “Q-learning with heuristic exploration in Simulated Car Racing,” 2013.

[16] N. Xu, B. Tan, and B. Kong, “Autonomous driving in reality with reinforcement learning and image translation,” arXiv Prepr. arXiv1801.05299, 2018.

[17] M. Jaritz, R. De Charette, M. Toromanoff, E. Perot, and F. Nashashibi, “End-to-end race driving with deep reinforcement learning,” in 2018 IEEE International Conference on Robotics and Automation (ICRA), 2018, pp. 2070–2075.

[18] A. El Sallab, M. Abdou, E. Perot, and S. Yogamani, “End-to-end deep reinforcement learning for lane keeping assist,” arXiv Prepr. arXiv1612.04340, 2016.

[19] X. Zong, G. Xu, G. Yu, H. Su, and C. Hu, “Obstacle avoidance for self-driving vehicle with reinforcement learning,” SAE Int. J. Passeng. Cars-Electronic Electr. Syst., vol. 11, no. 07-11-01–0003, pp. 30–39, 2017.

[20] F. Yang, P. Wang, and X. Wang, “Continuous Control in Car Simulator with Deep Reinforcement Learning,” in Proceedings of the 2018 2nd International Conference on Computer Science and Artificial Intelligence, 2018, pp. 566–570.

[21] A. Wagh, “Distributed Approach for implementation of A3C on TORCS,” 2019.

[22] D. Loiacono, A. Prete, P. L. Lanzi, and L. Cardamone, “Learning to overtake in TORCS using simple reinforcement learning,” in IEEE Congress on Evolutionary Computation, 2010, pp. 1–8.

[23] “ugo-nama-kun/gym_torcs.” https://github.com/ugo-nama-kun/gym_torcs (accessed May 16, 2021).

[24] R. S. Sutton and A. G. Barto, Reinforcement learning: An introduction. MIT press, 2018.

[25] C. J. C. H. Watkins and P. Dayan, “Q-learning,” Mach. Learn., vol. 8, no. 3–4, pp. 279–292, 1992.

[26] T. P. Lillicrap et al., “Continuous control with deep reinforcement learning.” Google Patents, Sep. 15, 2020.

[27] V. Mnih et al., “Human-level control through deep reinforcement learning,” Nature, vol. 518, no. 7540, pp. 529–533, 2015.

[28] “Using Keras and Deep Deterministic Policy Gradient to play TORCS | Ben Lau.” https://yanpanlau.github.io/2016/10/11/Torcs-Keras.html (accessed May 05, 2021).

[29] “Python Programming Tutorials.” https://pythonprogramming.net/q-learning-reinforcement-learning-python-tutorial/ (accessed Mar. 25, 2021).

[30] “Deep Reinforcement Learning Demysitifed (Episode 2) — Policy Iteration, Value Iteration and Q-learning | by Moustafa Alzantot | Medium.” https://medium.com/@m.alzantot/deep-reinforcement-learning-demysitifed-episode-2-policy-iteration-value-iteration-and-q-978f9e89ddaa (accessed Apr. 02, 2021).

[31] “Deep Q-Learning Demystified | Towards Data Science.” https://towardsdatascience.com/self-learning-ai-agents-part-ii-deep-q-learning-b5ac60c3f47 (accessed Apr. 12, 2021).

[32] M. Bojarski et al., “End to end learning for self-driving cars,” arXiv Prepr. arXiv1604.07316, 2016.

[33] “torcs › News.” http://torcs.sourceforge.net/ (accessed Mar. 17, 2021).

[34] D. Loiacono, L. Cardamone, and P. L. Lanzi, “Simulated Car Racing Championship: Competition Software Manual,” Apr. 2013, Accessed: Apr. 12, 2021. [Online]. Available: http://arxiv.org/abs/1304.1672.

[35] “Diving deeper into Reinforcement Learning with Q-Learning.” https://www.freecodecamp.org/news/diving-deeper-into-reinforcement-learning-with-q-learning-c18d0db58efe/ (accessed May 03, 2021).

[36] Q-learning algorithm on TORCS - YouTube. (n.d.). Retrieved May 2, 2021, from https://www.youtube.com/watch?v=dissvUj8hJ8&ab_channel=MinaMilad

[37] DDPG Reward1 on Train Track CG Speedway 1x speed - YouTube. (n.d.). Retrieved May 3, 2021, from https://www.youtube.com/watch?v=Ka1qTuL2Nzg&ab_channel=MinaMilad

[38] DDPG Reward 1 On Test Track Alpine 2x speed - YouTube. (n.d.). Retrieved May 3, 2021, from https://www.youtube.com/watch?v=9GgeXV0J5l4&ab_channel=MinaMilad

[39] DDPG Reward2 on Train Track CG Speedway 1x speed - YouTube. (n.d.). Retrieved May 3, 2021, from https://www.youtube.com/watch?v=8Ot3rcu-6Ik&ab_channel=MinaMilad

[40] DDPG Reward2 On Test Track Alpine - YouTube. (n.d.). Retrieved May 3, 2021, from https://www.youtube.com/watch?v=tjWznOgzAhw&ab_channel=MinaMilad

[41] V. Mnih et al., “Asynchronous methods for deep reinforcement learning,” in International conference on machine learning, 2016, pp. 1928–1937.

[42] C. Athanasiadis, D. Galanopoulos, and A. Tefas, “Progressive neural network training for the open racing car simulator,” in 2012 IEEE Conference on Computational Intelligence and Games (CIG), 2012, pp. 116–123.

[43] S. Wang, D. Jia, and X. Weng, “Deep reinforcement learning for autonomous driving,” arXiv Prepr. arXiv1811.11329, 2018.

[44] D. Sadigh, S. Sastry, S. A. Seshia, and A. D. Dragan, “Planning for autonomous cars that leverage effects on human actions.,” in Robotics: Science and Systems, 2016, vol. 2.

[45] D. Mehta, “State-of-the-Art Reinforcement Learning Algorithms,” B. Eng (Information Technol. Panjab Univ. Chandigarh, India, 2019.

[46] K. Güçkıran and B. Bolat, “Autonomous Car Racing in Simulation Environment Using Deep Reinforcement Learning,” in 2019 Innovations in Intelligent Systems and Applications Conference (ASYU), 2019, pp. 1–6.

[47] E. Barati and X. Chen, “An actor-critic-attention mechanism for deep reinforcement learning in multi-view environments,” arXiv Prepr. arXiv1907.09466, 2019.

[48] A. Jeerige, D. Bein, and A. Verma, “Comparison of Deep Reinforcement Learning Approaches for Intelligent Game Playing,” in 2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC), 2019, pp. 366–371, doi: 10.1109/CCWC.2019.8666545.

[49] D. Kamar, G. Akyol, A. Mertan, and A. İnceoğlu, “Comparative Analysis of Reinforcement Learning Algorithms on TORCS Environment,” in 2020 28th Signal Processing and Communications Applications Conference (SIU), 2020, pp. 1–4, doi: 10.1109/SIU49456.2020.9302358.

[50] Q. Zou, K. Xiong, and Y. Hou, “An end-to-end learning of driving strategies based on DDPG and imitation learning,” in 2020 Chinese Control And Decision Conference (CCDC), 2020, pp. 3190–3195, doi: 10.1109/CCDC49329.2020.9164410.

[51] S. Zuo, Z. Wang, X. Zhu, and Y. Ou, “Continuous reinforcement learning from human demonstrations with integrated experience replay for autonomous driving,” in 2017 IEEE International Conference on Robotics and Biomimetics (ROBIO), 2017, pp. 2450–2455, doi: 10.1109/ROBIO.2017.8324787.

[52] W. Xia, H. Li, and B. Li, “A Control Strategy of Autonomous Vehicles Based on Deep Reinforcement Learning,” in 2016 9th International Symposium on Computational Intelligence and Design (ISCID), 2016, vol. 2, pp. 198–201, doi: 10.1109/ISCID.2016.2054.

[53] Y. Zhang, P. Sun, Y. Yin, L. Lin, and X. Wang, “Human-like Autonomous Vehicle Speed Control by Deep Reinforcement Learning with Double Q-Learning,” in 2018 IEEE Intelligent Vehicles Symposium (IV), 2018, pp. 1251–1256, doi: 10.1109/IVS.2018.8500630.

[54] A. R. Fayjie, S. Hossain, D. Oualid, and D.-J. Lee, “Driverless Car: Autonomous Driving Using Deep Reinforcement Learning in Urban Environment,” in 2018 15th International Conference on Ubiquitous Robots (UR), 2018, pp. 896–901, doi: 10.1109/URAI.2018.8441797.

[55] Z. Huang, J. Zhang, R. Tian, and Y. Zhang, “End-to-End Autonomous Driving Decision Based on Deep Reinforcement Learning,” in 2019 5th International Conference on Control, Automation and Robotics (ICCAR), 2019, pp. 658–662, doi: 10.1109/ICCAR.2019.8813431.