# Designing Self-Driving Car based on DDPG & CNN DDPG - Reinforcement Learning

Tae H. Lee[1], Hyun J. Jung[1], Sung W. Lee[1], Seok I. Son[1]

Department of Mechanical Engineering, Yonsei University

***Abstract***

In this project, the goal is making the given vehicle to go through several laps while avoiding obstacles positioned randomly on an un-known track. The given information was one vehicle that had basic child-script and the track for practice-only. We trained the given vehicle by utilizing the policy-based reinforcement learning algorithm.

Since the vehicle has physical properties, we applied policy gradient Actor-Critic (DDPG) algorithm with OU-noise to our project. To prevent overfitting, we randomized tracks and obstacle's positions every episode. Lastly, we decided the Input function having variables such as distance, slope, velocity etc. and the Reward function having speed variables and obstacle detection variables making the reinforced learned vehicle able to detect obstacles and run through the test track more specifically.

And after the competition, we did another try with different special Architecture with CNN

## I. Introduction

### *I. Current status of AI & Effort for development*

**Until** now, Deep reinforcement learning has been applied to limited industry and fields. Representatively, there exist some application examples of it such as AlphaGo, automatic robots etc. It does not follow a known algorithm or environment-model, meaning the machine itself does experiential-learning (Trial & Error) through various factors such as action, state, reward etc. Based on these factors, in the future society it is expected that RL will be applied to various fields such as medical, control system, etc., which in these fields is difficult for people to visually judge or manually control.

In this paper, we made a self-driving car as a first step to learn reinforcement learning algorithms, using V-rep (Simulation program) and Python language tool in OS Ubuntu.

### *II. The goal of Reinforcement Learning*

The goal for Reinforcement learning is to obtain optimal policy. There are two main methods to reach the goal. The first one is Value-based RL, which calculates the policy gradient from the value-function to get the optimal policy. The opposite method is Policy-based RL, which parameterizes policy gradients as an explicit function therefore finding out the optimal policy via policy gradient function directly.

### *III. Outline of the Project*

In this project, the object that was getting reinforcement learning was the car. Considering a vehicle having physical properties such as mass inertia and continuous action, policy gradient Actor-Critic (DDPG, policy-based RL) with OU-Noise (as exploration process) would be good to be applied to obtain optimal policy gradient of objective function. Therefore, we constructed Architecture with anctor_net, critic_net in ddpg.py.

Input and Reward function are the key factors in order for our vehicle to detect states and perform properly on test environment-model. To make it detect obstacles and test track much more specifically, we put information such as (x, y) velocity, distance between car and track, slope of road etc into the input function and configured reward function as an equation for speed. Lastly, to prevent overfitting, we randomized track and obstacles position every episode by using generate_path, rand_int function.

### *IV. Resource*

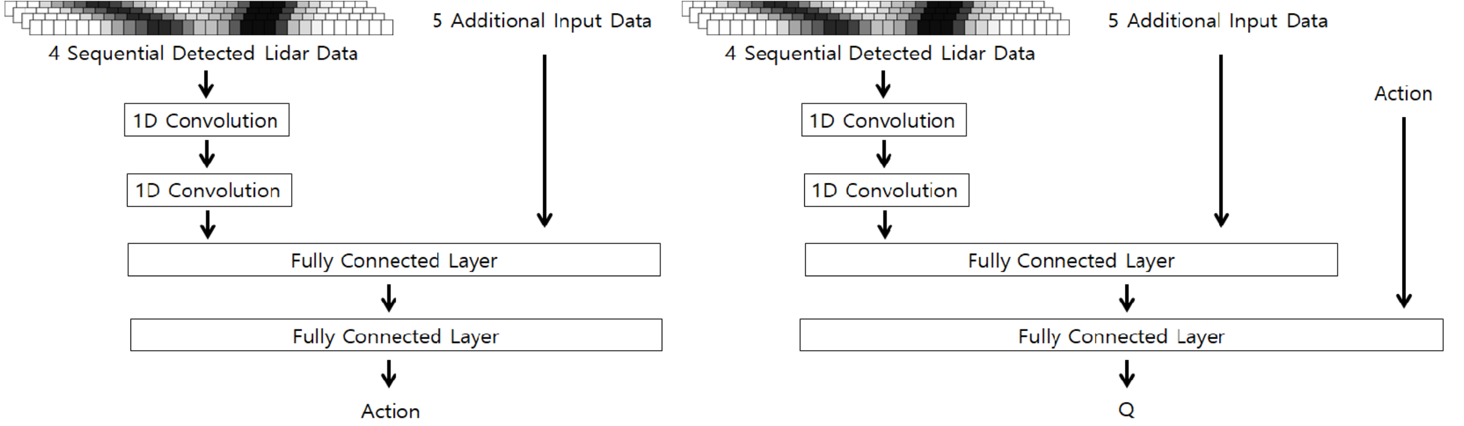
Computer specification:
[GPU: GeForce GTX 1050 Ti/PCIe/SSE2 Processor: Intel® Core™ i5-7600K CPU @ 3.80GHz × 4 Memory : 15.6 GiB]

## II. Method

### [Part1. Architecture of Algorithm]

A. Architecture of Algorithm (DDPG)

Architecture of our algorithm is called DDPG (Policy gradient actor-critic), which consists of actor_net, critic_net, etc. The




**[Figure 1. Architecture with CNN]**

reason why we selected this algorithm is that we must control a vehicle that has a continuous action and several action factors such as left, right wheel or steering actions.

In other words, it has high dimensions and continuous variables. To obtain optimal policy for continuous action, we needed to use off-policy learning that can get new state policy by referring previous policy. One of the off-policy learning which meets all conditions (high-Dimension, continuous action etc) is DDPG algorithm.

At DDPG, actor updates the policy parameter, and this updated policy is transferred to critic to evaluate whether the optimized policy or not by Q-function. After process, it updates the Q-function parameter. It takes place every action and process. and minibatch $N(s_i, a_i, r_i, s_{i+1})$ is set to 64.

Q-function and Policy gradient equations are below and:

$$Q^{\mu}(s_t, a_t) = \mathbb{E}_{r_t, s_{t+1} \sim E}[r(s_t, a_t) + \gamma Q^{\mu}(s_{t+1}, \mu(s_{t+1}))] - (1)$$

$$\nabla_{\theta^{\mu}} \mathcal{J} = \frac{1}{N} \sum_{i} \nabla_a Q(s, a|\theta^Q)|_{s = s_i, a = \mu(s_i)} \nabla_{\theta^{\mu}} \mu(s|\theta^{\mu})\Big|_{s_i} - (2)$$

And, furthermore, the discount factor is used to measure the return value that changes as the time-step passes. It has a value between 0 and 1. According to a well-known paper about RL, typically (0.1) value is used as discount factor.

### B. Architecture of Algorithm (CNN DDPG)

When compared to previous DDPG algorithm, as shown above Figure 1, there is a big difference which is that CNN layers inserted before the Fully-connected layers. Input data passed through CNN contain spatio-temporal features, meaning these make the agent consider spatial surrounding information to drive itself more specifically.

## [Part2. Components of Algorithm]

### 1 Environment

#### *I. Randomizing Tracks & Obstacles -notation*

In order for our reinforced learned vehicle to perform properly in an unknown environment-model, we had to prevent it from overfitting on the specific track. So as to block that kind of situation, we manipulated the learning environment using Bezier spline and the 'generate_path' function that generates paths randomly as setting 10 thousand points every episode on the simulation map.

Furthermore, for every episode, we set up moving and stationary obstacles to appear on the generated track randomly by utilizing the 'rand_int' function. In details, we made 2 types of moving obstacles. One kind was moving perpendicularly to the track as in the real world pedestrians crossing the street. The other kind was moving along the track like real cars racing down the street. To avoid situations where the obstacles block the vehicle's course on track, we manipulated the obstacles' orientation by calculating the slope of the track. In this way, we were able to prevent our car overfitting to only a specific track.

#### *II. Remote API functions (Operation mode)*

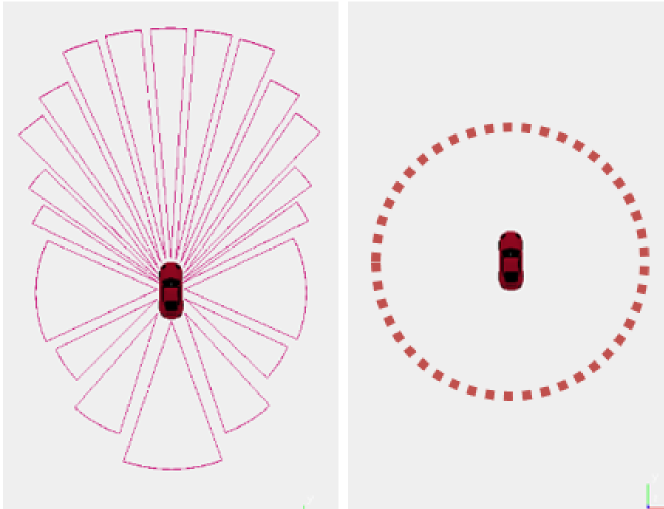
The training time is a very important factor in RL. The operation modes that are involved in training time are Streaming, Synchronous, Blocking function, and non-Blocking function. [2] When using the blocking or synchronous mode, buffering occurs since the client and server processes data before sending or receiving each other. This means that the total training time is prolonged by buffering. For this reason, we used streaming mode to get the data immediately whenever we needed a specific data.

### 2 Agent

#### *A. Sensor for DDPG algorithm*

For the sensor, we used the disk type proximity sensors instead of the ray proximity sensor. The reason we used the disk proximity sensor was because of the blind spot between the two ray proximity sensors. Since the ray types are straight lines, when the obstacle had distance from the car the area made by the ray between the arcs will increase, making the sensor unable to detect the obstacle. Since the disk type has continuous laser it will make it able for the car to detect the obstacles more accurately.

The operation principle of the disk type was to detect the closest distance from the obstacle. The car will treat this distance as input data, which is called minimum pooling. The minimum pooling will let the car extract one of the most



**[Figure 2 Agent model]**

important surrounding data for every disk types proximity sensors.

Total 20 disk types proximity sensors were installed all around the car but the lengths for each sensor was different. Since we want to give a different sensitivity to side and front of the agent. the proximity sensors on the side were much shorter than the LiDAR on the front as shown above.

*B. Sensor for CNN DDPG algorithm*

For the CNN DDPG, one channel LiDAR with 20 m diameter is adopted. And it can perceive 96 data at once a time.

### 3. Input Function

*A. Input data for DDPG algorithm*

As we mentioned before in the sensor part, each 20 data from the proximity sensors were used for detecting the obstacles more specifically.

And 20 data were stacked 3 times, which means the data from the previous steps were considered with the current proximity sensor data in order to consider previous spatial information; total 60 proximity data. In addition, 5 input data were used.

5 input data are as follow:

i. Distance between the car and the centerline
ii. Distance between the car and the boundary line.
iii. Angle between the car's direction and slope of the track.
iv. Velocity of the car's x direction.
v. Velocity of the car's y direction.

B. Input data for CNN DDPG algorithm

For the same reason, in this case, we have stacked 96 data four times; 376 LiDAR data. And 5 input data also were used.

C. Data normalization

Afterwards we normalized the data so as to prevent being stuck in the local optimum and train agent efficiently.

### 4. Reward Function

The main reward function are as follows.

$$R_{total} = R(Speed\ x) - [R(Crash) + R(LiDAR)]$$

The main components are as follows:

*i. R (speed x):*

This reward is given to the agent to move forward fast as much as possible on the track. Reward is simply proportional to speed x which is the speed for along the direction on the track.

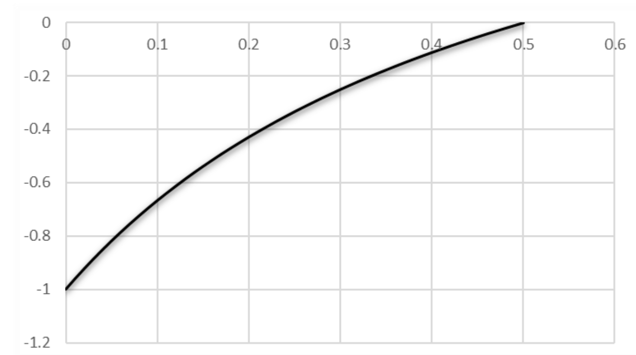
*ii. R (Crash):*

The crash reward is for avoiding crash with obstacles including guardrail. We set the crash reward as a high penalty value 20.

*iii. R (LiDAR):*

The reward function is shown below. we set the reward started to be given when the LiDAR length between the agent and the obstacles become half of the original length. Other than 0 to 0.5, reward was set to be 0 value.

$$y = \begin{cases} -\left(\dfrac{1}{x + 0.5}\right) + 1 & (0 \le x \le 0.5) \\ 0 & (x < 0, x > 0.5) \end{cases} \quad - (3)$$




**[Figure 3. Reward Function for the LiDAR]**

### 5. Action

*I. Exploration (OU-Noise)*

One problem generated by policy-gradient learning is that the agent's action can converge to a local optimum. It means there may be a more optimized policy that is appropriate for the environment, but the car's action is stuck in the local optimum and does not seek a better resolution; which is called global optimum.

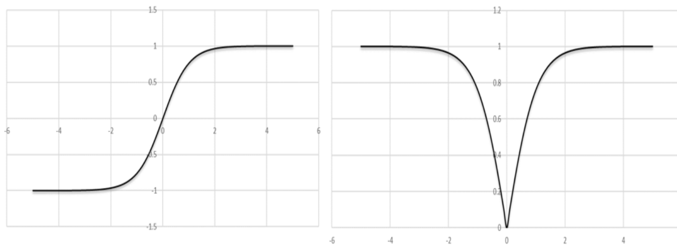
In order to avoid getting stuck in the local optimum, exploratory behavior needs to be led by parameter noise, the noise which is injected into the parameter space. Beyond many kinds of noise factor we've chosen Ornstein-Uhlenbeck process noise. (Matthias et al) [1] OU-Noise is a non-trivial solution that satisfies all of the gaussian process, Markov process, and temporally homogenous condition. That solution written as below

$$dx_t = \theta(\mu - x_t)dt + \sigma dW_t - (1)$$

This OU-process is suitable for the stochastic needed for free model and continuous-time process needed for inertial vehicle. The adjustable parameters in the OU noise (equation (1)) are

theta, mu, sigma. By adjusting the variables, we could obtain the optimized parameters such as mu=0, theta=0.15, Sigma=0.2.

### *II. Activation function (Throttle & Brake)*

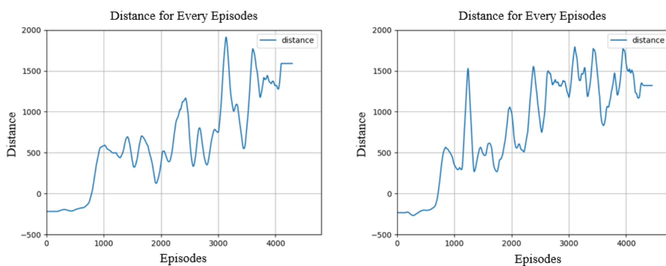


**[Figure 4. Throttle & Brake graph]**

For the agent action, there are two main actions to avoid obstacle effectively, one is throttling and the other is braking. Figure 4. is about speed with time when the agent throttles from stationary state or brake from running state. We used hyper tangent function as activation function (left side of Figure 4) But we modified negative action into positive action by applying the absolute function to it. (right side of Figure 4) [1] it can minimize the vanishing effect for the agent action. That is reason why we applied this modified function to our agent operation mechanism.

And about the steering part, we used the Ackerman steering which has the same curvature of the left, right wheel, which makes our car turn corner smoothly.

## III. RESULT




**[Figure 5. comparing results of two architectures]**

We trained total 4 thousand episodes which is equivalent to 300 thousand steps. And training simulation continue until the agent collide with obstacles or guardrail. Figure 5 is about the distance the car traveled till collision for every training episode. The algorithm used on the left graph is DDPG algorithm we used at competition. And the other side is CNN DDPG we applied newly.

| 50 episodes for the test | | |
|---|---|---|
| Architecture | DDPG | CNN DDPG |
| Distance | 1210 m | 1380 m |
| Difference rate (%) | 14.05 % | |

**[Table 1. Difference rate of distance of two architectures]**

After the training, we conducted the test for 50 episodes. Test simulation also continue till collision. As a result, the agent with DDPG algorithm moved 1210 m distance in average and CNN DDP algorithm moved 1380 m distance in average without collision. And difference rate of the distance is 14.05 %.

Based on the result of table 1. We could obtain the conclusion that CNN DDPG is more stable than DDPG algorithm.

## IV. CONCLUSION

In this project, we constructed two architectures of the algorithm which are DDPG & CNN DDPG for the self-driving car. We expected CNN DDPG to be more stable. This is because the data passed through CNN layers contain spatio-temporal features which are more about surrounding information. As a result, we checked each algorithm's performance as moving distance without collision and confirmed that the agent with CNN DDPG could go farther and more stables.

## V. REFERENCE

[1] Matthias Plappert, Rein Houthooft, Prafulla Dhariwal, Szymon Sidor, Richard Y. Chen, Xi Chen, Tamim Asfour, Pieter Abbeel, and Marcin Andrychowicz – Parameter space noise for exploration,Karlsruhe Institue of Technology (KIT)[1] University of California, Berkeley[2], ICLR 2018

[2] http://www.coppeliarobotics.com/helpFiles/ - V-REP User Manual version 3.5.0.

[3] Lei Tai, Giuseppe Paolo, Ming Liu – Virtual-to-real Deep Reinforcement Learning: Continuous Control of Mobile Robots for Maples Navigation, 1703.00420v4 21 July 2017.