Deep Reinforcement Learning for Autonomous Driving

Sen Wang
Carnegie Mellon University
5000 Forbes Ave,
Pittsburgh, PA 15213
senw@andrew.cmu.com

Daoyuan Jia
Carnegie Mellon University
5000 Forbes Ave,
Pittsburgh, PA 15213
daoyuanj@andrew.cmu.edu

Xinshuo Weng
Carnegie Mellon University
5000 Forbes Ave,
Pittsburgh, PA 15213
xinshuow@andrew.cmu.edu

Abstract

Reinforcement learning has steadily improved and outperform human in lots of traditional games since the resurgence of deep neural network. However, these success is not easy to be copied to autonomous driving because the state spaces in real world are extreme complex and action spaces are continuous and fine control is required. Moreover, the autonomous driving vehicles must also keep functional safety under the complex environments. To deal with these challenges, we first adopt the deep deterministic policy gradient (DDPG) algorithm, which has the capacity to handle complex state and action spaces in continuous domain. We then choose The Open Racing Car Simulator (TORCS) as our environment to avoid physical damage. Meanwhile, we select a set of appropriate sensor information from TORCS and design our own rewarder. In order to fit DDPG algorithm to TORCS, we design our network architecture for both actor and critic inside DDPG paradigm. To demonstrate the effectiveness of our model, We evaluate on different modes in TORCS and show both quantitative and qualitative results.

1 Introduction

Autonomous driving [10] is an active research area in computer vision and control systems. Even in industry, many companies, such as Google, Tesla, NVIDIA [3], Uber and Baidu, are also devoted to developing advanced autonomous driving car because it can really benefit human's life in real world. On the other hand, deep reinforcement learning technique has been successfully applied with great success to a variety of games [12] [13]. The success of deep reinforcement learning algorithm proves that the control problems in real-world environment could be naturally solved by optimizing policy-guided agents in high-dimensional state and action space. In particular, state spaces are often represented by image features obtained from raw images in vision control systems.

However, the current success achieved by deep reinforcement learning algorithms mostly happens in scenarios where controller has only discrete and limited action spaces and there is no complex content in state spaces of the environment, which is not the case when applying deep reinforcement learning algorithms to autonomous driving system. For example, there are only four actions in some Atari games such as SpaceInvaders and Enduro. For game Go, the rules and state of boards are very easy to understand visually even though spate spaces are high-dimensional. In such cases, vision problems are extremely easy to solve, then the agents only need to focus on optimizing the policy with limited action spaces. But for autonomous driving, the state spaces and input images from the environments contain highly complex background and objects inside such as human which can vary dynamically and behave unpredictably. These involve in lots of difficult vision tasks such as object detection, scene understanding, depth estimation. More importantly, our controller has to act correctly and fast in such difficult scenarios to avoid hitting objects and keep safe.

CMU 10703 Deep Reinforcement Learning and Control Course Project, (2017).

A straightforward way of achieving autonomous driving is to capture the environment information by using precise and robust hardwares and sensors such as Lidar and Inertial Measurement Unit (IMU). These hardware systems can reconstruct the 3D information precisely and then help vehicle achieve intelligent navigation without collision using reinforcement learning. However, there hardwares are very expensive and heavy to deploy. More importantly, they only tell us the 3D physical surface of the world instead of understanding the environment, which is not really intelligent. Both these reasons from hardware systems limit the popularity of autonomous driving technique.

One alternative solution is to combine vision and reinforcement learning algorithm and then solve the perception and navigation problems jointly. However, the perception problem is very difficult to solve because our world is extreme complex and unpredictable. In other words, there are huge variance in the world, such as color, shape of objects, type of objects, background and viewpoint. Even stationary environment is hard to understand, let alone the environment is changing as the autonomous vehicle is running. Meanwhile, the control problem is also challenging in real world because the action spaces is continuous and different action can be executed at the same time. For example, for smoother turning, We can steer and brake at the same time and adjust the degree of steering as we turn. More importantly, A safe autonomous vehicle must ensure functional safety and be able to deal with urgent events. For example, vehicles need to be very careful about crossroads and unseen corners such that they can act or brake immediately when there are children suddenly running across the road.

In order to achieve autonomous driving, people are trying to leverage information from both sensors and vision algorithms. Lots of synthetic driving simulators are made for learning the navigation policy without physical damage. Meanwhile, people are developing more robust and efficient reinforcement learning algorithm [18, 11, 20, 19, 2] in order to successfully deal with situations with real-world complexity. In this project, we are trying to explore and analyze the possibility of achieving autonomous driving within synthetic simulators.

In particular, we adopt deep deterministic policy gradient (DDPG) algorithm [9], which combines the ideas of deterministic policy gradient, actor-critic algorithms and deep Q-learning. We choose The Open Racing Car Simulator (TORCS) as our environment to train our agent. In order to learn the policy in TORCS, We first select a set of appropriate sensor information as inputs from TORCS. Based on these inputs, we then design our own rewarder inside TORCS to encourage our agent to run fast without hitting other cars and also stick to the center of the road. Meanwhile, in order to fit in TORCS environment, we design our own network architecture for both actor and critic used in DDPG algorithm. To demonstrate the effectiveness of our method, we evaluate our agent in different modes in TORCS, which contains different visual information.

2 Related Work

Autonomous Driving. Attempts for solving autonomous driving can track back to traditional control technique before deep learning era. Here we only discuss recent advances in autonomous driving by using reinforcement learning or deep learning techniques. Karavolos [7] apply the vanilla Q-learning algorithm to simulator TORCS and evaluate the effectiveness of using heuristic during the exploration. Huval2015 *et al.* [5] propose a CNN-based method to decompose autonomous driving problem into car detection, lane detection task and evaluate their method in a real-world highway dataset. On the other hand, Bojarski *et al.* [3] achieve autonomous driving by proposing an end to end model architecture and test it on both simulators and real-world environments. Sharifzadeh2016 *et al.* [17] achieve collision-free motion and human-like lane change behavior by using an inverse reinforcement learning approach. Different from prior works, Shalev-shwartz *et al.* [16] model autonomous driving as a multi-agent control problem and demonstrate the effectiveness of a deep policy gradient method on a very simple traffic simulator. Seff and Xiao [15] propose to leverage information from Google Map and match it with Google Street View images to achieve scene understanding prior to navigation. Recent works [22, 4, 6] are mainly focus on deep reinforcement learning paradigm to achieve autonomous driving. In order to achieve autonomous driving in th wild, You *et al.* [23] propose to achieve virtual to real image translation and then learn the control policy on realistic images.

Reinforcement Learning. Existing reinforcement learning algorithms mainly compose of value-based and policy-based methods. Vanilla Q-learning is first proposed in [21] and then become one of popular value-based methods. Recently lots of variants of Q-learning algorithm, such as DQN

[13], Double DQN [19] and Dueling DQN [20], have been successfully applied to a variety of games and outperform human since the resurgence of deep neural networks. By leveraging the advantage functions and ideas from actor-critic methods [8], A3C [11] further improve the performance of value-based reinforcement learning methods.

Different from value-based methods, policy-based methods learn the policy directly. In other words, policy-based methods output actions given current state. Silver *et al.* [18] propose a deterministic policy gradient algorithm to handle continuous action spaces efficiently without losing adequate exploration. By combining idea from DQN and actor-critic, Lillicrap *et al.* [9] then propose a deep deterministic policy gradient method and achieve end-to-end policy learning. Very recently, PGQL [14] is proposed and can even outperform A3C by combining off-policy Q-learning with policy gradient. More importantly, in terms of autonomous driving, action spaces are continuous and fine control is required. All these policy-gradient methods can naturally handle the continuous action spaces. However, adapting value-based methods, such as DQN, to continuous domain by discretizing continuous action spaces might cause curse of dimensionality and can not meet the requirements of fine control.

3 Methods

In autonomous driving, action spaces are continuous. For example, steering can vary from $-90°$ to $90°$ and acceleration can vary from 0 to 300km. This continuous action space will lead to poor performance for value-based methods. We thus use policy-based methods in this project. Meanwhile, random exploration in autonomous driving might lead to unexpected performance and terrible consequence. So we determine to use Deep Deterministic Policy Gradient (DDPG) algorithm, which uses a deterministic instead of stochastic action function. In particular, DDPG combines the advantages of deterministic policy gradient algorithm, actor-critics and deep Q-network.

In this section, we describe deterministic policy gradient algorithm and then explain how DDPG combines it with actor-critic and ideas from DQN together. Finally we explain how we fit our model in TORCS and design our reward signal to achieve autonomous driving in TORCS.

3.1 Deterministic Policy Gradient (DPG)

A stochastic policy can be defined as:

$$\pi_\theta = P[a|s;\theta] \tag{1}$$

Then the corresponding gradient is:

$$\nabla_\theta J(\pi_\theta) = E_{s\sim p^\pi, a\sim\pi_\theta}[\nabla_\theta log\pi_\theta(a|s)Q^\pi(s,a)] \tag{2}$$

This shows that the gradient is an expectation of possible states and actions. Thus in principle, in order to obtain an approximate estimation of the gradient, we need to take lots of samples from the action spaces and state spaces. Fortunately, mapping is fixed from state spaces to action spaces in deterministic policy gradient, so we do not need to integrate over whole action spaces. Thus deterministic policy gradient algorithm needs much fewer data samples to converge over stochastic policy gradient. Deterministic policy gradient is the expected gradient of the action-value function, so it can be estimated much efficiently than stochastic version.

In order to explore the environment, DPG algorithm achieves off-policy learning by borrowing ideas from actor-critic algorithms. An overall work flow of actor-critic algorithms is shown in Figure 1. In particular, DPG composes of an actor, which is the policy to learn, and a critic, which estimating Q value function. Essentially, the actor produces the action a given the current state of the environment s, while the critic produces a signal to criticizes the actions made by the actor. Then the critic is updated by TD learning and the actor is updated by policy gradient. Assume the function parameter for critic is w and the function parameter for Actor is θ, the gradient for deterministic policy is:

$$\nabla_\theta J(\mu_\theta) = E_{s\sim p^\mu}[\nabla_\theta\mu_\theta(s)\nabla_a Q^\mu(s,a)|a=\mu_\theta(s)] \tag{3}$$

For exploration stochastic policy β and off deterministic policy $\mu_\theta(s)$, we can derive the off-policy policy gradient:

$$\nabla_\theta J_\beta(\mu_\theta) = E_{s\sim p^\beta}[\nabla_\theta\mu_\theta(s)\nabla_a Q^\mu(s,a)|a=\mu_\theta(s)] \tag{4}$$

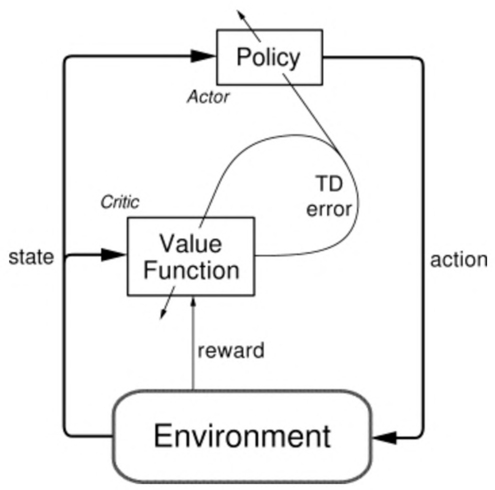


Figure 1: Overall work flow of actor-critic paradigm.

Notice that the formula does not have importance sampling factor. The reason is that the importance sampling is to approximate a complex probability distribution with a simple one. But the output of the policy here is a value instead of a distribution. The Q value in the formula corresponds to the critics and is updated by TD(0) learning. Given the policy gradient direction, we can derive the update process for Actor-Critic off-policy DPG:

$$\delta_t = r_t + \gamma Q^w(s_{t+1}, \mu_\theta(s_{t+1})) - Q^w(s_t, a_t) \tag{5}$$

$$w_{t+1} = w_t + \alpha_w \delta_t \nabla_w Q^w(s_t, a_t) \tag{6}$$

$$\theta_{t+1} = \theta_t + \alpha_\theta \nabla_\theta \mu_\theta(s_t) \nabla_a Q^w(s_t, a_t)|a = \mu_\theta(s) \tag{7}$$

3.2 Deep Deterministic Policy Gradient (DDPG)

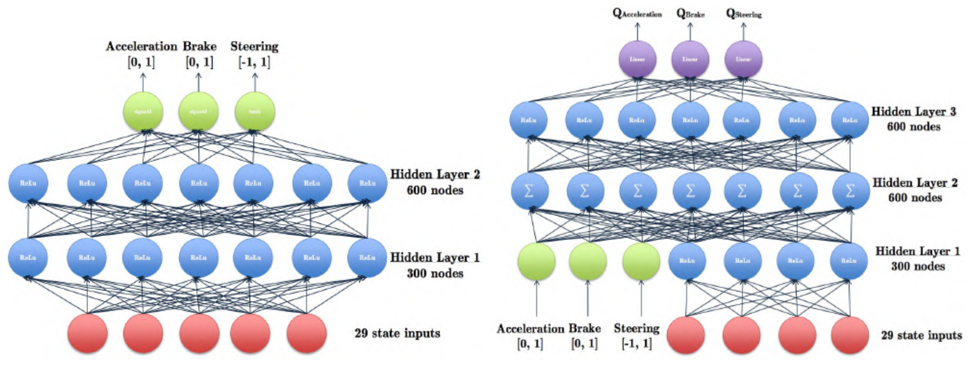
DDPG algorithm mainly follow the DPG algorithm except the function approximation for both actor and critic are represented by deep neural networks. Instead of using raw images as inputs, Torcs supports various type of sensor input other than images as observation. Here, we chose to take all sensor input listed in Table 1, make it a 29 dimension vector. The action of the model is a 3 dimension vector for **Acceleration** (where 0 means no gas, 1 means full gas), **Brake** (where 0 means no brake, 1 full brake) and **Steering** (where -1 means max right turn and +1 means max left turn) respectively.

The whole model is composed with an actor network and a critic network and is illustrated in Figure 2. The actor network serves as the policy, and will output the action. Both hidden layers are comprised of ReLU activation function. The critic model serves as the Q-function, and will therefore take action and observation as input and output the estimation rewards for each of action. In the network, both previous action the actions are not made visible until the second hidden layer. The first and third hidden layers are ReLU activated, while the second merging layer computes a point-wise sum of a linear activation computed over the first hidden layer and a linear activation computed over the action inputs.

Meanwhile, in order to increase the stability of our agent, we adopt experience replay to break the dependency between data samples. A target network is used in DDPG algorithm, which means we create a copy for both actor and critic networks. Then these target networks are used for providing target values. The weights of these target networks are then updated in a fixed frequency. For actor and critic network, the parameter w and θ are updated respectively by:

$$\theta^{'} = \tau\theta + (1 - \tau)\theta^{'} \tag{8}$$

$$w^{'} = \tau w + (1 - \tau)w^{'} \tag{9}$$



(a) Actor Network (b) Critic Network

Figure 2: Actor and Critic network architecture in our DDPG algorithm.

Name	Range(Unit)
ob.angle	[-pi,pi]
ob.track	(0,200)(meters)
ob.trackPos	$(-\inf, \inf)$
ob.speedX	$(-\inf, \inf)$(km/h)
ob.speedY	$(-\inf, \inf)$(km/h)
ob.speedZ	$(-\inf, \inf)$(km/h)

Table 1: Selected Sensor Inputs.

3.3 The Open Racing Car Simulator (TORCS)

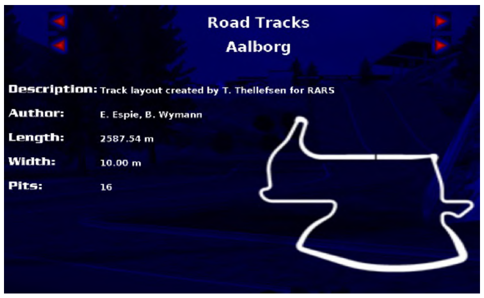
TORCH provides 18 different types of sensor inputs. After experiments we carefully select a subset of inputs, which is shown in Table 1.

- ob.angle is the angle between the car direction and the direction of the track axis. It reveals the car's direction to the track line.
- ob.track is the vector of 19 range finder sensors: each sensor returns the distance between the track edge and the car within a range of 200 meters. It let us know if the car is in danger of running into obstacle.
- ob.trackPos is the distance between the car and the track axis. The value is normalized w.r.t. to the track width: it is 0 when the car is on the axis, values greater than 1 or -1 means the car is outside of the track. We want the distance to the track axis to be 0.
- ob.speedX, ob.speedY, ob.speedZ is the speed of the car along the longitudinal axis of the car (good velocity), along the transverse axis of the car, and along the Z-axis of the car. We want the car speed along the axis to be high and speed vertical to the axis to be low.

Reward Design TORCS does not have internal rewarder, so we need to design our own reward function. The reward should not only encourage high speed along the track axis, but also punish speed vertical to the track axis as well as deviation from the track. We formulate our reward function as follows:

$$R_t = V_x cos(\theta) - \alpha V_x sin(\theta) - \gamma |trackPos| - \beta V_x |trackPos| \tag{10}$$

$V_x cos(\theta)$ denotes the speed along the track, which should be encouraged. $V_x sin(\theta)$ denotes the speed vertical to the track. |trackPos| measures the distance between the car and the track line. Both $|trackPos|$ and $V_x|trackPos|$ punish the agent when the agent deviates from center of the road. α, β, γ denote the weight for each reward term respectively.


Road Tracks
Aalborg
Description: Track layout created by T. Thellefsen for RARS
Author: E. Espie, B. Wymann
Length: 2587.54 m
Width: 10.00 m
Pits: 16

(a) Map Aalborg

(b) Training Mode: shaky at beginning of training

(c) Compete Mode: falling behind at beginning

(d) Compete Mode: Overtake other cars at corner

Figure 3: Train and evaluation on map Aalborg

4 Experiments

4.1 Experimental Setting

We experiment the simple actor-critic algorithm on TORCS engine, and asynchronous actor-critic algorithm on OpenAI Universe. We choose TORCS as the environment for TORCS, since it was wrapped OpenAI-compatible interfaces. Other softwares include Anaconda 2.7, Keras and Tensorflow v0.12. All our experiments were made on an Ubuntu 16.04 machine, with 12 cores CPU, 64GB memory and 4 GTX-780 GPU (12GB Graphic memory in total).

We adapt the implementation and hyper-parameter choice from [1]. Specifically, the replay buffer size is 100000 state-action pairs, with a discount factor of $\gamma = 0.99$. The optimizer is Adam with learning rates of 0.0001 and 0.001 for the actor and critic respectively, and a batch-size of 32. Target networks are updated gradually with $\tau = 0.001$.

4.2 Experiment Analysis

The TORCS engine contains many different modes. We can generally categorize them into two types: training mode and compete mode. In training mode, no other competitors in the view, and the view-angle is first-person as in Figure 3b. In compete mode, we can add other computer-controlled AI into the game and racing with them, as shown in Figure 3c. Notably, the existence of other competitors will affect the sensor input of our car.

We train the game with about 200 episodes on map Aalborg in train mode, and evaluate the game in compete mode with 9 other competitors. Each episode terminates when the car rush out of the track or when the car orientated to the opposite direction. Therefore, the length of each episode is highly variated, and therefore a good model could make one episode infinitely. Thus, we also set the maximum length of one episode as 60000 iterations. The map is shown in Figure 3a. In the train mode, the model is shaky at beginning, and bump into wall frequently (Figure 3b), and gradually stabilize as training goes on. In evaluation (compete mode), we set our car ranking at 5 at beginning among all competitors. Therefore, our car fall behind 4 other cars at beginning (Figure 3c). However, as the race continues, our car easily overtake other competitors in turns, shown in Figure 3d.

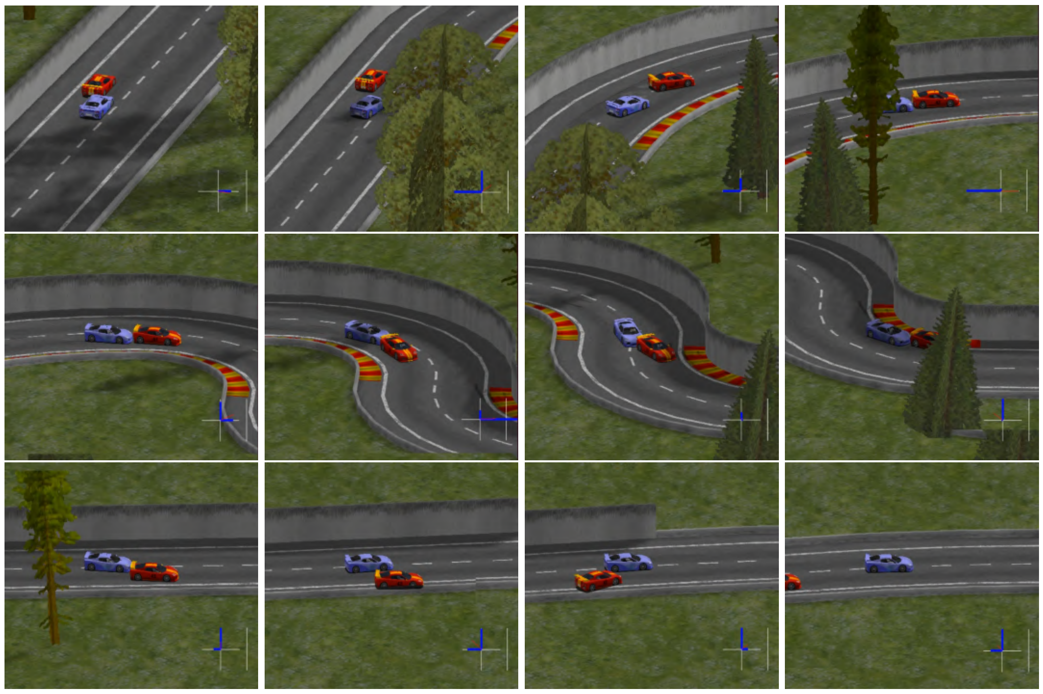


Figure 4: Compete Mode: our car (blue) over take competitor (orange) after a S-curve. For a complete video, please visit `https://www.dropbox.com/s/balm1vlajjf50p6/drive4.mov?dl=0`.

We illustrate 12 consecutive images in Figure 4 to show how our agent (blue) round a s-curve and how the overtake happens. Notably, TORCS has embedded a good physics engine and models vehicle drifting when the speed is fast. We found that the drifting is the main reason of driving in wrong direction after passing a corner and causes terminating the episode early. To deal with this issue, our agent has to decrease the speed before turning, either by hitting the brake or releasing the accelerator, which is also how people drive in real life. After training, we found our model do learned to release the accelerator to slow down before the corner to avoid drifting. Also, from Figure 4 we can find that our model did not learn how to avoid collision with competitors. This is because in training mode, there is no competitors introduced to the environment. Therefore, even our car (blue) can passing the s-curve much faster than the competitor (orange), without actively making a side-over-take, our car got blocked by the orange competitor during the s-curve, and finished the overtaking after the s-curve. We witnessed lots of overtakes near or after turning points, this indicates our model works better in dealing with curves. Usually after one to two circles, our car took the first place among all competitors. We uploaded the complete video at Dropbox.

We plot the performance of the model during the training in Figure 5, which contains 3 sub-figures and we refer them from top to bottom as (top), (mid), (bottom). The x-axis of all 3 sub-figures are aligned episodes of training.

In Figure 5(top), the mean speed of the car (km/h) and mean gain for each step of each episodes were plotted. Specifically, speed of the car is only calculated the speed component along the front direction of the car. In other words, drifting speed is not counted. The gain for each step is calculated with eq.(10). From the figure, as training went on, the average speed and step-gain increased slowly, and stabled after about 100 episodes. This indicates the training actually get stabled after about 100 episodes of training. Apart from that, we also witnessed simultaneously drop of average speed and step-gain. This is because even after the training is stale, the car sometimes could also rushed out of track and got stuck. Normally, when the car rushed out of the track, the episode should terminate. However, it did not guarantee successful termination every time, and this might because of imprecise detection of this out-of-track in TORCS. When the stuck happens, the car have 0 speed till and stuck up to 60000 iterations, and severely decreased the average speed and step-wise gain of this episode. Also, lots of junk history from this episode flush the replay buffer and unstabilized the training. Since this problem originates in the environment instead of in the learning algorithm, we did not spent too

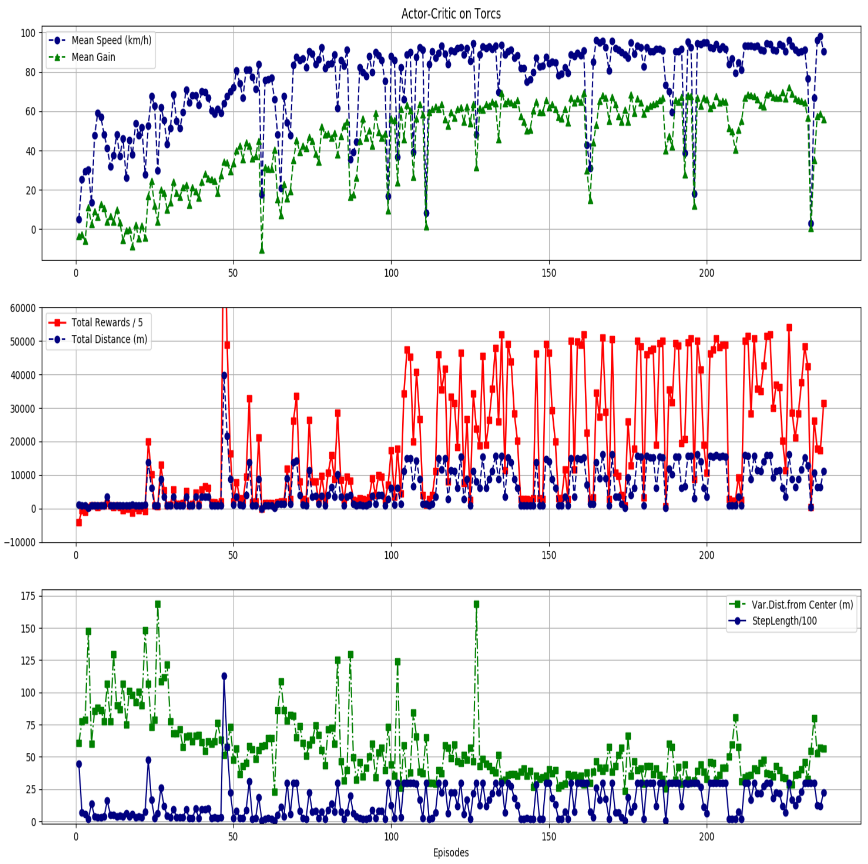


Figure 5: Model performance in episodes

much time to fix it, but rather terminated the episode and continue to next one manually if we saw it happen.

In Figure 5(mid), we plot the total travel distance of our car and total rewards in current episode, against the index of episodes. Intuitively, we can see that as training continues, the total reward and total travel distance in one episode is increasing. This is because the model was getting better, and less likely crash or run out track. Ideally, if the model is optimal, the car should run infinitely, and the total distance and total reward would be stable. However, because of the same reason we mention above, we constantly witness the sudden drop. Notably, most of the "drop" in "total distance" are to the same value, this proves for many cases, the "stuck" happened at the same location in the map.

In Figure 5(bottom), we plot the variance of distance to center of track (Var.Dist.from.Center(m)), and step length of one episode. The variance of distance to center of the track measures how stable the driving is. We show that our trained agent often drives like a "drunk" driver, in 8-shape, at the beginning, and gradually drives better in the later phases. The Var.Dist.from.Center curve decreases and stabilizes after about 150 episodes. This indicates the our model still drive unstably after 100 episodes, when the speed and episode rewards already get stabilized. So the extra 50 episodes of training stabilize the training.

5 Conclusion

In order to bridge the gap between autonomous driving and reinforcement learning, we adopt the deep deterministic policy gradient (DDPG) algorithm to train our agent in The Open Racing Car

Simulator (TORCS). In particular, we select appropriate sensor information from TORCS as our inputs and define our action spaces in continuous domain. We then design our rewarder and network architecture for both actor and critic inside DDPG paradigm. We demonstrate that our agent is able to run fast in the simulator and ensure functional safety in the meantime.

References

[1] Using keras and deep deterministic policy gradient to play torcs `https://yanpanlau.github.io/2016/10/11/Torcs-Keras.html`.

[2] P. Abbeel, P. Cs, and B. Edu. Benchmarking deep reinforcement learning for continuous control. *ICML*, 2016.

[3] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, et al. End to end learning for self-driving cars. *arXiv preprint*, 2016.

[4] H. Chae, C. M. Kang, B. Kim, J. Kim, C. C. Chung, and J. W. Choi. Autonomous Braking System via Deep Reinforcement Learning. *arXiv preprint*, 2017.

[5] B. Huval, T. Wang, S. Tandon, J. Kiske, W. Song, J. Pazhayampallil, M. Andriluka, P. Rajpurkar, T. Migimatsu, R. Cheng-yue, F. Mujica, A. Coates, and A. Y. Ng. An empirical evaluation of deep learning on highway driving. *arXiv preprint*, 2015.

[6] D. Isele, A. Cosgun, K. Subramanian, and K. Fujimura. Navigating Intersections with Autonomous Vehicles using Deep Reinforcement Learning. *arXiv preprint*, 2017.

[7] D. Karavolos. Q-learning with heuristic exploration in Simulated Car Racing. 2013.

[8] V. R. Konda and J. N. Tsitsiklis. Actor-Critic Algorithms. *NIPS*, 1999.

[9] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint*, 2015.

[10] Q. Memon, M. Ahmed, and S. Ali. Self-Driving and Driver Relaxing Vehicle. 2016.

[11] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *ICML*, 2016.

[12] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *NIPS*, 2013.

[13] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 2015.

[14] B. O'Donoghue, R. Munos, K. Kavukcuoglu, and V. Mnih. PGQ: Combining policy gradient and Q-learning. *ICLR*, 2016.

[15] A. Seff and J. Xiao. Learning from Maps : Visual Common Sense for Autonomous Driving. *arXiv preprint*, 2016.

[16] S. Shalev-shwartz, S. Shammah, and A. Shashua. Safe, Multi-Agent, Reinforcement Learning for Autonomous Driving. *arXiv preprint*, 2016.

[17] S. Sharifzadeh, I. Chiotellis, R. Triebel, and D. Cremers. Learning to drive using inverse reinforcement learning and deep q-networks. *arXiv preprint*, 2016.

[18] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller. Deterministic policy gradient algorithms. *ICML*, 2014.

[19] H. Van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. *AAAI*, 2016.

[20] Z. Wang, T. Schaul, M. Hessel, H. van Hasselt, M. Lanctot, and N. de Freitas. Dueling network architectures for deep reinforcement learning. *arXiv preprint*, 2015.

[21] C. J. C. H. Watkins. *Learning from delayed rewards*. PhD thesis, University of Cambridge England, 1989.

[22] S. Yang, W. Wang, C. Liu, K. Deng, and J. K. Hedrick. Feature Analysis and Selection for Training an End-to-End Autonomous Vehicle Controller Using the Deep Learning Approach. *arXiv preprint*, 2017.

[23] Y. You, Z. Wang, and C. Lu. Virtual to Real Reinforcement Learning for Autonomous Driving. *arXiv preprint*, 2017.