A SIM2REAL METHOD BASED ON DDQN FOR TRAINING A SELF-DRIVING SCALE CAR

Qi Zhang
School of Information Science and Technology
North China University of Technology, Beijing 100144, China

Tao Du*
School of Information Science and Technology
North China University of Technology, Beijing 100144, China
State Key Laboratory of Turbulence and Complex Systems
College of Engineering, Peking University, Beijing 100871, China

Changzheng Tian
School of Information Science and Technology
North China University of Technology, Beijing 100144, China

(Communicated by Guoshan Zhang)

Abstract. The self-driving based on deep reinforcement learning, as the most important application of artificial intelligence, has become a popular topic. Most of the current self-driving methods focus on how to directly learn end-to-end self-driving control strategy from the raw sensory data. Essentially, this control strategy can be considered as a mapping between images and driving behavior, which usually faces a problem of low generalization ability. To improve the generalization ability for the driving behavior, the reinforcement learning method requires extrinsic reward from the real environment, which may damage the car. In order to obtain a good generalization ability in safety, a virtual simulation environment that can be constructed different driving scene is designed by Unity. A theoretical model is established and analyzed in the virtual simulation environment, and it is trained by double Deep Q-network. Then, the trained model is migrated to a scale car in real world. This process is also called a sim2real method. The sim2real training method efficiently handles these two problems. The simulations and experiments are carried out to evaluate the performance and effectiveness of the proposed algorithm. Finally, it is demonstrated that the scale car in real world obtains the capability for autonomous driving.

1. Introduction. It is well known that any accident is unacceptable for passengers in driving. The security and reliability must be satisfied under the stringent standard. In order to satisfy safety standards, it seems to be reasonable that the self-driving car is equipped with many kinds of high-precision sensors. The high-precision sensors can improve the accuracy of the algorithms, but they are very expensive for the consumer [7]. Thus, it is necessary to balance accuracy and cost.

2010 Mathematics Subject Classification. Primary: 58F15, 58F17; Secondary: 53C35.
Key words and phrases. Deep reinforcement learning, self-driving, double Q-learning, unity.
* Corresponding author: Tao Du.
Recently, the artificial intelligence technology has got rapid development, especially in deep learning (DL), which has made a breakthrough in object recognition and intelligent control. Deep learning, typically such as convolutional neural networks (CNN), is widely used in image processing. It has been proved that DL is suitable for self-driving applications. At first, DL is used to learn end-to-end self-driving strategy through convolutional neural networks (CNN) under supervised learning, then the mapping relationship is obtained. Finally, the pattern-replicating driving skills are gained [5]. Although end-to-end driving is easy to scale and adaptable, it has limited ability to handle long-term planning which involves the nature of imitation learning [6, 21]. Since there are many problems with the replication pattern, especially on the sensor. We expect the scale cars learn how to drive on their own than under human supervision. The traffic accidents of Tesla are caused by the failure of the perceived module in a bright light environment. Deep reinforcement learning can make appropriate decisions even some modules fail in working [8].

This paper focuses on the issue of self-driving based on the deep reinforcement learning, we modify a 1:16 radio-controlled car and train it by double deep Q network (DDQN), shown as Figure 1. The virtual-to-reality process is used to train under different environments, in other words, the car is trained in the virtual environment and test in reality. To get a reliable simulation environment, we create a Unity simulation training environment based on OpenAI gym. A reasonable reward mechanism is set up, and DDQN is modified, which makes the algorithm suitable for training a self-driving car. The car is trained in the Unity simulation environment for many episodes. At last, the scale car can learn a pretty good policy to drive itself, and the learned policy is successfully transferred to the real world.

![Figure 1. The reinforcement learning scale car based on DDQN.](image)

2. Related works. Our aim is to make a self-driving car trained by deep reinforcement learning. Right now, the most common methods to train the car to perform self-driving are behavioral cloning and line following. On a high level, behavioral cloning works by using CNN to learn a mapping between car images (taken by the front camera) and driving behavior through supervised learning [5]. Indeed, the behavioral cloning methods based on end-to-end deep learning can efficiently achieve the self-driving task. However, each part of the network is used for feature extractor and controller (For example, the full connect layer trains the features extracted by
convolutional layer and then outputs the control signals of the turn), the boundary between the feature extractor layer and controller layer is vague. Therefore, if we want to improve the adaptability of the model, the data should be increased continually to traverse all the possible driving scenes of driving. Furthermore, the distributions of the driving data under different driving scenes are relatively different, because the environment is dynamic, the training data and test data are independently identically distribution. Then the self-driving behavior trained by these data may result in a terrible accident.

The other method, the line-following, works by using computer vision techniques to track the middle line and utilizes a PID controller to make the car follow the line. Aditya Kumar Jain used CNN technology to complete the self-driving car with a camera [2]. Kaspar Sakmanni proposed a typical supervised behavioral learning method by CNN [11], where the human driving data were collected through a camera for learning. However, these are the capabilities that under manual intervention. An intelligent way that a car quickly learns how to safely drive by itself need to be researched.

In 1989, Watkins et al proposed the noted Q-learning algorithm, which is mainly based on the Q-table to record the state value of the action pair, and each episode updates the state value [22]. In 2013, Mnih et al pioneered the use of deep reinforcement learning [12], they proposed the deep Q-network and successfully applied in Atari games. Then they ameliorated the deep Q-network in 2015 [13]. Two identically structured networks are used in DQN: behaviour network and target network. Although this method improves the stability of the model, the Q-learning’s problem overestimating the value is hard to solve. To fix this problem, Hasselt et al. proposed a DDQN method that the double Q learning method is applied to DQN [20]. The proposed DDQN method is to implement the selection of actions and the evaluation of actions with different value functions.

Recently, the virtual simulation techniques are introduced to train an intensive learning model, and the trained model can be migrated to reality, which has been verified. A robotic arm, called Dactyl, developed by OpenAI, was trained in a virtual environment and applied to the reality successfully [23]. The performance of Dactyl has been verified by picking up and placing objects [3], visual servo [16], and flexible movement [14], etc., all indicating the feasibility of the migrated learning method. In 2019, Luo et al. proposed an end-to-end active target tracking method based on reinforcement learning, which trained a robust active tracker in a virtual environment through a custom reward function and environment enhancement technology.

From the above works, visual autopilot algorithms can learn self-driving to control the car by neural network under the condition of supervised learning. However, this supervised method contains many flaws. For example, Tesla’s driverless accident is caused by perceived module failure in a bright light environment [15]. To solve this problem, the reinforcement learning is a better learning method even some modules are invalid. Reinforcement learning makes it easier to learn a range of behaviors. Besides, autopilot requires a series of corrective actions to drive successfully which can be satisfied by reinforcement learning method because it can learn to automatically correct the offset generated by the labeled data-set. Self-learning requires better coordination rather than more sensors used [8]. Hence, we select the deep reinforcement learning method to guarantee the safety of self-driving car.
3. Proposed method.

3.1. Self-driving scale car. Autonomous vehicles are usually composed of traditional sensing systems, computer decision systems and driving control systems [10]. The function of the sensing system is to capture surrounding environmental information vehicle driving state and provide information support for decision controller. According to the scope of perception, it can be divided into environmental information perception and vehicle state perception. The environmental information includes roads, pedestrians, obstacles, traffic control signals, and vehicle geographic location. The vehicle information includes driving speed, gear position, engine speed, wheel speed, and the amount of oil, etc.. The sensing systems are commonly composed of ultrasonic radar, video acquisition sensor and positioning device [17].

In our desired experiments, only the visual data is used as a sensing device. The radio-controlled car, shown as Figure 2, is utilized as a benchmark for retrofitting. And the hardware includes:

- Raspberry Pi 3: It is a low-cost computer with 1.2 GHz processing speed and 1 GB memory. It is equipped with a customized version of the Linux system, supports bluetooth, wifi communication, and offer adequate support for i2c, etc.. The agreement amount is GPIO port, which is the calculation brain for the auto-driving car.
- Servo Driver (PCA 9685): It includes an i2c-controlled PWM driver with a built-in clock to drive the modified servo system.
- Wide Angle Raspberry Pi Camera: Its resolution is $2592 \times 1944$ and the viewing angle is 160 degrees. It is an environmental sensing device.
- Other: According to the design provided by the Donkey Car community, a 3D-printed car bracket is used for carrying various hardware devices.

![Figure 2](image.png)

**Figure 2.** One 1:16 scale car. There is an opensource DIY self-driving platform for small scale cars called donkeycar [18].

3.2. Setting environment.

3.2.1. Scale car simulator. The first step is to create a high friendly simulator for scale car. Currently, the researcher from the scale car community has generously created a scale car simulator in Unity. However, it is specifically designed to perform
behavioral learning (i.e. save the camera images with the corresponding steering angles and throttle values in a file for supervised learning), but it fails to cater to reinforcement learning. What we expected is an OpenAI gym like interface where the simulated environment can be manipulated through calling some functions to reset the environment and to step through the environment. We made some modifications to make it compatible with reinforcement learning. Since the reinforcement learning algorithms are realized by Python, we have to first figure out a way to make Python communicate with Unity. It turns out that the Unity simulator created by Tawn Kramer also comes with Python code for communicating with Unity. The communication is done through the Websocket protocol which, unlike HTTP, allows two-way bidirectional communication between server and client. In this case, the Python “server” can push messages directly to Unity (e.g. steering and throttle actions), and the Unity “client” can also push information (e.g. states and rewards) back to the Python server.

3.2.2. OpenAI gym environment for scale car. The next step is to create an OpenAI gym like interface for training reinforcement learning algorithms. The gym environment is customized by extending the OpenAI gym class. The resulting environment is compatible with the OpenAI gym. Besides, we can interact with the scale environment using the familiar gym like interface. The environment also allows to set frame skipping and train the reinforcement learning agent in headless mode (i.e. without Unity GUI). Therefore, a virtual environment is established. The pixel images are taken by the front camera of the scale car, and the transformations are performed as follow

- Resize image from $120 \times 160$ to $80 \times 80$;
- Turn images into grayscale;
- Frame stacking: stack 4 frames from previous time steps together;
- The final state is of dimension $(1, 80, 80, 4)$.

3.3. Algorithm.

3.3.1. Reinforcement learning. Figure 3 shows the elements and processes of reinforcement learning. The agent takes action and interacts with the environment, then the environment returns rewards and moves to the next state. Through multiple interactions, the agent gains experience and seeks the optimal strategy in the experience. This interactive learning process is similar to the human learning style whose main characteristics are trial, error and delayed return. The learning process can be described as the Markov decision process (MDP). MDP consists of triples “$S, A, P, r$”, where

$$S = \{s_1, s_2, s_3 \ldots\}$$

$$A = \{a_1, a_2, a_3 \ldots\}$$

$$P_{ss'} = P[S_{t+1} = s' | S_t = s_t, A_t = a_t]$$

$$r_t = r(s_t, a_t)$$

$S, A, P, r$ are respectively defined as

- $S$ delegates the set of all states, whose dimension is $D_S = 1 \times |S|$.
- $A$ delegates the set of all actions, whose dimension is $D_A = 1 \times |A|$.
- $P$ is the matrix of state transition probability, which means the transition probability when the agent takes action “a” and changes state “s” to $s'$ and always indicated as $P_{ss'}$, whose dimension is $D_P = |S| \times |A| \times |S|$.
320 QI ZHANG, TAO DU AND CHANGZHENG TIAN

- $r$ delegates the reward function, which means the reward of taking action “$a$” under state “$s$”.

**Figure 3.** The process of reinforcement learning.

The agent forms an interaction trajectory in each round of interaction with the environment $(s_1, a_1, r_1, s_2, a_2, r_2, ..., s_T, a_T, r_T)$, and the cumulative return under the state is formulated as (1)

$$R_t = \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$  \hspace{1cm} (1)

Where the $\gamma \in [0, 1]$. $\gamma$ means the discount coefficient of the return, is used to weigh the relationship between current returns and long-term returns. The higher value the more attention is paid to long-term returns, and vice versa.

The goal of reinforcement learning is to learn a policy while maximize the expectations of cumulative returns, which indicates as equation (2). The dimension of the policy is $|S| \times |A|$.

$$\pi(a|s) = \arg \max_a E[R_t]$$  \hspace{1cm} (2)

To obtain the optimal policy, the value function and the action state value function are introduced to evaluate the advantages and disadvantages of a certain state and action. The value function $V_\pi(s)$ is defined as the equation (3):

$$V_\pi(s) = E_\pi[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s]$$  \hspace{1cm} (3)

Where the value function $V_\pi(s)$ whose dimension is $1 \times |S|$ means when the agent takes policy $\pi$, the mathematical expectation of reward accumulation under state “$s$”.

Define the state-action value function as the equation (4):

$$Q_\pi(s,a) = E_\pi[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a]$$  \hspace{1cm} (4)

Where the dimension of $Q_\pi(s,a)$ is $|S| \times |A|$, it means when agent takes policy $\pi$ the mathematical expectation of reward accumulation under state “$s$” and taking action “$a$”.
3.3.2. MDP formulation. The actions of a car (agent in MDP) can be taken as forwarding (a=1) or being stationary (a=0). The state is represented by a sequence of frames from the camera. Specifically, the state is the sequence shown in the equation (5)

$$s_t = (x_{t-Len+1}, a_{t-Len+1}, ..., x_{t-1}, a_{t-1}, x_t)$$ (5)

Where $s_t$ is the state at time $t$, $x_t$ is the pixel input through the camera at time $t$, and $a_t$ is the action taken at time $t$. $Len$ is a hyperparameter that specifies how many of the most recent frames to keep track of, which is to reduce the storage and state space compared to saving all frames and actions starting from $t = 1$. The reason why storing multiple frames rather than storing a single frame is that the agent needs temporal information to play. The discount factor is setting as $\gamma = 0.95$. The transition probabilities and the rewards are unknown for the agent. Since Q-learning is model-free, the transition probabilities and rewards are not explicitly estimated, but the optimal Q-function needs to be calculated directly. This is described further in the following section.

Both the car in the real world and the simulator take continuous steering and throttle values as input. For the sake of simplifying, we set throttle value as constant (i.e. 0.7) to control the steering only. The steering value ranges from $-1$ to 1. However, DQN can only handle discrete actions, therefore, we discretize the steering value into 15 categorical bins.

The reward is a function of cross track error (cte) which is provided by the Unity environment. The indicator cte measures the distance between the center of the track and car. The revised reward is given by the following formulation (6):

$$r = 1 - \frac{|cte|}{cte_{max}}$$ (6)

Where $cte_{max}$ is a normalizing constant so that the reward is within the range of 0 and 1. We terminate the episode once the absolute value of cte is larger than $cte_{max}$.

3.3.3. Q-learning. The goal in reinforcement learning is to maximize the expected value of the total payoff (or expected return). In Q-learning, which is off-policy, we use the Bellman equation as an iterative update by

$$Q_{i+1}(s, a) = E_{s', \varepsilon} \left[ r + \gamma \max_{a'} Q_i(s', a') | s, a \right]$$ (7)

Where $s'$ is the next state, $r$ is the reward, $\varepsilon$ is the environment, and $Q_i(s, a)$ is the Q-function at the $i$-th iteration. From the equation (7), it can be shown that the iterative update converges to the optimal Q-function (the Q-function associated with the optimal policy). To prevent rote learning, the function approximations are necessary for the Q-function to allow generalization for unseen states. The deep Q-learning approach which a neural network is used to approximate the Q-function.

For a given experience $e = (s, a, r, s')$, a common loss used for training a Q-function approximator is given by

$$L_i(\theta_i) = E_{s, a, \varepsilon} \left[ \frac{1}{2} (y_i - Q_i(s, a; \theta_i))^2 \right]$$

Where $\theta_i$ is the parameters of the Q-network at the $i$-th iteration, and $y_i$ is the target at the $i$-th iteration. The target $y_i$ is defined as:

$$y_i = E_{s', \varepsilon} \left[ r + \gamma \max_{a'} Q_i(s', a'; \theta_{i-1}) | s, a \right]$$
An experience is analogous to a data point such as in linear regression and the replay memory, a list of experiences are analogous to a dataset such as in linear regression. The gradient of the loss function concerning the weights is given as the equation (8).

\[ \nabla \theta_i L_i(\theta_i) = E_{s,a \sim \rho}((r + \gamma \max_{a'} Q_i(s', a'; \theta_{i-1}) - Q(s, a; \theta_i)) \nabla \theta_i Q(s, a; \theta_i) \]  

(8)

Thus, we can simply use stochastic gradient descent and back-propagation on the equation (8) to update the weights of the network.

Additionally, we take a \( \varepsilon \)-greedy approach to handle the exploration-exploitation problem in Q-learning, where a random action with probability \( \varepsilon \) and the optimal action \( a_{\text{opt}} = \arg\max_a Q(s, a) \) are chosen. In the implementation, we linearly change the exploration probability \( \varepsilon \) from 1 to 0.1 as the agent trains. This is to encourage a lot of exploration at the beginning where the agent has no idea how to play the game and the state space is extremely large. It takes a large number of random actions and as it starts to figure out which actions are better in different situations, and exploits more and tries to narrow down what the optimal actions are.

3.3.4. Experience replay and stability. The problem of traditional Q-learning is that the experiences from consecutive frames of the same episode (a run from start to finish of a single game) are closely correlated. This hinders the training process and leads to inefficient training. Therefore, to de-correlate the experiences, the method of experience replay is used [9]. In the experience replay, we store an experience \((s, a, r, s')\) at every frame into the replay memory. The replay memory has a certain size and contains the most recent replay memory size experiences. It is constantly updated (like a queue) so that they are associated with the actions taken with the recent Q-functions. The batch used to update the DQN is composed of uniformly sampling experiences from the replay memory. Therefore, the experiences are no longer correlated.

Moreover, to encourage more stability in decreasing the loss function, a target network \( \hat{Q}(s, a) \) is employed. \( Q(s, a) \) is essentially the same as \( Q(s, a) \). The network has the same structure, but the parameters may be different. When parameters update to the DQN \( Q(s, a) \), \( \hat{Q}(s, a) \) is updated at the same time. Then \( \hat{Q}(s, a) \) is used for computing the target \( y_i \) according to:

\[ y_i = E_{s' \sim \varepsilon}[(r + \gamma \max_{a'} \hat{Q}_i(s', a'; \theta_{i-1})) | s, a] \]  

(9)

Equation (9) leads to better stability when updating the DQN.

3.3.5. Double deep Q-network. The DQN method has the problem of the overestimation of Q-values because the Q-learning is a typical time difference (TD) target problem [1], shown in the (10).

\[ Q(s, a) = r(s, a) + \gamma \max_a Q(s', a) \]  

(10)

According to the section Q-learning and reinforcement learning model, the \( Q(s, a) \) is the Q target, \( r(s, a) \) is the reward of taking action ‘a’ at state ‘s’, and \( \gamma \max_a Q(s', a) \) is the discounted max Q value among all possible actions from next state. It is known that the accuracy of Q-values depends on what action we tried and what neighboring state we explored. As a consequence, at the beginning of the training, we have few information about the best action to take. Therefore, taking the maximum Q value (which is noisy) as the best action to take can lead to false positives.
If non-optimal actions are regularly given a higher Q value than the optimal best action, the learning will be complicated. To solve this problem, the DDQN was introduced by Hado Van Hasselt [20]. The solution is that when computing the Q target, the author uses two networks to decouple the action selected from the target Q-value generation shown as:

- Use DQN network to select what is the best action to take for the next state (the action with the highest Q value).
- Use target network to calculate the target Q value of taking that action at the next state.

Then equation (10) is revised by

\[ Q(s, a) = r(s, a) + \gamma Q(s', \text{arg max}_a Q(s', a)) \]

Where \( \gamma Q(s', \text{arg max}_a Q(s', a)) \) is the target network that calculates the Q value of taking action ‘a’ at state ‘s’’. \( \text{arg max}_a Q(s', a) \) is the DQN network choosing action for next state. Therefore, DDQN is used to reduce the overestimation of Q values and, as a consequence, and faster and more stable learning is obtained.

In the section Environment require, the pre-processings of pixel images taken by the front camera are introduced. Hence, it is rescaled to 80 × 80 pixels and normalized from [0, 255] to [0, 1], which is called by feature extractor \( \phi(s) \).

Therefore, the Q-function is approximated by a convolutional neural network. The network takes as input an 80 × 80 × Len image and has a single output for every possible action. The architecture of the network is shown in figure 4. The first layer convolves the input image with an 8 × 8 × 4 × 32 kernel at a stride size of 4. The output is then put through a 2 × 2 max-pooling layer. The second layer convolves with a 4 × 4 × 32 × 64 kernel at a stride of 2. Then we max the pool again. The third layer convolves with a 3 × 3 × 64 × 64 kernel at a stride of 1. The pool is maxed one more time. The last hidden layer consists of 256 fully connected ReLU nodes. These are all about DDQN.

**Figure 4.** The architecture of the network
3.3.6. Pipeline. The pipeline for the entire DDQN training process is shown in Algorithm 1. It is as previously described earlier in this section. The Q-learning is applied with three experience replays used, which stores every experience in the replay memory at every frame. When we perform an update to the DDQN, a batch of experiences are obtained by sampling uniformly to update the DDQN. This is analogous to sampling batches from a dataset using SGD/mini-batch gradient descent in convolutional neural networks for image classification or deep learning in general. Then the exploration probability is updated as well as the target network $\hat{Q}(s,a)$ if necessary.

Algorithm 1 Double Deep Q-learning algorithm for self-driving scale car

initialize replay memory
initialize DDQN to random weights
repeat
  new episode
  initialize state $s_0, Q_A, Q_B,$
  repeat
    extract $x_t$ from raw pixel data update state $s_t$ with $x_t$
    add experience $e_t = (\phi(s_{t-1}), a_{t-1}, r_{t-1}, \phi(s_t))$ to replay memory
    choose $a$, based on $Q_A(s, \cdot)$ and $Q_B(s, \cdot)$, observe $r, s'$
    choose (e.g random) either UPDATE(A) or UPDATE(B)
    if UPDATE(A) then
      define $a^* = \arg \max_a Q_A(s', a)$
      $Q_A(s,a) \leftarrow Q_A(s,a) + \alpha(s,a)(r + \gamma Q_B(s', a^*) - Q_B(s,a))$
    else if UPDATE(B) then
      define $b^* = \arg \max_a Q_B(s', a)$
      $Q_B(s,a) \leftarrow Q_B(s,a) + \alpha(s,a)(r + \gamma Q_A(s', b^*) - Q_A(s,a))$
    end if
  uniformly sample a batch of experiences from the replay memory
  backpropagate and update DDQN with the minibatch
  update exploration probability $\epsilon$
  if $C_{update}$ to DDQN since last update to target network then
    update the target Q-network
    $\hat{Q}(s,a) \leftarrow Q(s,a)$
  end if
  update state $s_t$ with $a_t$
  update current reward $r_t$ and total reward $totalReward$
  update car’s parameters
  refresh screen
until Car crashes
restart a car
until convergence or number of iterations reached

The frame skipping is set to 2 for stably training. Memory replay buffer (i.e. storing $<state, action, reward, state_{next}>$ tuples) has a capacity of 10000. Target Q network is updated at the end of each episode. The batch size for training the CNN is 64. Epsilon-greedy is used for exploration, which is initially set to 1 and gradually annealed to a final value of 0.02 in 10,000-time steps.
4. Simulation tests and experiments.

4.1. Simulation tests. Essentially, reinforcement learning agent is expected to make its output decision (i.e. steering) only on the location and orientation of the lane lines and neglect everything else in the background. However, since the full pixel camera images are given as inputs, it might overfit to the background patterns instead of recognizing the lane lines. This is especially problematic in the real-world settings where there are undesirable objects lying next to the track (e.g. tables and chairs) and people walking around the track. The best way to transfer the learned policy from the simulation to the real world is neglecting the background noise and just focusing on the track lines for an agent.

To address this problem, a pre-processing pipeline is created to segment out the lane lines from the raw pixel images before feeding them into the CNN. The procedure is described as follows:

- Detect and extract all edges using Canny Edge Detector.
- Identify the straight lines through Hough Line Transform.
- Separate the straight lines into positive sloped and negative sloped (candidates for left and right lines of the track)
- Reject all the straight lines that do not belong to the track utilizing slope information.

The result transformed images consists of 0 to 2 straight lines representing the lane, illustrated as the figure 5.

![Fig. 5.](image.png)

**Figure 5.** The examples of raw images transfer to the segmented images.

We then took the segmented images, resize them to 80 × 80, stack 4 successive frames together and use it as the new input states. Then DDQN was trained again with the new states. The resulting reinforcement learning agent was again able to learn a good policy to drive the car. With the setup above, the simulation tests ran on a single CPU and a TITAN-X GPU to train DDQN for around 3300 episodes.
The learning curve (average rewards - train episodes curve) is shown as figure 6. The entire training took around 3 hours.

![Learning Curve](image)

**Figure 6.** The learning curve of “average rewards - train episodes”.

As we can see from figure 7, the car was able to learn a pretty good policy to drive itself. Notice that the car learned to drive and stayed at the center of the track most of the time.

![Scale Vehicle Car in Unity Simulation](image)

**Figure 7.** The scale vehicle car in the Unity Simulation.

4.2. Experiments. We have customized a $3.5m \times 4m$ simulation track, shown as Figure 8. The track and Unity environment has a high degree of reduction, which is similar to the real road (according to China’s right-hand drive standard).

We modified the program to change the trained model input from Unity’s output to the camera’s real-time input. The program is transferred to the Raspberry Pi, shown as Figure 9. After several experiments, the car successfully followed the rules and realized self-driving along the road. The first image shows that the car meets a sharp turn. In the second and third image, the car is in a “S” curve. The fourth image illustrates the scene of straight road.

To shorten the convergence time, an improved reward function is given as

$$reward = \text{abs}(cte_{prev}) - \text{abs}(cte)$$

(11)
The cte\textsuperscript{prev} means the previous cte. The improved reward function (11) make the training results converge to good policy in 2000 episodes from the experiment, which reduces about 1000 episodes compared with the equation (6).

Furthermore, we add an obstacle on the road as shown in Figure 10, to increase challenge. After trained by the improved reward function, the self-driving car bypasses the obstacle successfully.

The experiments demonstrate the feasibility of DDQN that training a self-driving car in the Unity Simulator and successfully transfer to reality.

4.3. **Comparison with the CNN-based self-driving scale car.** The self-driving approach based on CNN is another feasible method by mapping pixels from the camera input to the steering commands [2, 4]. The CNN-based algorithm requires people’s intervention to control the car and collect training data under different illumination and obstacles to enhance the robustness to environment. To compare these two algorithms, we make use of the same radio-controlled scale car and hardware to assemble a new car trained by CNN [19]. The experiments were conducted on the same road. Both throttle values were set to a constant (i.e. 0.7).
Figure 10. An obstacle is added on the road. The angle of view of the car in the lower left of the figure.

Theoretically, the scale car should drive within the specified range (within the range of the right lane) and turn smoothly.

Table 1. Performance of CNN, DDQN in the same road. The number means the times of the car outside.

|       | CNN  | DDQN |
|-------|------|------|
| 5(night) | 3    | 0    |
| 10(daylight) | 4    | 1    |
| 10(night) | 8    | 2    |
| 15(daylight) | 9    | 1    |
| 15(night) | 12   | 3    |

Table 2. Performance of CNN, DDQN in the same road with obstacle(s) and in five laps. The number means the times of the car hitting the obstacle(s).

|       | CNN  | DDQN |
|-------|------|------|
| 3     | 0    | 0    |
| 5     | 4    | 0    |

From the tables 1 and 2, it is clear that not only does the DDQN-based scale car perform better on the different illumination, but it also performs better the road with obstacles, depicted as figure 11. We also conducted experiments that an obstacle is put suddenly on the road when the scale car driving in the second lap. The DDQN-based method could avoid it successfully, but the CNN-based one failed. The reason is that the DDQN-baed scale car had been trained for 3 hours in the Unity simulation environment which contains virtually all the possibilities such as different obstacles, different illumination, and some sporadic conditions. However, the CNN-based relies on the dataset collected by people, it is not sufficient to contain different conditions. Therefore, the scale car trained by DDQN has strong robustness than CNN-based.
5. Conclusion and discussion. In this paper, a DDQN-based self-driving method was proposed. A car with a camera was trained in the Unity and then transferred to the reality, which was called by “sim-to-real”. From the experiments, the feasibility of training an automatic scale car through the virtual environment had been proved. The self-driving car trained by reinforcement learning needs to get some rewards from the environment, which may damage the car. To avoid the scale car damaging, the training process was finished in the virtual environment. Besides, the virtual environment can be developed under all the possible conditions of the driving, therefore, the trained car possesses better robustness to the environment.

The trained self-driving scale car achieved the goal of autonomous driving successfully, whereas the learned policy was also less stable, and the car wriggled frequently especially at the turn. From the analysis, we found that the useful background information and line curvature information were ignored. In return, the agent should be less prone to overfitting and can even be generalized to unseen and real-world tracks. However, it is no an easy task to bridge the gap between reality and virtual environment. In the future, we will adopt some sim-to-real tricks involved domain randomization, such as randomizing the width, color, friction of the track, adding shadows, randomizing throttle values etc., to guarantee the learned policy is robust enough to the real environment.

Currently, the reinforcement learning agent only generates steering output with constant the throttle value. In the future, the agent will learn to output an optimal throttle value. For example, it should learn to increase throttle value when the vehicle is driving straight and decrease throttle value when the car is making sharp turns.

Acknowledgment. This work is supported by Beijing Municipal Commission of Education and North China University of Technology, in part by the National Natural Science Foundation of China under Grant 61703006, in part by the Beijing Municipal Excellent Talent Program through the Youth Core Individual Project under Grant 2017000020124G027, in part by the Beijing Municipal Education Commission Science Plan through the General Research Project under Grant KM201810009001, in part by the Yu You Talent Planning of the North China University of Technology under Grant 107051360018XN012-011, in part by the Scientific Research Starting Foundation of the North China University of Technology under Grant 110051360002.
The authors would like to thank Tawn Kramer and Mr. Felix for creating a high fidelity Unity simulator for scale car. What we did was modifying their existing code to make it reinforcement learning compatible. We also want to thank the Donkey car community for initiating this wonderful project for us to learn about self-driving.

REFERENCES

[1] H. Abraham, C. Lee, S. Brady, C. Fitzgerald, B. Mehler, B. Reimer and J. F. Coughlin, Autonomous vehicles, trust, and driving alternatives: a survey of consumer preferences, Massachusetts Inst. Technol, AgeLab, Cambridge, (2016), 1–16.

[2] K. J. Aditya, Working model of self-driving car using Convolutional Neural Network, Raspberry Pi and Arduino, in 2018 Second International Conference on Electronics, Communication and Aerospace Technology, IEEE, 2018, 1630–1635.

[3] P. Andhare and S. Rawat, Pick and place industrial robot controller with computer vision, in 2016 International Conference on Computing Communication Control and automation, 2016, 1–4.

[4] C. Chen, A. Seff, A. Kornhauser and J. Xiao, Deepdriving: Learning affordance for direct perception in autonomous driving, in IEEE International Conference on Computer Vision, 2015, 2722–2730.

[5] Z. Chen and X. Huang, End-to-end learning for lane keeping of self-driving cars, in IEEE Intelligent Vehicles Symposium, IEEE, 2018, 1856–1860.

[6] F. Codevilla, M. Müller, A. Lopez, V. Koltun and A. Dosovitskiy, End-to-end driving via conditional imitation learning, in IEEE International Conference on Robotics and Automation, IEEE, 2018, 4693–4700.

[7] D. Dorr, D. Grabengiesser and F. Gauterin, Online driving style recognition using fuzzy logic, in 17th International IEEE Conference on Intelligent Transportation Systems, IEEE, 2014, 1021–1026.

[8] X. Liang, T. Wang, L. Yang and E. Xing, CIRL: controllable imitative reinforcement learning for vision-based self-driving, in Proceedings of the European Conference on Computer Vision, 2018, 604–620.

[9] L. J. Lin, Reinforcement Learning for Robots Using Neural Networks, Ph.D thesis, Carnegie Mellon University in Pittsburgh, 1993.

[10] R. R. Meganathan, A. A. Kasi and S. Jagannath, Computer vision based novel steering angle calculation for autonomous vehicles, in 2018 Second IEEE International Conference on Robotic Computing, 2018, 143–146.

[11] https://github.com/naokishibuya/car-behavioral-cloning

[12] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra and M. Riedmiller, Playing atari with deep reinforcement learning, preprint, arXiv:1312.5602.

[13] V. Mnih, et al., Human-level control through deep reinforcement learning, Nature, 518 (2015), 529–533.

[14] C. J. Pretorius, M. C. du Plessis and J. W. Gonsalves, The transferability of evolved hexapod locomotion controllers from simulation to real hardware, in 2017 IEEE International Conference on Real-time Computing and Robotics, 2017, 567–574.

[15] Understanding the Fatal Tesla Accident on Autopilot and the NHTSA Probe, Electrek, 2016. Available from: https://electrek.co/2016/07/01/understanding-fatal-tesla-accident-autopilot-nhtsa-probe/.

[16] M. Sadeghzadeh, D. Calvert and H. A. Abdullah, Self-learning visual servoing of robot manipulator using explanation-based fuzzy neural networks and Q-learning, Journal of Intelligent and Robotic Systems, 78 (2015), 83–104.

[17] R. S. Sutton and A. G. Barto, Reinforcement Learning: An Introduction, 2nd edition, Adaptive Computation and Machine Learning. MIT Press, Cambridge, MA, 2018.

[18] https://docs.donkeycar.com/

[19] https://github.com/autorope/donkeycar

[20] H. Van, A Guez and D Silver, Deep reinforcement learning with double Q-Learning, in National Conference on Artificial Intelligence, 2016, 2094–2100.

[21] D. Wang, J. Wen, Y. Wang, X. Huang and F. Pei, End-to-end self-driving using deep neural networks with multi-auxiliary tasks, Automotive Innovation, 2 (2019), 127–136.
[22] C. J. Watkins and P. Dayan, Q-learning, Machine Learning, 8 (1992), 279–292.

[23] T. Yamawaki and M. Yashima, Application of adam to iterative learning for an in-hand manipulation task, ROMANSY 22 Robot Design, Dynamics and Control, 584 (2019), 272–279.

E-mail address: zhangqi131072@gmail.com
E-mail address: tdu@ncut.edu.cn
E-mail address: cztianncut@126.com