金陵科技學院

毕 业 设 计（论 文）外 文 参 考 资 料 及 译 文

译 文 题 目： Deep Reinforcement Learning for Autonomous Driving

深度强化学习在自主驾驶中的应用

学生姓名： 王家骅 **学　　号：** 1613902007

专　　业： 计算机科学与技术

所在学院： 计算机工程学院

指导教师： 龚如宾

职　　称： 讲师

2019 年 12 月 20 日

Deep Reinforcement Learning for Autonomous Driving

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.

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 RelatedWork

AutonomousDriving. 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 valuebased 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 DeterministicPolicyGradient(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 μθ(s), we can derive the off-policy policy gradient:

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

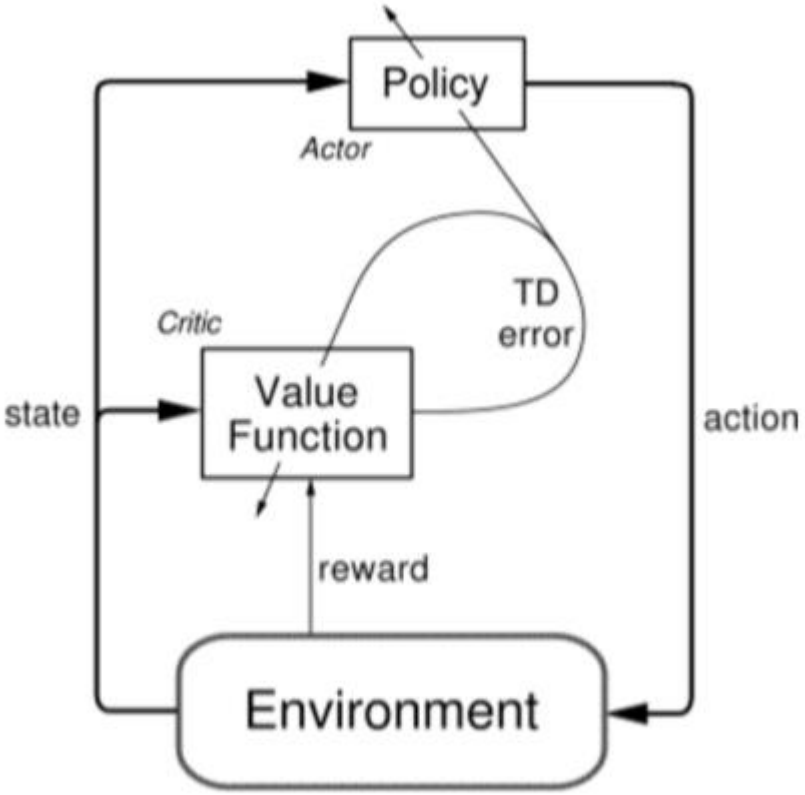


Figure 1: Overall work flow of actor-critic paradigm.

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:

$$\delta_t = r_t + \gamma Q^w(s_{t+1}, \mu_\theta(s_{t+1})) - Q^w(s_t, a_t) \tag{5}$$
$$w_{t+1} = w_t + \alpha_w \delta_t \nabla_w Q^w(s_t, a_t) \tag{6}$$
$$\theta_{t+1} = \theta_t + \alpha_\theta \nabla_\theta \mu_\theta(s_t) \nabla_a Q^w(s_t, a_t) | a = \mu_\theta(s) \tag{7}$$

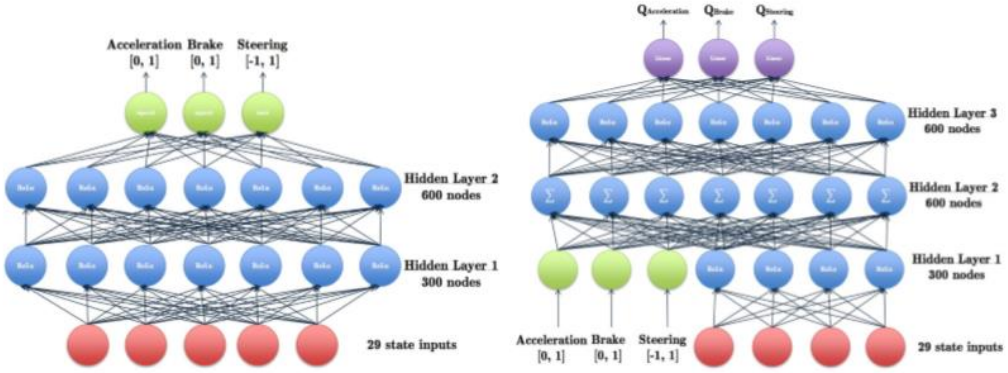
3.2 DeepDeterministicPolicyGradient(DDPG)

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.

Thewholemodeliscomposedwithanactornetworkandacriticnetworkandisillustrated inFigure2. 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 TheOpenRacingCarSimulator(TORCS)

TORCS 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.

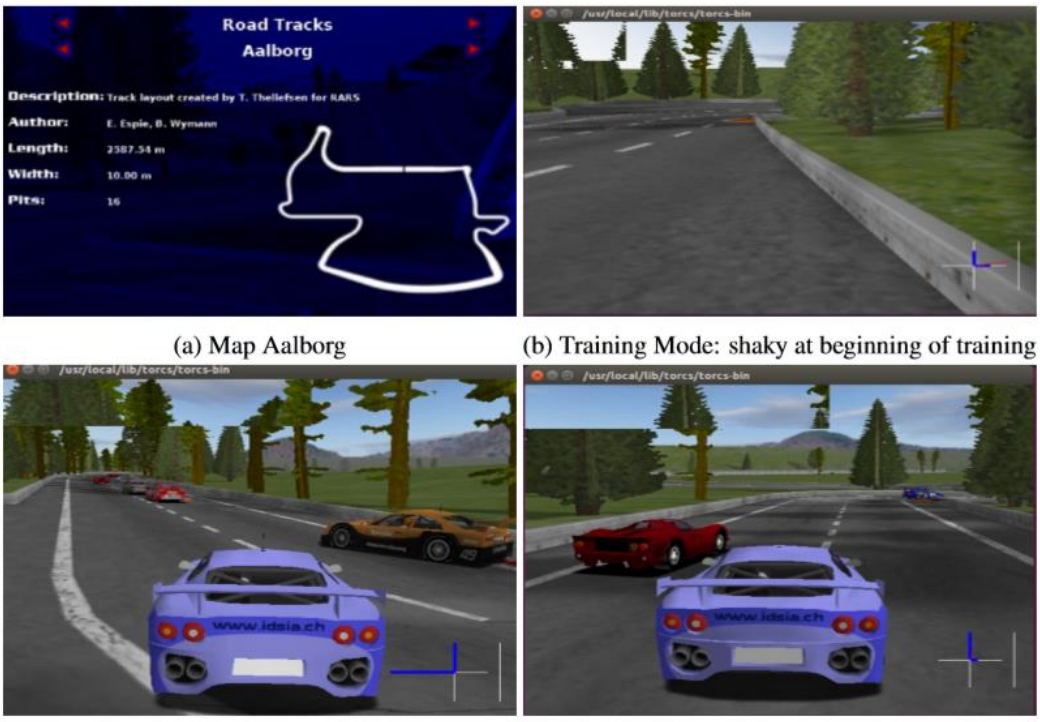
ob.speedX, ob.speedY, ob.speedZ is the speed of the car along the longitudinal axis of the car (good velocity), along the transverse axis of the car, and along the Z-axis of the car. We want the car speed along the axis to be high and speed vertical to the axis to be low.

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:

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

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



(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

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 Open AI-compatible interfaces. Other softwares include Anaconda2.7, 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 τ = 0.001.

4.2 ExperimentAnalysis

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.

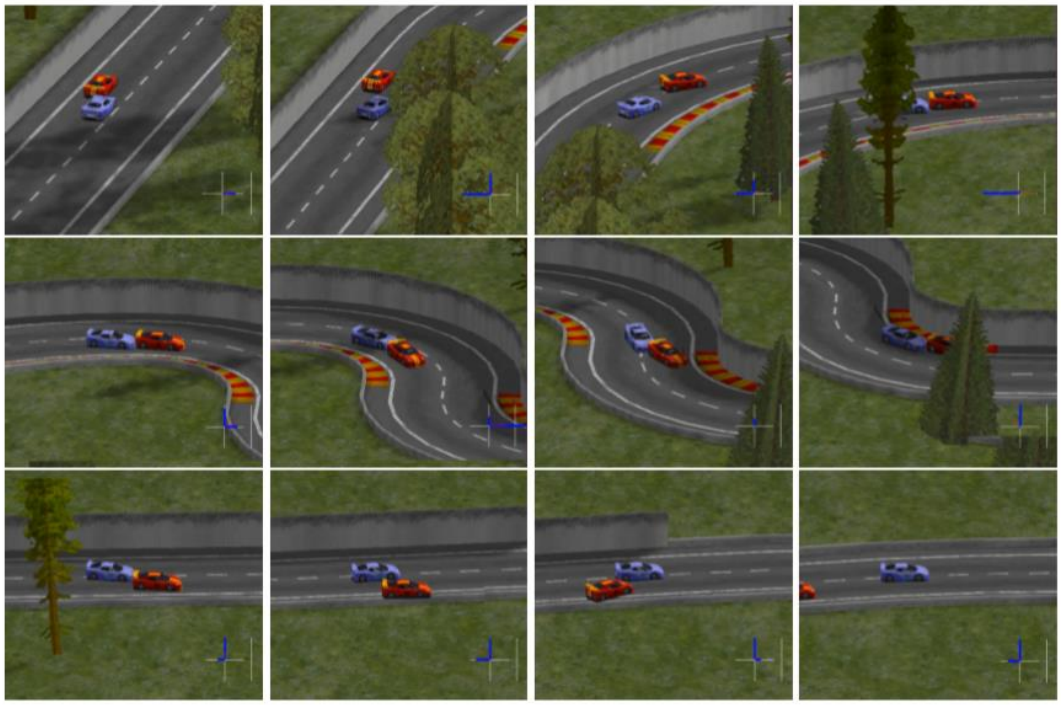


Figure 4: Compete Mode: our car (blue) over take competitor (orange) after a S-curve. For a complete video, please visit `https://www.dropbox.com/s/balm1vlajjf50p6/drive4.mov?dl=0`.

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 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.

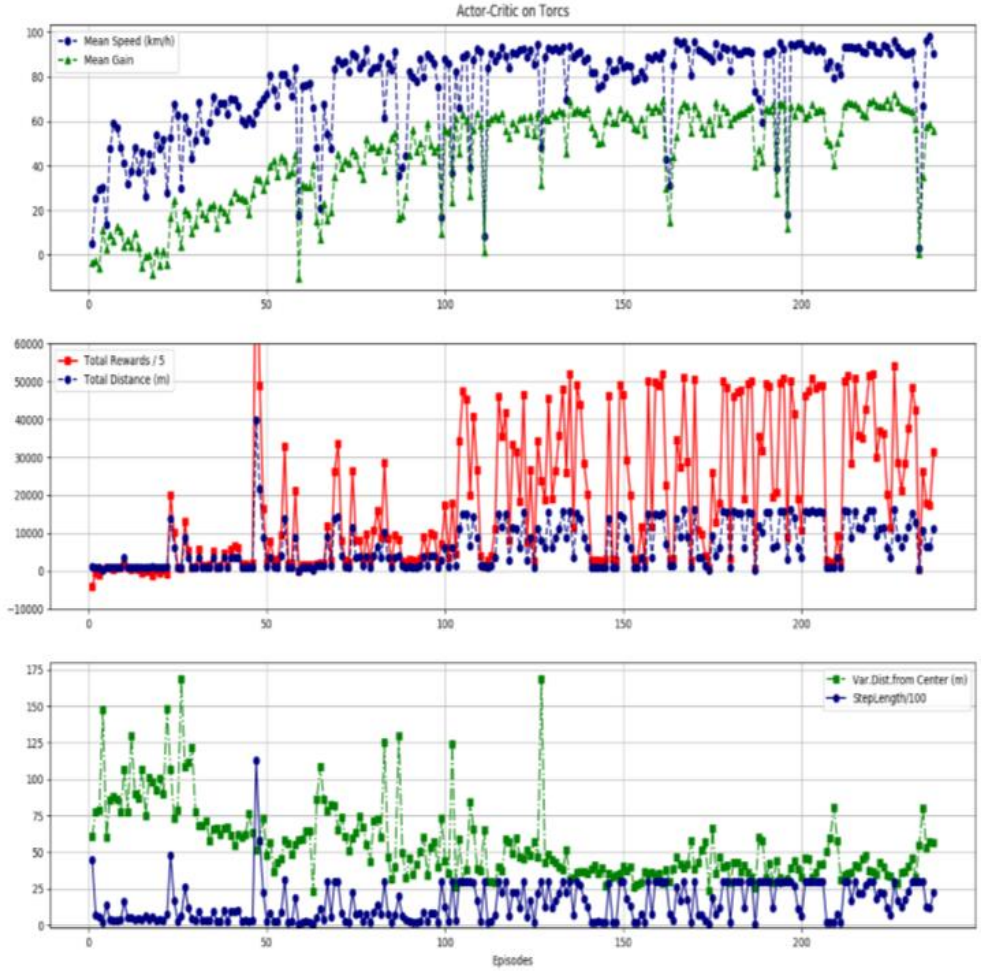


Figure 5: Model performance in episodes

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.

深度强化学习在自主驾驶中的应用

摘要

自从深度神经网络的兴起以来，强化学习一直在稳步发展，并且在许多传统游戏中超越人类。但是，这些成功并不容易复制到自动驾驶上，因为现实世界中的状态空间极其复杂，动作空间是连续的，并且需要精细控制。此外，自动驾驶车辆还必须在复杂环境下保持功能安全。为了应对这些挑战，我们首先采用深度确定性策略梯度（DDPG）算法，该算法具有处理连续域中复杂状态和动作空间的能力。然后，我们选择“开放赛车模拟器”（TORCS）作为我们的环境，以避免物理损坏。同时，我们从 TORCS 中选择了一组合适的传感器信息，并设计了自己的奖励器。为了使 DDPG 算法适合 TORCS，我们为 DDPG 中的 actor 和 critic 都设计了网络结构。为了证明我们模型的有效性，我们评估了 TORCS 中不同的模式，并显示了定量和定性的结果。

1 介绍

自动驾驶[10]是计算机视觉和控制系统中一个活跃的研究领域。即使在工业界，谷歌，特斯拉，英伟达[3]，优步和百度等许多公司也致力于开发先进的自动驾驶汽车，因为它确实可以改善现实世界中的人类生活。另一方面，深度强化学习技术已成功应用于各种游戏[12] [13]。深度强化学习算法的成功证明了通过在高维状态和动作空间中优化策略指导主体，可以自然地解决现实环境中的控制问题。因为在大多数情况中，状态空间通常由从视觉控制系统中的原始图像获得的图像特征来表示。

然而，目前通过深度强化学习算法获得的成功大多发生在以下情况下：控制器在环境状态空间中仅具有离散且有限的动作空间，并且没有复杂的内容，而将深度强化学习算法应用于自动驾驶系统则不是这种情况。例如，在某些 Atari 游戏中只有四个动作，例如 SpaceInvaders 和 Enduro。对于游戏 Go，即使接续空间是高维的，板的规则和状态也很容易从视觉上理解。在这种情况下，视觉问题非常容易解决，因此 agent 仅需专注于在行 动空间有限的情况下优化策略。但是对于自动驾驶，状态空间和来自环境的输入图像包含高度复杂的背景和诸如人类

之类的内部物体，它们可以动态变化并且行为无法预测。这些涉及许多困难的视觉任务，例如目标检测，场景理解，深度估计。更重要的是，在这种困难的情况下，我们的控制器必须正确，快速地采取行动，以避免撞到物体并保持安全。

实现自动驾驶的一种直接方法是通过使用精确，可靠的硬件和传感器（例如激光雷达和惯性测量单元（IMU））来捕获环境信息。这些硬件系统可以精确地重建 3D 信息，然后使用强化学习帮助车辆

实现智能导航而不会发生碰撞。但是，那些硬件非常昂贵且部署繁重。更重要的是，他们只告诉我们世界的 3D 物理表面，而不是了解环境，而这并不是真正的智能。硬件系统的这两个原因限制了自动驾驶技术的普及。

一种替代解决方案是将视觉和强化学习算法结合起来，然后共同解决感知和导航问题。但是，由于我们的世界极其复杂且不可预测，因此很难解决感知问题。换句话说，世界上存在着巨大的差异，例如颜色，物体的形状，物体的类型，背景和视点。甚至静止的环境也很难理解，更不用说环境随着自动驾驶汽车的运行而改变了。同时，由于动作空间是连续的并且可以同时执行不同的动作，因此控制问题在现实世界中也具有挑战性。例如，为了使转向更平稳，我们可以同时转向和制动，并在转向时调整转向度。更重要的是，安全的自动驾驶汽车必须确保功能安全并能够应对紧急事件。例如，车辆必须非常小心十字路口和看不见的弯道，以便在有小孩突然横穿马路时立即行动或制动。

为了实现自动驾驶，人们试图利用来自传感器和视觉算法的信息。许多合成驾驶模拟器都是为了学习导航策略而设计的，用来避免物理损坏。同时，人们正在开发更强大，更有效的强化学习算法[18、11、20、19、2]，以便成功处理现实世界中的复杂情况。在这个项目中，我们正在尝试探索和分析在合成模拟器中实现自动驾驶的可能性。

尤其是，我们采用深度确定性策略梯度（DDPG）算法[9]，该算法结合了 DPG，actor-critic 算法和深度 Q 学习的思想。我们选择开放式赛车模拟器（TORCS）作为我们训练 agent 的环境。为了学习 TORCS 中的策略，我们首先选择一组合适的传感器信息作为 TORCS 的输入。根据这些输入，我们在 TORCS 内设计自己的奖励器，以鼓励我们的 agent 快速驾驶而不会撞到其他汽车，并且保持自己处于道路的中心。同时，为了适应 TORCS 环境，我们为 DDPG 算法中的 actor

和 critic 设计了自己的网络体系结构。为了证明我们方法的有效性，我们在 TORCS 中以不同的方式评估 agent，其中包含不同的视觉信息。

2 相关工作

自主驾驶。解决自动驾驶的尝试可以追溯到深度学习时代之前的传统控制技术。在这里，我们仅讨论通过使用强化学习或深度学习技术在自动驾驶方面的最新进展。Karavolos [7]将 vanilla Q-Learning 算法应用于仿真器 TORCS，并在探索过程中评估了启发式方法的有效性。Huval2015 等人[5]提出了一种基于 CNN 的方法，将自动驾驶问题分解为汽车检测，车道检测任务，并在现实世界的高速公路数据集中评估其方法。另一方面，Bojarski 等人[3]通过提出端到端模型架构并在模拟器和真实环境中进行测试来实现自动驾驶。Sharifzadeh2016 等。[17]通过使用逆向强化学习方法来实现无碰撞运动和类似人的车道改变行为。与以前的工作不同，Shalev-shwartz 等人[16]将自动驾驶建模为多主体控制问题，并在非常简单的交通模拟器上演示了深度策略梯度方法的有效性。Seff 和 Xiao [15]提议利用 Google Map 和 Google Street View 图像中的匹配信息，以实现对先前工作的理解。最近的工作[22、4、6]主要关注于深度强化学习范例，以实现自动驾驶。为了在野外实现自动驾驶，You 等人[23]提出实现虚拟到真实图像的转换，然后学习对真实图像的控制策略。

强化学习。现有的强化学习算法主要由基于值和基于策略的方法组成。Vanilla Q 学习是在[21]中首次提出的，后来成为流行的基于值的方法之一。最近，由于深度神经网络的兴起，许多 Q 学习算法的变体，例如 DQN [13]，Double DQN [19]和 Dueling DQN [20]，已成功应用于各种游戏，并且表现优于人类。通过利用 actor-critic 方法的优势功能和思想[8]，A3C [11]进一步提高了基于值的强化学习方法的性能。与基于值的方法不同，基于策略的方法直接学习策略。换句话说，基于策略的方法输出给定当前状态的操作。Silver 等人[18]提出了一种确定性策略梯度算法，可以有效地处理连续的动作空间而又不会失去足够的探索性。通过结合 DQN 和 actor-critic 的想法，Lillicrap 等人[9]然后提出了一种深度确定性策略梯度方法，并实现了端到端策略学习。最近，提出了 PGQL [14]，通过将非策略性性 Q 学习与策略梯度相结合甚至可以胜过 A3C。更重要的是，就自动驾驶

而言，动作空间是连续的，需要精细控制。所有这些策略梯度方法都可以自然地处理连续的操作空间。但是，通过离散化连续动作空间使基于值的方法（例如DQN）适应连续域可能会导致维数诅咒，无法满足精细控制的要求。

3 方法

在自动驾驶中，动作空间是连续的。例如，转向的范围从-90°到 90°，加速度的范围从 0 到 300km。这种持续的行动空间将导致基于值的方法的性能下降。因此，我们在此项目中使用基于策略的方法。同时，自动驾驶中的随机探索可能会导致意外的性能和糟糕的序列。因此，我们决定使用深度确定性策略梯度（DDPG）算法，该算法使用确定性而非随机动作函数。特别是，DDPG 结合了确定性策略梯度算法，actor-critic 和深度 Q 网络的优点。

在本节中，我们描述确定性策略梯度算法，然后说明 DDPG 如何将 actor-critic 和 DQN 的思想结合在一起。最后，我们解释了如何在 TORCS 中拟合模型并设计奖励信号以在 TORCS 中实现自动驾驶。

3.1 确定性策略梯度（DPG）

随机策略可以定义为：

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

那么相应的梯度是：

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

这表明梯度是对可能的状态和动作的期望。因此，原则上为了获得梯度的近似估计，我们需要从动作空间和状态空间中获取大量样本。幸运的是，映射是在确定性策略梯度中从状态空间固定到动作空间的，因此我们不需要在整个动作空间上进行集成。因此，确定性策略梯度算法只需很少的数据样本即可在随机策略梯度上收敛。确定性策略梯度是作用值函数的预期梯度，因此可以比随机模型更有效地进行估计。

为了探索环境，DPG 算法通过借鉴 actor-critic 算法的思想来实现非策略学习。图 1 显示了 actor-critic 算法的总体工作流程。特别是，DPG 由 actor（要学习的策略）和 critic（用来估计 Q 值函数）组成。本质上，actor 在给定当前环境 s 的

情况下产生动作，而 critic 则产生评判 actor 的行为的信号。然后，通过 TD 学习来更新 critic，并通过策略梯度来更新 actor。假设 critic 的函数参数为 w 且 Actor 的函数参数为 θ，确定性策略的梯度为

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

对于勘探随机策略 β 和偏离确定性策略 μθ（s），我们可以得出偏离策略的梯度：

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

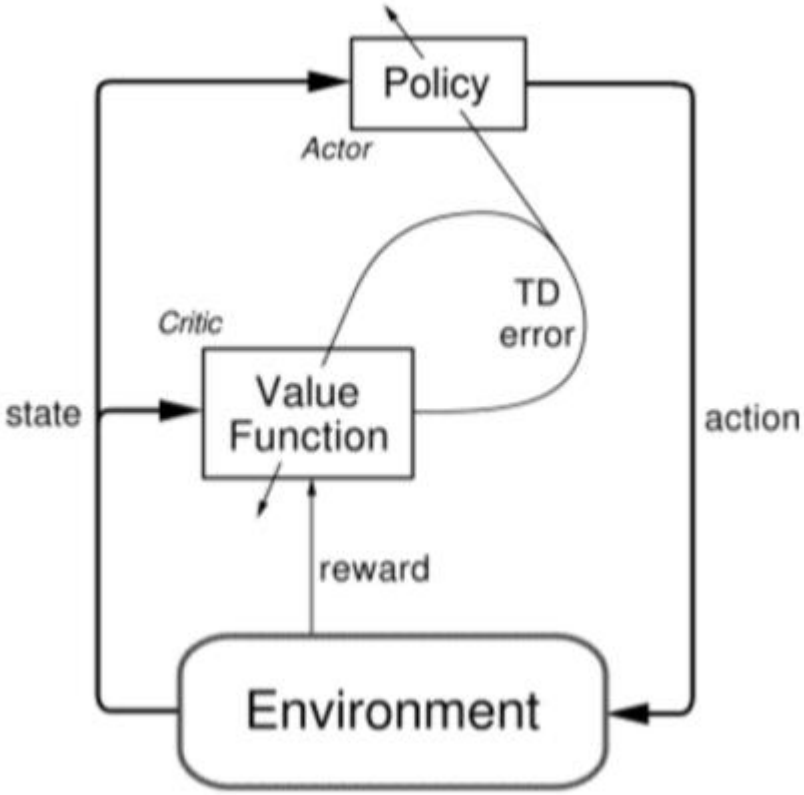


图 1：actor-critic 范式的整体工作流

请注意，该公式没有重要性采样因子。因为重要性采样是用一个简单的概率近似一个复杂的概率分布。但是这里的策略输出是一个值而不是一个分布。公式中的 Q 值对应于 critic 们，并通过 TD（0）学习进行更新。给定策略梯度方向，我们可以得出 Actor-Critic 非策略 DPG 的更新过程：

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

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

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

3.2 深度确定性策略梯度（DDPG）

DDPG 算法主要遵循 DPG 算法，但 actor 和 critic 的功能逼近均由深度神经网络表示。除了使用原始图像作为输入之外，Torcs 还支持各种类型的传感器输入，而不是将图像用作观察。在这里，我们选择采用表 1 中列出的所有传感器输入，使它成为一个 29 位向量。模型的动作是一个 3 维向量，分别为加速（其中 0 表示无油门，1 表示全油），制动（其中 0 表示无制动，1 表示全制动）和转向（其中-1 表示最大右转和+1 表示最大左转）。

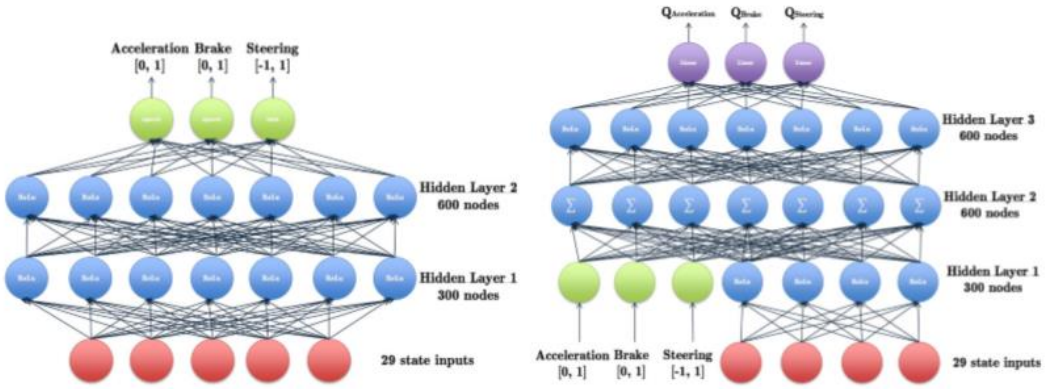
整个模型由一个 actor 网络和一个 critic 网络组成，如图 2 所示。actor 网络充当策略，并将输出动作。两个隐藏层都包含 ReLU 激活功能。评论器模型用作 Q 函数，因此将采取行动和观察作为输入，并输出每个行动的估计奖励。在网络中，之前的两个动作直到第二个隐藏层都不可见。第一和第三隐藏层被 ReLU 激活，而第二合并层计算在第一隐藏层上计算的线性激活与在动作输入上计算的线性激活的逐点求和。

整个模型由一个 actor 网络和一个 critic 网络组成，如图 2 所示。actor 网络充当策略，并输出动作。两个隐藏层都包含 ReLU 激活功能。评论器模型用作 Q 函数，因此将采取行动和观察作为输入，并输出每个行动的估计奖励。在网络中，之前的两个动作直到第二个隐藏层都不可见。第一和第三隐藏层被 ReLU 激活，而第二合并层计算在第一隐藏层上计算的线性激活与在动作输入上计算的线性激活的逐点求和。

同时，为了提高 agent 的稳定性，我们采用经验重播来打破数据样本之间的依赖性。DDPG 算法中使用了目标网络，这意味着我们同时为 actor 和 critic 网络创建了副本。然后，将这些目标网络用于提供目标值。然后以固定的频率更新这些目标网络的权重。对于 actor 和 critic 网络，参数 w 和 θ 分别通过以下方式更新：

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

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



(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 开放式赛车模拟器（TORCS）

TORCH 提供 18 种不同类型的传感器输入。经过实验后，我们仔细选择了一部分输入，如表 1 所示。

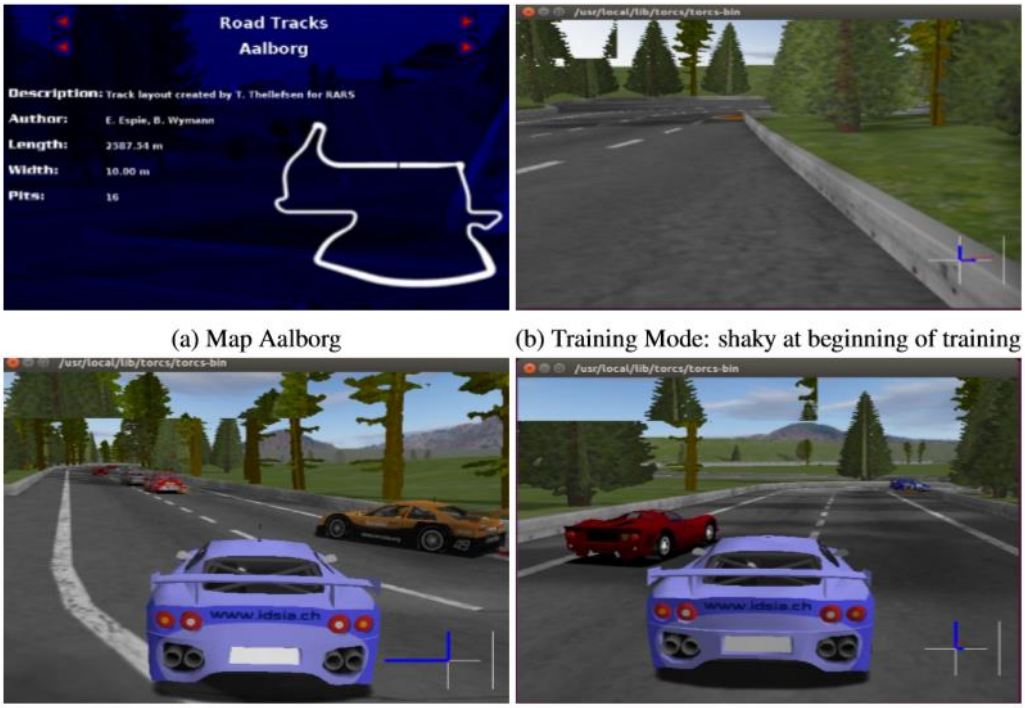
- ob.angle 是轿厢方向与轨道轴线方向之间的角度。它揭示了汽车向轨道线的方向。
- ob.track 是 19 个测距传感器的矢量：每个传感器返回 200 米范围内的轨道边缘与汽车之间的距离。它让我们知道汽车是否有撞上障碍物的危险。
- ob.trackPos 是汽车与轨道轴线之间的距离。该值归一化为 w.r.t. 到轨道宽度：当汽车在轴上时为 0，值大于 1 或-1 表示汽车在轨道外。我们希望到轨道轴的距离为 0。
- ob.speedX，ob.speedY，ob.speedZ 是轿厢沿轿厢纵轴（良好速度），轿厢横轴和 Z 轴的速度。我们希望汽车沿轴的速度较高，而垂直于该轴的速

度较低。

奖励设计 TORCS 没有内部奖励器，因此我们需要设计自己的奖励功能。奖励不仅应鼓励沿着轨道轴线的高速行驶，而且应惩罚垂直于轨道轴线的速度以及偏离轨道的速度。我们将奖励函数表述为：

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

Vxcos（θ）表示沿着轨道的速度，应鼓励这样做。Vxsin（θ）表示垂直于轨道的速度。| trackPos | 测量汽车与轨道线之间的距离。| trackPos | 和 Vx | trackPos | 当 agent 偏离道路中心时，对 agent 进行惩罚。α，β，γ 分别表示每个奖励项的权重。



(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 实验

4.1 实验设置

我们在 TORCS 引擎上实验了简单的 actor-critic 算法，并在 OpenAI Universe 上实验了异步 actor-critic 算法。我们选择 TORCS 作为 TORCS 的环境，因为它是与 OpenAI 兼容的接口包装的。其他软件包括 Anaconda2.7，Keras 和 Tensorflow

v0.12。我们所有的实验都是在 Ubuntu 16.04 计算机上进行的，该计算机具有 12 核 CPU，64GB 内存和 4 个 GTX-780 GPU（总共 12GB 图形内存）。我们从[1]中调整实现和超参数选择。具体来说，重播缓冲区的大小为 100000 个状态操作对，折扣系数为 $\gamma = 0.99$。优化器是 Adam，actor 和 critic 的学习率分别为 0.0001 和 0.001，批处理大小为 32。目标网络逐渐更新， $\tau = 0.001$。

4.2 实验分析

TORCS 引擎包含许多不同的模式。我们通常可以将它们分为两种类型：训练模式和比赛模式。在训练模式下，视图中没有其他竞争者，并且如图 3b 所示，视角是第一人称视角。在竞赛模式下，我们可以将其他计算机控制的 AI 添加到游戏中并与之竞赛，如图 3c 所示。值得注意的是，其他竞争对手的存在将影响我们汽车的传感器输入。

我们以训练模式在奥尔堡地图上训练了约 200 次的游戏，并与其他 9 个竞争对手在比赛模式下评估了游戏。当汽车驶出赛道或方向相反时，每个事件都会终止。因此，每个 episode 的长度差异很大，因此好的模型可以无限期运行一个 episode。因此，我们还将每次的最大长度设置为 60000 次迭代。该图如图 3a 所示。在训练模式下，模型开始时会晃动，并且经常撞到墙壁上（图 3b），并且随着训练的进行逐渐稳定。在评估（竞争模式）中，我们将所有竞争者的汽车排名从开始时的第 5 位开始。因此，我们的汽车在开始时就落后于其他 4 辆汽车（图 3c）。但是，随着比赛的继续，我们的赛车很容易依次超越其他竞争对手，如图 3d 所示。

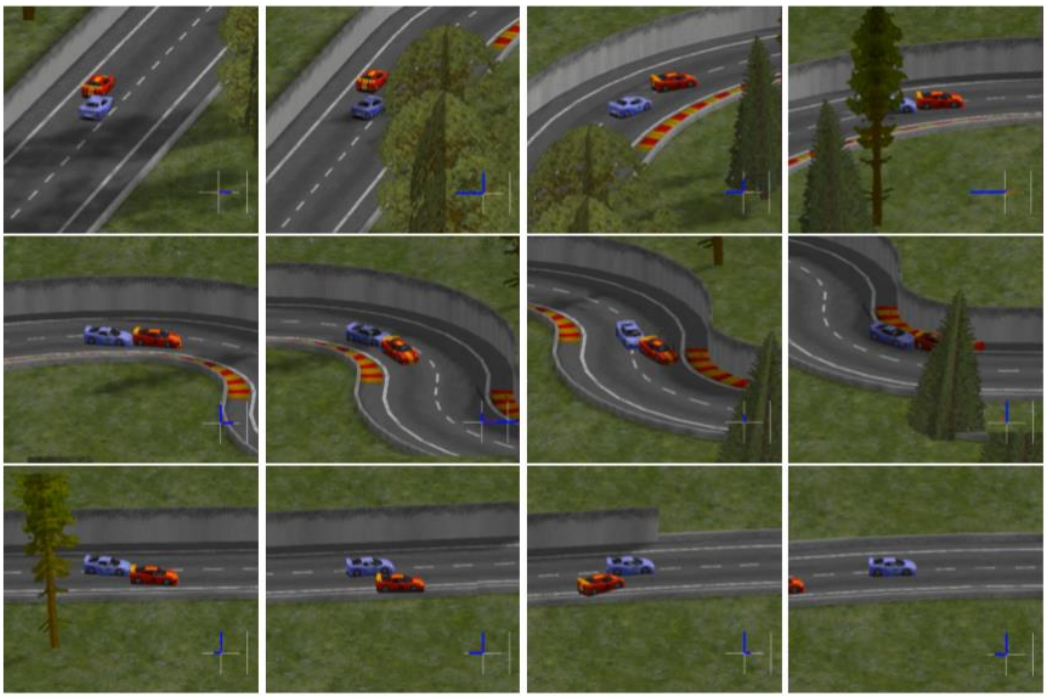


Figure 4: Compete Mode: our car (blue) over take competitor (orange) after a S-curve. For a complete video, please visit `https://www.dropbox.com/s/balm1vlajjf50p6/drive4.mov?dl=0`.

我们在图 4 中说明了 12 个连续的图像，以显示我们的 agent（蓝色）如何绕 S 形曲线以及如何超车。值得注意的是，当速度快时，TORCS 嵌入了良好的物理引擎和模型来对车辆进行漂移。我们发现漂移是弯道过弯后朝错误方向行驶的主要原因，并导致提前终止 episode。为了解决这个问题，我们的 agent 必须在转弯之前降低速度，方法是踩刹车或松开加速器，这也是人们在现实生活中的驾驶方式。经过训练后，我们发现我们的模型确实学会了在弯道前释放加速器以减慢速度以避免漂移。同样，从图 4 中我们可以发现我们的模型没有学习如何避免与竞争对手发生冲突。这是因为在培训模式下，没有竞争对手引入环境。因此，即使我们的汽车（蓝色）也可以比竞争对手（橙色）更快地通过 s 曲线，而无需主动进行侧移，在 s 曲线期间，我们的汽车被橙色竞争对手挡住了，并完成了 S 曲线后超车。我们在转折点附近或之后目睹了许多超车，这表明我们的模型在处理曲线时效果更好。通常，在经过一到两个圆圈之后，我们的汽车在所有竞争对手中排名第一。我们将完整的视频上传到 Dropbox。

我们在图 5 的训练过程中绘制了模型的性能图，其中包含 3 个子图，并将它们从上到下分别称为（顶部），（中间），（底部）。所有 3 个子图的 x 轴是对齐的

训练集。

在图 5（顶部）中，绘制了汽车平均速度（km / h）和每个情节每一步的平均增益。具体而言，仅通过计算沿汽车正面方向的速度分量来计算汽车的速度。换句话说，不计算漂移速度。每步的增益用公式（10）计算。从图开始，随着训练的进行，平均速度和步阶增益缓慢增加，并在大约 100 次发作后稳定下来。这表明在经过约 100 次训练后，训练实际上变得稳定了。除此之外，我们还目睹了平均速度和步进增益的同时下降。这是因为即使培训过时，汽车有时也可能冲出赛道并卡住。通常，当汽车冲出赛道时，该情节应该终止。但是，它不能保证每次都能成功终止,这可能是由于在 TORCS 中无法准确检测到这种偏离轨道的缘故。当发生卡死时，汽车将一直保持 0 速行驶并卡住多达 60000 次迭代，并严重降低了此 episode 的平均速度和逐步增益。此外，此 episode 中的许多垃圾历史记录都冲破了重播缓冲区并使训练变得不稳定。由于此问题起源于环境而不是学习算法，因此我们没有花费太多时间来解决它，而是终止了该 episode 并在看到发生的情况下手动继续下一个 episode。

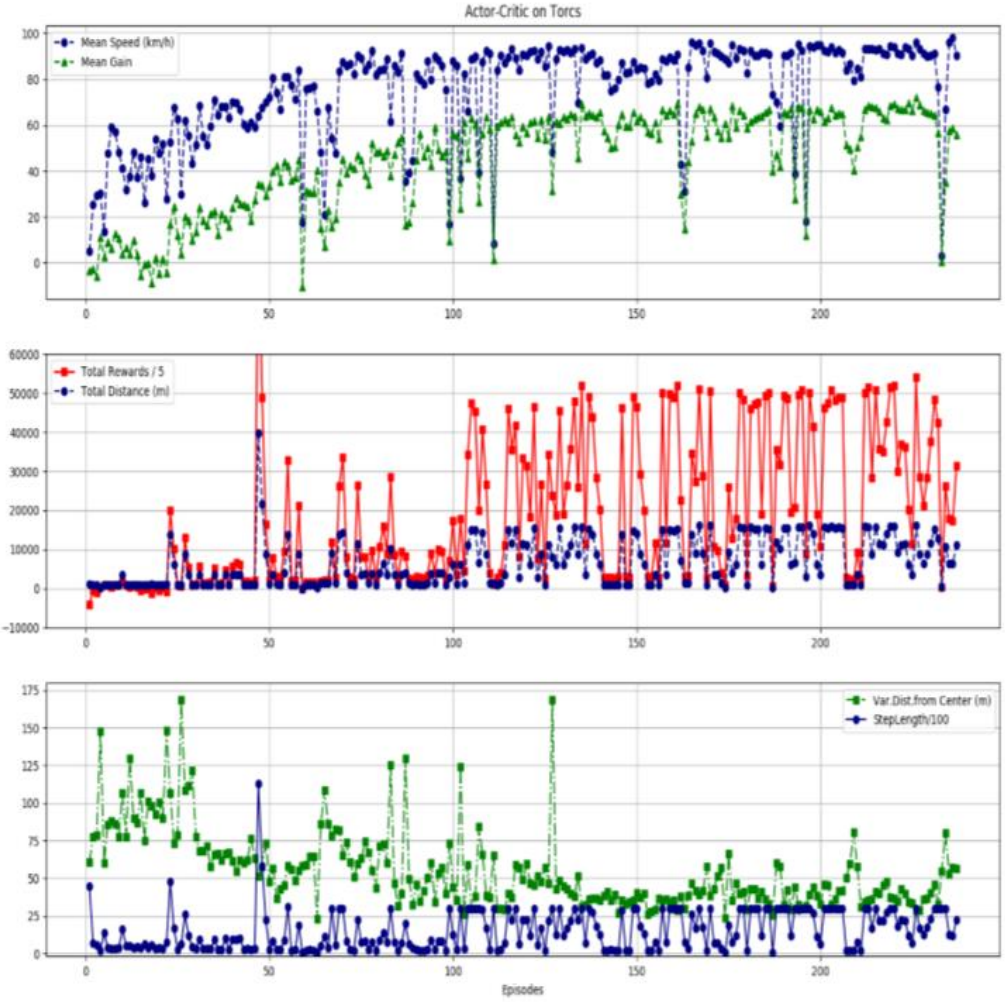


Figure 5: Model performance in episodes

在图 5（中）中，我们将汽车的总行驶距离和当前 episode 的总奖励与 episode 的指数作图。我们可以直观地看到，随着训练的继续，每次的总奖励和总距离在增加。这是因为模型变得越来越好，并且不太可能发生崩溃或跟踪失败。理想情况下，如果模型是最佳的，则汽车应无限行驶，并且总距离和总报酬将保持稳定。但是，由于与上述相同的原因，我们不断看到突然的下降。值得注意的是，“总距离”中的大多数“下降”都具有相同的值，这在许多情况下证明了“卡住”发生在地图上的相同位置。

在图 5（底部）中，我们绘制了距轨迹中心的距离（Var.Dist.from.Center（m））的变化和一集的步长。到轨道中心的距离的变化量衡量了行驶的稳定性。我们显示出，我们训练有素的 agent 通常一开始就像 8 字型的“醉汉”车手那样驾驶，而在以后的阶段中逐渐变得更好。中心点附近的距离变化曲线在约 150 次发作后

减小并稳定下来。这表明当速度和 episode 奖励已经稳定时，我们的模型在 100 个 episode 后仍然不能稳定行驶。因此，额外的 50 次训练能够使训练结果趋于稳定。

5 结论

为了弥合自动驾驶和强化学习之间的差距，我们采用深度确定性策略梯度（DDPG）算法在开放赛车模拟器（TORCS）中训练我们的 agent。特别是当我们从 TORCS 中选择适当的传感器信息作为我们的输入，并在连续域中定义我们的动作空间的时候。然后，我们在 DDPG 范式中为 actor 和 critic 设计奖励器和网络体系结构。我们证明了我们的 agent 能够在模拟器中快速行驶，并同时确保功能安全。