Received: 19 March 2020 | Revised: 7 April 2021 | Accepted: 13 April 2021

IET Intelligent Transport Systems

DOI: 10.1049/itr2.12067

The Institution of Engineering and Technology WILEY

ORIGINAL RESEARCH PAPER

Deep Q-network implementation for simulated autonomous vehicle control

Yang Thee Quek[1] | Li Ling Koh[2] | Ngiap Tiam Koh[2] | Wai Ann Tso[2] | Wai Lok Woo[2,3]

[1] School of Engineering, Republic Polytechnic, Singapore

[2] School of Electrical and Electronic Engineering, Newcastle University International Singapore, Singapore

[3] Department of Computer and Information Sciences, Northumbria University, England, UK

Correspondence
Y. T. Quek, School of Engineering, Republic Polytechnic, 9 Woodlands Avenue 9, 738964, Singapore.
Email: quek_yang_thee@rp.edu.sg
W. L. Woo, School of Electrical and Electronic Engineering, Newcastle University International Singapore, 172A Ang Mo Kio Avenue 8 #05-01, 567739, Singapore.
Email: wailok.woo@northumbria.ac.uk

Abstract

Deep reinforcement learning is poised to be a revolutionised step towards newer possibilities in solving navigation and autonomous vehicle control tasks. Deep Q-network (DQN) is one of the more popular methods of deep reinforcement learning that allows the agent that controls the vehicle to learn through its mistakes based on its actions and interactions with the environment. This paper presents the implementation of DQN to an autonomous self-driving vehicle control in two different simulated environments; first environment is in Python which is a simple 2D environment and then advanced to Unity software separately which is a 3D environment. Based on the scores and pixel inputs, the agent in the vehicle learns and adapts to its surrounding. It develops the best solution strategy to direct itself in the environment where its task is to manoeuvre the vehicle from point to point on a simulated highway scenario. The implemented DQN technique approximates the action value function with convolutional neural network. This evaluates the Q-function for the Q-learning architecture and updates the action value function. This paper shows that DQN is an effective learning method for the agent of an autonomous vehicle. In both simulated environments, the autonomous vehicle gradually learnt the manoeuvre operations and progressively gained the ability to successfully navigate itself and avoid obstacles without prior information of the surrounding.

1 | INTRODUCTION

Autonomous self-driving vehicle is a vehicle that has the ability to perform navigation by perceiving its surrounding environment and making sense of it without the input or manual operation of human. The autonomous self-driving vehicle learns and makes sense of its environment and makes its own driving decisions such as obstacle avoidance and route planning. The Society of Automotive Engineers (SAE) defines six levels of driving automation in SAE J3016 ranging from 0 (fully manual) to 5 (fully autonomous).

Self-driving vehicle development is attracting much attention and interest because of its potential prospects such as increasing efficiency, lower accident rate, lower pollution, and minimising traffic jam. Autonomous vehicles can provide independent mobility for people who for any reason cannot or should not drive. Several automotive manufacturers and research institutions are investing into the research and development to bring the idea of autonomous self-driving vehicle to fruition [1]. Although in the recent years, we have seen autonomous vehicles placed on public roads for trial [2], there are still much works to be done in the technology development of autonomous self-driving vehicles.

The rapid development of machine learning and artificial intelligence in the recent years has aided the progress of autonomous self-driving vehicle, most notably in the camera perception. More recently, the introduction of deep learning coupled with advancements in computational power, especially in the Graphic Processing Unit (GPU), has increased the awareness and interest in applying Deep Neural Network (DNN) technology in the intelligent driving systems. DNN, which is the stacking of multiple layers of Artificial Neural Networks (ANN), has been shown to be beneficial in several applications such as sentence classification and speech recognition [3, 4]. The DNN architecture has been successful in learning feature representation, thus reducing the effort to manually engineer feature extractions. The number of hidden layers in a DNN can improve learning ability and task performance. Deep

Reinforcement Learning (DRL) technique [5–7] is one such technique that has improved significantly with the implementation of DNN, which has been used in various applications such as autonomous voltage control for power grid operations [8, 9], battery management system [10], network traffic signal control [11] and human–machine collaborations [12]

One of the commonly used DRL is Deep Q-network (DQN), which approximates Q-value function using DNN [13, 14]. In this paper, DQN is used to train the agent of the autonomous self-driving vehicle in two simulated environments. The agent in this paper makes use of DQN to learn about the surrounding environment and how to navigate the vehicle around obstacles. The implemented DQN in the self-driving vehicle's control approximates the action value function by Convolutional Neural Network (CNN) that evaluates the Q-function for the Q-learning architecture and updates the action value function. The vehicle eventually gains the ability to manoeuvre in the environment avoiding obstacles without prior information. The score given is based on speed of the vehicle and the number of obstacles it avoided. If the vehicle hits an obstacle, the simulation ends, and it will restart from the beginning. Thus, in order to obtain a high score, the vehicle must attempt to avoid collision with the obstacles while maintaining its speed for as long as possible.

Using simulated environments and scenarios in training and testing autonomous vehicle is a well-established practice. This paper is to develop an agent that corresponds to the point to point auto-cruise navigation with straight line driving and collision avoidance along a highway or expressway with no split roads or integrated traffic. The desired environments only include limited static elements of road layout and lane structure and dynamic elements of other cars only. Although there are several options of software in the market with sophisticated and complex scenarios however due to the nature of the above objective, it is necessary to keep the environment clean and simple with no frills. As such, the environments used were first created in python language with simple navigation and no speed control and advanced to Unity software with additional perceptions of the 3D environment and the option of varying speed. These two no frills environments allow effective demonstration of the algorithm.

The rest of the paper is organised as follows. Section 2 will provide a literature review of related work. Section 3 introduces the methodology and details of the algorithms used in the project. The two environments created to be used as the simulated environment are discussed in Section 4. Section 5 discusses the implementation of the technique and results. Concluding remarks and discussions of future works are provided in Section 6.

2 | RELATED WORK

Reinforcement learning (RL) is one subsection of machine learning that can be described as learning through trial and error [15, 16]. In [17], RL has been used to train the vehicle agent to learn an automated lane change behaviour such that it can intelligently make a lane change under diverse scenarios. The detection of lane marking is an important part of advanced driving assistance systems (ADAS) and in [18], CNN was used for lane marking detection. CNN is also used in obstacle avoidance is various applications, in [19], CNN-based single image obstacle avoidance is implemented on a quadrotor. The performance of the top three CNNs in road feature extraction was compared in [20] and a direct perception deep learning algorithm for autonomous driving was proposed in the paper. [21, 22] presented an end-to-end controller for steering autonomous vehicles based on CNN. The trained CNN directly mapped pixel data from a front facing camera to steering commands so that the agent manoeuvres the vehicle without hitting obstacles.

Q-learning algorithm belongs to a group of RL algorithm widely used for its simplicity. An example of Q-learning based neural network in learning action selection of mobile robot is discussed in [23]. The algorithm applied in the paper allows the autonomous mobile robot to select proper action in unknown environment for goal-directed obstacle avoidance.

In recent years, DRL has seen exciting development, especially in the research area of autonomous self-driving vehicle. [24] proposed a new control strategy of self-driving vehicles using DRL model, in which learning with an experience of professional driver and a Q-learning algorithm with filtered experience replay are proposed. [25] used a camera and a Lidar sensor in the car front and applied DQN to a simulated car in an urban environment.

This project applies DQN technique, which is a form of DRL technique, on the control of autonomous vehicle. [26] introduces the DQN method in a robot controller to explore a corridor environment with the depth information from a red green blue depth (RGB-D) sensor only. The paper in [27] demonstrated that DQN was able to perform very well in 49 games by just receiving only the pixel inputs and the game score as inputs. In fact, it was able to outperform human gamer. In another paper, [28], a similar approach was used to solve the autonomous car problem. It described the theory of RL and DQN and implemented the algorithm into "the open-source racing car simulator" (Torcs) driving environment to control the lane changing of the car. [29] applied DQN and deal with the simulation results of an autonomous car learning to drive in a simplified environment containing only lane markings and static obstacles using input images of the street captured by the car front camera. Similarly, DQN is implemented in [30, 31] to navigate the autonomous self-driving vehicle in highway scenarios. The training is also done in simulations.

3 | METHODOLOGY

This paper tackles the task of autonomous vehicle with the implementation of DQN in the agent that control the vehicle. This project extends the general Q-learning RL algorithm into Deep Q-network with the integration of CNN. In this section, the CNN is first introduced, followed by the RL model. Then the Q-learning, a model-free reinforcement learning method, is discussed. The last sub-section will elaborate the expansion of Q-learning into DQN.

3.1 | Convolutional neural network

Convolutional neural network (CNN) is preferred over other classification techniques because it makes efficient use of patterns and structural information in an image and demonstrates high effectiveness in recognising and classifying images. This is particularly useful for this project as the input data is the pixel frame of the environment. It is a deep learning module that extracts feature representations through back-propagation. CNN mainly comprises of the following.

(i) Convolutional layers
(ii) Activation functions
(iii) Pooling layer
(iv) Fully connected layers

Images are read as pixels in the convolutional layers and each image is expressed as a matrix of $H \times W \times D$ (height × width × depth). A set of learnable filters is applied to the convolutional layer. The filter works by identifying the existence of unique features or arrangements that exist in the input image. It alters the dimension of the original matrix to the size of $H \times W \times 3$. The reason for the depth to be 3 in this project is because each pixel has three colour channels, red, green and blue. This filter is convolved along the height and width of the image, after which a dot product is performed to produce an activation map.

An input image has millions of pixels depending on the size. Local Receptive Field represents a small portion of the input layer holding each neuron. The neurons represent input pixels that passes through a convolutional layer. Each local receptive field has a unique hidden neuron located in the convolutional layer. In the hidden layer, a neuron consists of shared weights and bias. The weight represents the extract features from the input units. Due to the weights, it allows feature detection regardless of the environment or the position of the image. All hidden neurons can detect a similar feature regardless of the local receptive field in different locations. The bias represents a random value that would be added with shared weights.

In a neural network, activation functions are essential for the learning process due to the complexity between the input units and the response variable. It serves to convert an input unit of a neural network to an output unit. neural network uses nonlinear activation function such as Sigmoid, Tanh, ReLU or other trainable activation layer.

Comparing with other activation function, the ReLU activation function is more effective with lower runtime and less computation costs, see Equation (1) where z is the input. It is also preferred as it avoids the vanishing gradients problem.

$$\text{ReLU}\ (z) = \begin{cases} 0 & \text{when} \quad z < 0 \\ z & \text{when} \quad z \geq 0 \end{cases} \tag{1}$$

This project uses Leaky ReLU, which is ReLU activation function with a small modification of having a small negative slope when $x < 0$, as seen in Equation (2). This modification prevents ReLU neuron to always output 0 if it is stuck in the negative side. The issue usually occurs when the learning rate is too high or there is a large negative bias.

$$\text{LReLU}\ (z) = \begin{cases} 0.01z & \text{for } z < 0 \\ z & \text{for } z \geq 0 \end{cases} \tag{2}$$

Moving on from the activation function is the pooling layer that reduces the number of parameters and calculations done in the network. This is also known as down-sampling. This takes sections of the image and aggregates them into one solitary value. There are two main methods for pooling: max pooling or average pooling [32]. This project uses the max pooling method that takes the largest element in the pooling sector. Max pooling extracts the sharpest features of an image. So given an image, the sharpest features are the best lower-level representation of an image. Equation (3) depicts the max pooling method where h^l is the activation of layer l.

$$h^l_{xy} = \max_{i=0\ldots s, j=0\ldots s} h^{l-1}_{(x+i)(y+j)} \tag{3}$$

The fully connected layer represents the interconnection from all previous neuron from each individual layer. As the convolutional and pooling layers consists of distinct feature from the input units, the fully connected layer interconnects these features for classifying or recognition purposes. Even though convolutional and pooling layers holds distinct features which is enough for classification tasks, a combination of it would increase performances. The fully connected layer utilises the probability output which equates to 1. An activation function known as "softmax", see Equation (4), which is placed in the output layer of the fully connected layer to ensure that the summation of the probability equates to 1.

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}} \tag{4}$$

Overall, the convolutional layer processes the image or video using its pixels to the agent to identify the obstacles. Figure 1 illustrates the CNN model.

3.2 | Reinforcement learning

The fundamental principle of reinforcement learning is deeply related to the psychology of human or animal behaviour. When a toddler moves its limbs, turns its head, or fidgets about; it is interacting with its environment. It does not have an explicit teacher, but it does have a way to receive information from its environment through its senses. The abundance of information received conveys the consequences of its actions, correlates cause and effects, and what actions to take to accomplish its objectives. These interactions provide a vast source of knowledge regarding the environment and situation.

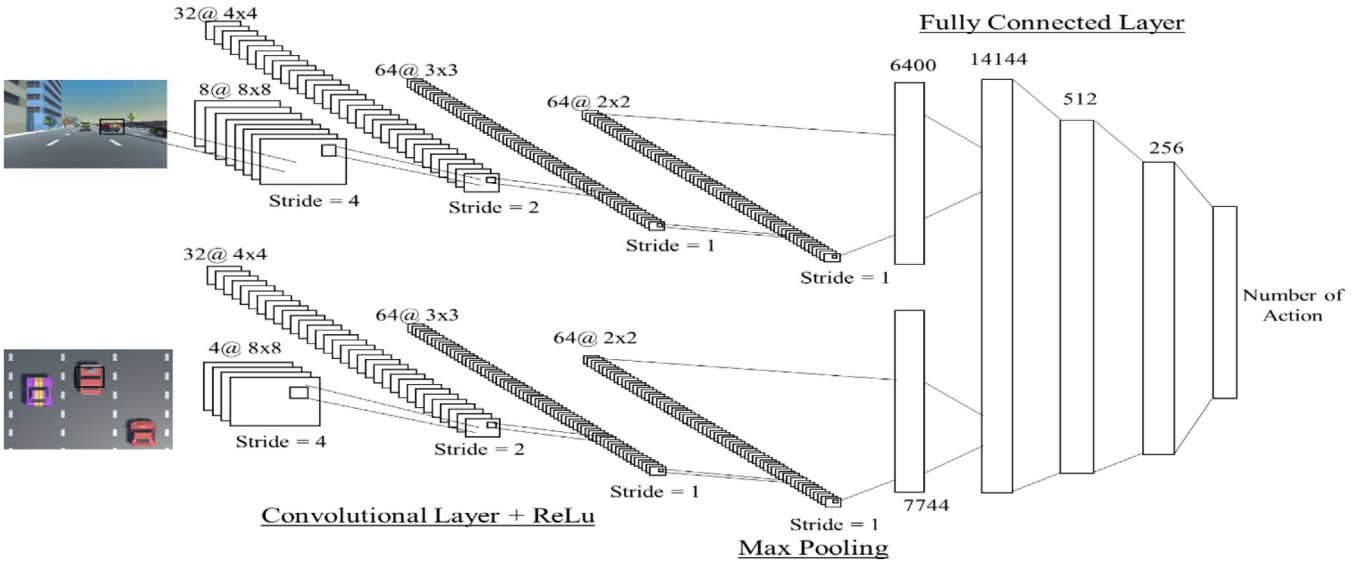

FIGURE 1 Reinforcement learning interaction

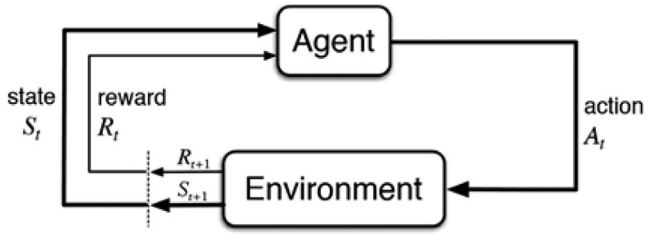

FIGURE 2 Reinforcement learning interaction

As such, in RL, there is no external feedback or answer key to the problem, but the agent of the vehicle is to decide how to act in each situation. In its computational form, RL does not simply refer to a machine-learning paradigm but also a learning problem. In order to solve this learning problem, RL seeks a method of control to influence an agent into obtaining its long-term goal of maximising a numerical reward from an environment. Figure 2 illustrates the RL interaction.

In this project, the vehicle has a current state, *s*, of the environment, *E*. The vehicle can choose to perform an action, *a*, which will translate into next state, *s'*. Each action is to be evaluated by a reward *r* and the set of action the vehicle chooses to perform defines the policy Π. The goal is to select the right policy such that the rewards are maximised. Equation (5) shows the sum of all rewards the agent can accumulate with the introduction of discount factor, γ. The discount factor, γ, decides whether the agent priorities short-term or long-term rewards and is typically set to a value closer to 1 (usually 0.9) to focus on the long-term rewards.

$$R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} + \cdots + \gamma^{n-t} r_n \quad (5)$$

The RL scenario is in Markov decision process (MDP) framework where the agent interacts with its environment and progresses through the state transitions. The transition goes from starting state to the terminal state and then the environment resets and begins again. A single sequence of these transitions is called an episode. The state transition diagram is as shown in Figure 3. The Markov property is defined as the probability of the next state depends solely on current state and action. The number of states and reward values are considered finite to simplify the mathematics.

In order to solve RL problems in MDP framework, two assumptions are made:

- Markov property is applicable
- MDP assumes that there are finite transitions of states, actions and rewards in an episode.

3.3 | Q-learning

Q-learning is a model-free reinforcement learning method. This method is useful for solving self-driving problem where the agent does not know the model environment. The agent only knows the number of possible states in the environment and the actions that are available in each state.

In Q-learning, each state is assigned with an estimated value, called a *Q*-value. This *Q*-value is the comparable to the value function and is also known as the action value function. They both similarly represent the total future rewards. However, the value function describes how good a state is, while the *Q*-value represents how good it is to select a certain action, in a particular state, following an optimal policy.

The *Q*-value is denoted by:

$$Q_\pi(s,a) = E_\pi\left[r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} | S_t = S, = A_t = a\right] \quad (6)$$

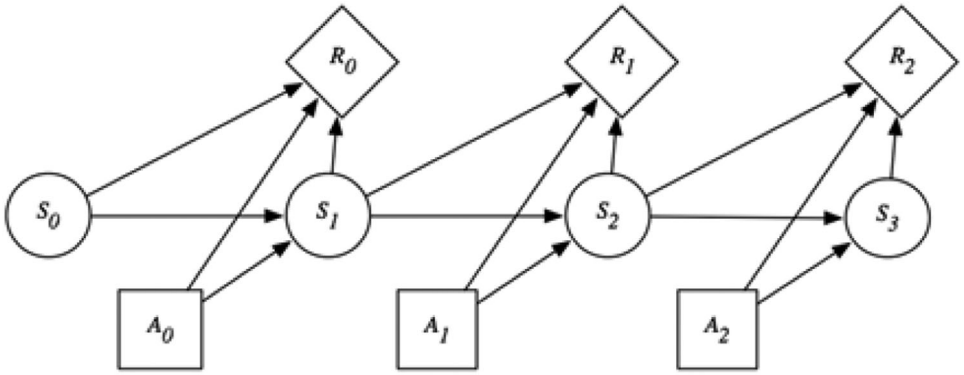

FIGURE 3 State transition diagram

The Bellman's equation for the Q-value is rewritten as:

$$Q_\pi(s,a) = \sum_{s',r} p(s',r\,|s,a)\left[r + \gamma \sum_{a'} \pi(a'\,|s')Q_\pi(s',a')\right] \tag{7}$$

By follow the best policy or optimal policy, the Q-value obtained will also be optimal. This value can be termed the optimal Q-value and is defined as:

$$Q_*(s,a) = \sum_{s',r} p(s',r\,|s,a)\left[r + \gamma \max_{a'} Q_\pi\left(s',a'\right)\right] \tag{8}$$

Thus, Bellman's equation can simply be written as:

$$Q\left(s,a\right) = r + \gamma \max_{a'} Q\left(s',a'\right) \tag{9}$$

Equation (9) denotes that the Q-value for the state, action pair of (s,a) is equals to the immediate reward of taking action a while in state s, plus the discounted optimal value of the state-action pair for the successive or next state. Every time the agent passes through the state, the Q-value is updated as the environment could be stochastic causing the value to be different. Hence, iteratively updating the Q-value for each of the state until it reaches optimality or convergence is the key to the Q-learning algorithm.

The updating of the Q-value is achieved by applying Equation (10). The learning rate is the rate at which the agent learns after each iteration. It is a probability of 0 to 1, if it is closer to 1, than the old Q-value is updated close to the new Q-value. It is usually set to a value closer to 0, (usually 0.01) as it is desired that the agent learns in small increments.

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha \cdot [r_{t+1} + \gamma max_a(Q(s_{t+1},a_{t+1})) - Q(s_t,a_t)] \tag{10}$$

The learned value is the new Q-value estimated using the Bellman's equation. The goal is to incrementally update the existing Q-value with the difference between the old Q-value and the new estimate.

3.4 | Exploration vs exploitation (greedy policy)

The goal of agent is to try to accumulate the maximum future rewards also known as Q-values to complete a task. It is crucial to find a balance between continuing to explore and exploiting the maximum know Q-values is crucial. A good solution to this issue is to implement a ε-greedy policy where ε is a probability between 0 (Exploitation) to 1 (Exploration). The agent is allowed to take a random action at a probability of ε instead of always choosing the 'greedy' action (the action with the largest Q-value). It is a good practice to set ε at 1 in the beginning so that there is a 100% chance that the agent will act randomly. As the agent continues to explore, ε is gradually decayed towards increasingly exploitative. This decay can be done using a decay rate, where ε is decreased by the decay rate at every time step.

3.5 | Deep Q-network

The Q-learning algorithm uses a matrix to store the Q-values. This method is feasible in an environment where the number of states is relatively small. However, the project of creating a simulation for autonomous cars requires the environment to be much more complex, considering the pixel data on the screen of the simulation as the number states. For instance, each screen has a size resolution of 80 by 80 pixels for a total 6400 pixels. Each pixel has 256 possible values when converted to greyscale. This adds up to an enormous 256^{6400} states.

The agent can have multiple actions in each state; thus, the number of Q-values equates to multiplying the number of actions by 256^{6400}. In this context, it is unmanageable to store such a large set of Q-values in a matrix or by other tabular means. This is where DQN is beneficial.

The Q-matrix is replaced with a function approximator, more specifically an artificial neural network. The neural network accepts the states (pixels) as its input and estimates a

corresponding Q-value as its output. There are multiple Q-values output by the neural network, one for each action available to the agent. The agent chooses the action that corresponds to the largest Q-value.

3.6 | Double DQN

DQN is known to, at times, learn unrealistically high action values because it includes a maximisation step over estimated action values, which tends to favour overestimated to underestimated values. If the overestimations are not uniform and not concentrated at states about which the agent wishes to learn more, then this might give negative effect to the quality of the resulting policy. Thus, Double DQN is applied to reduce the effects of overestimating the Q-values by decomposing the max operation in the target into action selection and evaluation selection.

Double DQN algorithm is as shown below.

Double DQN algorithm

Initialise replay memory
Initialise primary network $Q(s, a)$ arbitrarily with random weights
Initialise target network $Q\_\text{target}(s, a)$ arbitrarily
iteration = 0
Repeat (for each episode):
  Initialise s
    Repeat (for each step of episode):
      choose action a
      with probability ε choose random a
      else choose $a = argmaxQ(s, a')$
  take action a, observe r, s'
  store experience (s, a, r, s') in replay memory
  sample random minibatches of (s, a, r, s')
  from replay memory
  calculate y for each minibatch
    if s' is end of episode, $y = reward$
    else
      $y = reward + \gamma Qtarget(s', argmax(Q(s', a')))$
  train primary Q network with $Loss = y - Q(s, a)^2$
  $s \leftarrow s'$;
  iteration ++1
  if iteration = updatenum
    update target network weighs = primary network
Until s is terminal

Double DQN employs two networks: a main network for the action selection and a target for policy evaluation. In Double DQN, the target Q-value y is calculated from equation in order to find the loss function and train the primary network. The greedy policy decides upon the max values which action to select. This allows the target network to select the action and at the same time evaluates its quality. After every few iterations, the weights in the target network are updated to be the same as the primary network. This helps with the stability. The target network Q-value is calculated by:

$$y = \text{reward} + \gamma Q\ \text{target}(s',\ \text{argmax}(Q(s', a'))) \quad (11)$$

3.7 | Experience replay

As the agents explores its environment, all the transitions of (s, a, r, s') are accumulated in a replay memory. Random samples from the replay memory called minibatches, are used during the training of the network. By not using the latest transitions to train the network, there would be less correlation between the training samples. This prevents the neural network from deriving at a local minimum.

4 | ENVIRONMENTS

The target task for the vehicle is to manoeuvre and navigate around the problem environment without collision to the walls. It is risky and cost-ineffective to train a vehicle in a real-time real-life environment such as a residential area, heavy traffic urban setting or an expressway. Thus, this paper proposed applying vehicular control training and learning in simulation environment. Simulation environments allow customisation of the environment at the same time eliminate any risk factors. Two simulated environments created are discussed in the following sub-sections.

4.1 | Python-coded environment

In the python-coded environment, the agent moves the vehicle up or down, in a 2D environment. The obstacles come in the form of pillars that the vehicle aims to avoid, see Figure 4.

If it collides with any of them, the episode ends and the interface resets. There is a score at the bottom left of the screen that increases by one every time the car avoids an obstacle. The score is also an indication of how well the agent is performing. The higher the score, the better the agent is doing. The score also corresponds to the reward given to the agent. The frames of this interface in fed into the convolutional neural network as the states of the environment.

In the experiments and simulations, the DQN rewards are recording with reference to result of the action taken by the vehicle. Tables 1 and 2 shows examples of the logging of DQN reward after the vehicle performed an action in various stages such as the "OBSERVE" and "TRAIN" stages respectively. The parameter action refers to the movement of the vehicle; it will be labelled as 0 if no action is required and 1 if otherwise. It is shown that the rewards are recording correctly and the

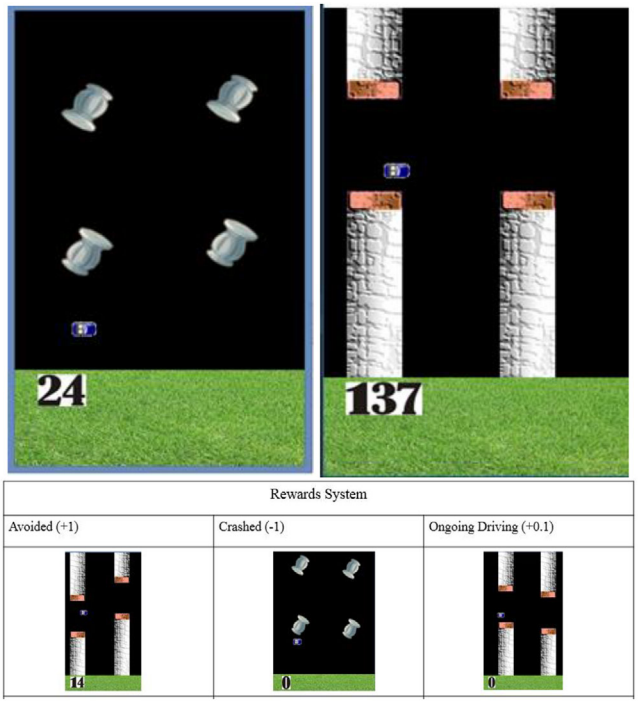

FIGURE 4 Python-coded environment for autonomous vehicle simulation

TABLE 1 Example of recording of rewards in "OBSERVE" stage

Stage	Action	Reward
OBSERVE	0	0.1
OBSERVE	1	1
OBSERVE	1	0.1

integration between the algorithm and the interface is successful where the agent can interact with the environment based on the rewards in stages while the Q-values are being compute. This shows that the Q-values is based on the rewards.

4.2 | Unity environment

Unity provides an environment for self-driving vehicles to optimise learning models such as neural networks for navigations such as steering angle and direction. It is a well-structured development platform that offers incredible feature-rich tools to work on when creating a simplistic game environment interface. The attractive graphics layout and design on 2D interface makes

TABLE 2 Example of recording of rewards in "TRAIN" stage

Stage	Action	Rewards
TRAIN	0	1
TRAIN	0	0.1
TRAIN	1	0.1

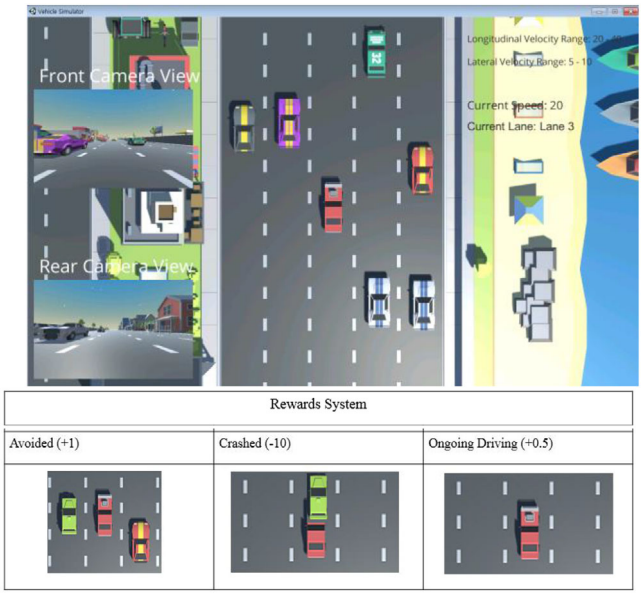

FIGURE 5 Unity software environment for autonomous vehicle simulation

it one of the most popular software users choose to create environments.

In the Unity interface (see Figure 5), the agent able to control a car in various direction such as forward, backward, left and right using a camera view to allow agent to sense upcoming unforeseen circumstances ahead avoid obstacles. The obstacles come in the form of multiple type of cars driving on road. The car receives reward based on the obstacles it manages to avoid. The rewards indicate how well an agent perform on the car. With an increase number of rewards, it demonstrates the performance of the agent is reliable to handle real life situation, as the interface is created in 3D environment similar to real life scenario.

5 | IMPLEMENTATION AND RESULT DISCUSSION

In this program, there are three stages of learning for the agent, "OBSERVE", "TRAIN" and "TEST". In the "OBSERVE" stage, the agent merely acts randomly and observes the environment. It takes the transitions of state, action, reward, next state $(s, a, r, s')$and stores them inside a replay memory and not training of the network occurs.

The variables that were considered include staying in a single driving lane, progress around the track, speed (S) as a fraction of max speed and going off-road (O) or colliding with other vehicles/stationary objects on the course (C). The rewards given to the agent in this project is as follows:

$$R = \begin{cases} -10, & C \text{ or } O \\ -1, & S = 0 \\ \min\left(10,\ 0.2 + 5S\right), & \text{otherwise} \end{cases} \tag{12}$$

where

$$S = \frac{\text{current speed}}{\text{maximum attainable speed}} \tag{13}$$

After observing for the set number of time-steps, the "TRAIN" stage begins. The agent selects actions based on the greedy-policy and the greedy factor epsilon is decreased after every time step. New transitions of (s, a, r, s') are still continually stored in the replay memory. Using random samples of (s, a, r, s') taken from the reply memory, the CNN produce a prediction of $Q(s, a)$ in a forward propagation for each of the actions available in a particular state. A target value y is computed using the actual reward obtained and the next state observed by apply the Bellman's Equation (9).

$$\text{Loss} = (r + \gamma \max_a Q(s', a') - Q(s, a))^2 \tag{14}$$

A loss function (14) is computed by squaring the difference between the target value and the predicted value. After the "TRAIN" stage will be the "TEST" stage. At this point, the agent already has the true Q-values. The greedy factor is reduced to its final value. The agent no longer explores but instead exploits the maximum Q-values.

Below sub-section discusses the results obtained from the simulation in the two environments.

5.1 | Results and discussion from python-coded environment

In the Python-coded environment, the obstacles came in the form of tiny blocks and these blocks are randomly placed in the environment. The parameters of actions taken, reward observed, and maximum Q-values computed were logged and printed out at every time-step to track the agent's progress. The agent was trained in the program for about 8 h.

Due to the high dimensional state, the image screen needs to be pre-processed to a smaller pixel size to reduce the dimensional and state space before the input into the CNN architecture. The original image size of 512 × 288 pixels is converted into greyscale before furthering reducing it to 80 × 80 pixels. Furthermore, it has been normalised from a (0,255) to a (0, 1.0). Based on the last four frame stacks, the CNN inputs are 80 × 80 × 4 pixels.

The input of 80 × 80 × 4 image size is being fed into a convolutional 2D layer 8 × 8 × 4 × 32 with a ReLU activation function by addition of a bias of 32 with a stride of 4. With reference to Figure 1, the 8 × 8 is the convolutional 2D layer, while the 4 represents the amount of frame stack together. The 32 is the amounts of frames produced each time it passes through the convolutional 2D layer resulting the 8 × 8 × 4 × 32 configuration. The output passed through a max pooling of 2 × 2 with a stride of 1 to retain original output size. The previous output is being fed into the second convolutional layer of 4 × 4 × 32 × 64 with a ReLU activation function followed by an addition of bias

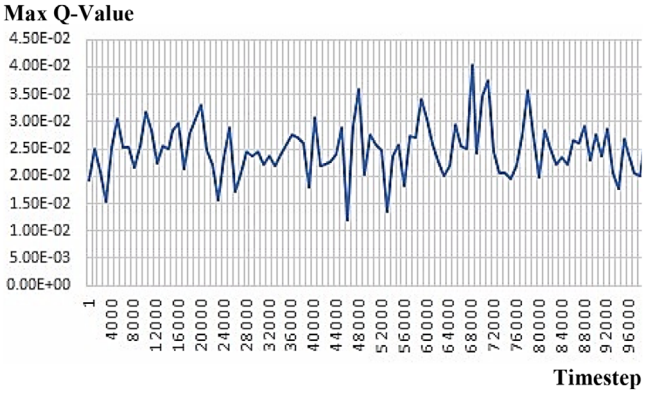

FIGURE 6 Max Q-value for "OBSERVE" stage

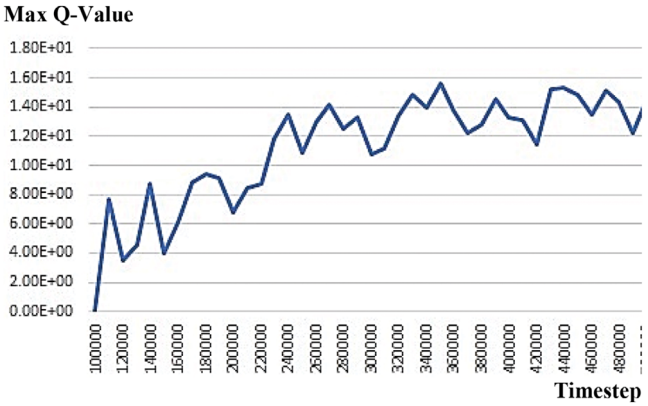

FIGURE 7 Max Q-value in the "TRAIN" stage

of 64 with a stride of 2. The output is then pass through the same maxpooling layer. Lastly, the output is being fed to the third convolutional layer 3 × 3 × 64 × 64 with a ReLU activation function followed by adding a bias of 64 with a stride of 1 retaining the output pixel size. The output is a fully connected layer with 1600 to produce a 512 with a bias of 512 which is further fully connected of 512 to the number of actions needed. The fully connected layer uses a ReLU activation function and matrix multiplication of the output with the weights of the fully connected layer in addition to the bias in the fully connected layer to produce an output.

During the "OBSERVE" stage, the agent acted randomly at very insignificant value of around 0.025 and therefore kept colliding with the obstacles. This corresponds to the initial stage of the DQN learning process, thus the Max Q-value fluctuated without improvement as seen in Figure 6.

As the agent continues to be trained, this corresponds to the "TRAIN" stage. It is shown in Figure 7 that max Q-value increases along each timestep, and it was observed that the agent acts less erratically in the simulation as it begins to learn the true Q-values. After several episodes, the max Q-value also appears to begin to stabilise, as the values get more consistent around 14 with smaller deviations.

In the final "TEST" stage, obstacles were changed into wall-like structure to determine the DQN performance. The

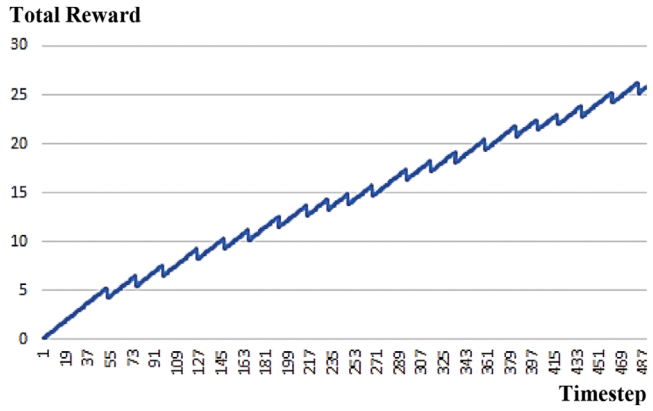

FIGURE 8 Reward against Timestep during training phase

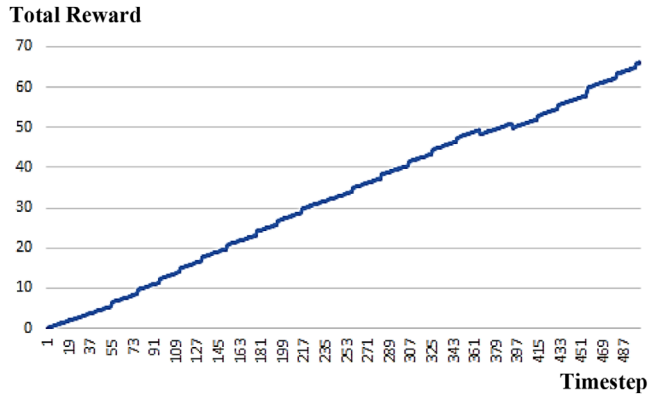

FIGURE 9 Reward against timestep after completion of training

initial phases have many routes for the simulated vehicle to pass through the obstacles that lead to easier performance results. The final phase would determine the DQN performance with only one path for the vehicle through. The structure of the DQN is identical as the initial phase to evaluate the performance without any extra enhancement.

Reward is proportional to the program efficiency and performance index, in this project, they refer to the speed of the autonomous vehicle and the score respectively. The reward helps the Q value to obtain the max Q value by taking the previous reward and going through the predicted value to determine the next course of action. Figures 8 and 9 are the reward plots obtained from the simulations. The key description of the reward plots is the gradient of the reward curve, i.e. rate of the change of reward over the timestep. Figure 8 plots the reward value against the timestep before the agent learns how to avoid the obstacles.

It shows the slow accumulation of reward since the agent is still in the learning phase. The presence of saw-tooth patterns, which is caused by the sharp decrease in the reward value, in the plot reveals that the agent was occasionally penalised due to taking the action of stopping (hitting an obstacle).

Figure 9 shows the reward plot after the agent has been trained or learnt how to avoid the obstacles. The rapid accumulation of the reward without the sharp decrease in the reward value indicates that the agent has learnt to avoid obstacles by changing lanes and staying in lane with maximum attainable speed when there is no obstruction.

TABLE 3 Q-values against the number of hours

Time	0 h	2 h	4 h	6 h	8 h
Q-values	0.02294508	7.034746	12.12914	12.52646	12.67253
	0.02226432	8.277883	12.19345	12.70074	12.71674
	0.02412466	8.084210	12.29293	12.88766	12.90756
	0.01814430	8.504424	12.50941	12.96565	12.98205

As every few second the application is producing out the Q-values, it is not ideal to present all the values here. Thus, in Table 3 below, only the first four Q-values based on the hours given were shown. It is shown that the agent has a high tendency to collide into obstacles in the first 2 h. This is due to the agent having low Q-value as there is little training involved. As the agent went through more hours of training, the Q-value increased exponentially resulting in lower frequency of collision into obstacles. This indicates that the agent has learnt that it can achieve higher score which corresponds to the increase in Q-value by avoiding obstacles. The Q-value converged and fluctuated in a small percentage between certain values when the application had reached a certain score and there was nothing further to learn.

The above observations showed that the integration between the DQN and the interface was successful. The agent interacts with the environment and rewards are being received while Q-values are being computed. It is observed that the rewards and actions affect the Q-values as previously theorised.

5.2 | Results and discussions from Unity environment

The work progressed from the above-mentioned 2D python environment which only has 2 controls to the more complex 3D Unity environment that allows more controls. The Unity convolutional network is much different from the previous simulation done in python as it passes through three convolutional layers first before passing through a pooling. The network has two types of images: "image network" and "map network". This is much different from the first program where the vision was only image network. In this Unity environment, as seen in Figure 5, the Double DQN utilises both the front camera and the bird eye view of the map; both are interconnected using a fully connected layer before an action is chosen.

Based on Figure 1, the parameters of the convolutional neural network of unity car are given by first convolutional image is [8, 8, 8, 32] while the first convolutional map is [8, 8, 4, 32]. The following neural network is the same with the second convolutional is [4, 4, 32, 64] and third convolutional is [3, 3,64,64]. This is passed through a pooling layer of [1, 2, 2, 1]. The fully connected layer for the image is [6400, 1024] while the map is [7744, 1024]. Combining both of it to a [14144, 512] follow by a

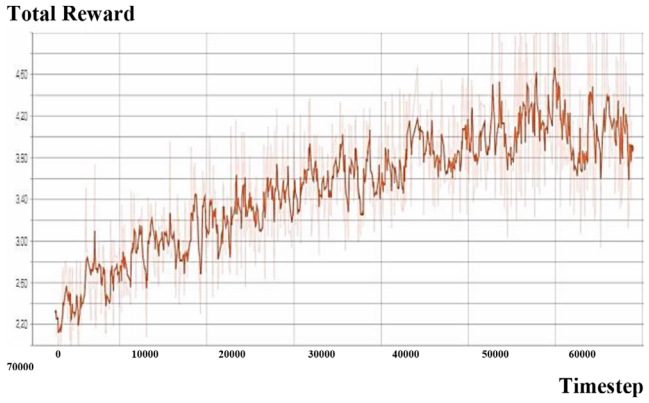

FIGURE 10 Reward against timestep

second fully connected layer [512, 256] to the last layer of [256, number of actions].

Based on these parameters, the CNN is formed to assist the loss function that help to generate the best course of action for the vehicle. The reward based on timestep would further depicts the performance of the simulation.

It is observed that when the agent was untrained, it acts randomly and constantly changes lanes and varies its speed without clear aims. It will also frequently collide with other cars. The double DQN successfully trained the agent as its performance steadily improves. As seen in Figure 10, it is observed that the total rewards increased in an upward trend which indicates the agent is constantly learning until it reaches steady state. Based on Figure 10, the total reward versus timestep graph illustrates that the agent is learning with more controls to gain the highest score possible. The skillset to accelerate while switching lane requires much time to achieve, but eventually, the agent is able to learn and understand that not switching lane unless necessary will gain higher score. After training, it is observed that the autonomous vehicle is able to avoid the traffic much easier and does not change lanes as often as it did in the beginning. It will try to achieve its maximum speed for best result.

6 | CONCLUSION

Developing the control strategy of an agent in an autonomous vehicle is not easy, given that the agent is oblivious to the environment and with no prior instructions provided. This project implemented the use of DQN in training the agent to autonomous manoeuvre a vehicle in simulated environments. The reward in this paper refer to the speed of the autonomous vehicle and the score. This is used to help the Q value to obtain the max Q value by taking the previous reward and going through the predicted value to determine the next course of action. To implement the approach, two simulated environments were developed using Python and in Unity software separately. Experimental results presented in in this paper showed that, after hours of training, the agent can obtain effective control strategies and successfully control the vehicle in both environments. Thus, the agent learnt to control the vehicle to avoid obstacles while attempting to maintain its maximum speed.

7 | FUTURE WORK

Understanding the DQN behaviour is paramount. There could be a higher potential if the agent is provided with more controls similar to real-life applications. This paper presented work on the 2D Python and 3D Unity environments, however, placing the agent in more different set of environments and not limiting it to just 3D simulation potentially could give a more promising result. There could also be more work done in terms of training the agent in a more complex program where the environment has better resemblance to the real world. Different sensors integration into the simulation with the DDQN could also improve the performance as more controls are taken into consideration.

Other future work in this project can also involve the training of agents in other RL algorithms such as asynchronous actor–critic (A3C) method and deep deterministic policy gradient method and compare the results. Another avenue for future work could be to apply the DQN methods in actual small-scale vehicles and test out the algorithm in actual test tracks or environment.

ORCID

Yang Thee Quek https://orcid.org/0000-0003-1981-1607

REFERENCES

1. Bagloee, S.A., et al. Autonomous vehicles: Challenges, opportunities, and future implications for transportation policies. J. Modern Transport. 24, 284–303 (2016)
2. Guizzo, E.: How Google's self-driving car works. IEEE, (2011). https://spectrum.ieee.org/automaton/robotics/artificial-intelligence/how-google-self-driving-car-works. Accessed on October 2011.
3. Zhao, R., Mao, K.: Topic-aware deep compositional models for sentence classification. IEEE/ACM Trans. Audio, Speech, Language Process. 25, 248–260 (2017)
4. David, K.B.H., et al. Deep temporal convolution network for time series classification. Sensors 21, 603 (2021)
5. Arulkumaran, K., et al.: Deep reinforcement learning: A brief survey. IEEE Signal Process Mag. 34, 26–38 (2017)
6. Al-Nima, R.R.O., et al.: Robustness and performance of deep reinforcement learning. Appl. Soft Comput. 105, 107295 (2021).
7. Koh, B.H.D., Woo, W.L.: Multi-view temporal ensemble for classification of non-stationary signals. IEEE Access 7, 32482–32491 (2019)
8. Duan, J., et al.: Deep-reinforcement-learning-based autonomous voltage control for power grid operations. IEEE Trans. Power Syst. 35, 814–817 (2020)
9. Quek, Y.T., Woo, W.L., Logenthiran, T.: Load disaggregation using one-directional convolutional stacked long short-term memory recurrent neural network. IEEE Syst. J. 14, 1395–1404 (2020)
10. Wei, Q., Liu, D., Shi, G.: A novel dual iterative Q-learning method for optimal battery management in smart residential environments. IEEE Trans. Ind. Electron. 62, 2509–2518 (2015)
11. Xie, D., et al.: IEDQN: Information exchange DQN with a centralized coordinator for traffic signal control. In: *2020 International Joint Conference on Neural Networks*, pp. 1–8.IEEE, Piscataway, NJ (2020)
12. Woo, W.L.: Future trends in I&M: Human-machine co-creation in the rise of AI. IEEE Instrum. Meas. Mag. 23, 71–73 (2020)

13. Kiran, B.R., et al.: Deep reinforcement learning for autonomous driving: A survey, IEEE Trans. Intell. Transport. Syst. (2021). https://doi/10.1109/TITS.2021.3054625
14. An, H., Jung, J.-I.: Decision-making system for lane change using deep reinforcement learning in connected and automated driving. Electronics 8, 543 (2019)
15. Sutton, R.S., Barto, A.G., Williams, R.J.: Reinforcement learning is direct adaptive optimal control. IEEE Control Syst. Mag. 12, 19–22 (1992)
16. Zhang, L., et al.: Decentralized control of multi-robot system in cooperative object transportation using deep reinforcement learning. IEEE Access 8, 184109–184119 (2020)
17. Wang, P., Chan, C., Fortelle, A.d.L.: A reinforcement learning based approach for automated lane change maneuvers. In: *2018 IEEE Intelligent Vehicles Symposium*, pp. 1379–1384.IEEE, Piscataway, NJ (2018)
18. Ye, Y.Y., Hao, X.L., Chen, H.J.: Lane detection method based on lane structural analysis and CNNs. IET Intel. Transport Syst. 12, 513–520 (2018)
19. Chakravarty, P., et al.: CNN-based single image obstacle avoidance on a quadrotor. In: *2017 IEEE International Conference on Robotics and Automation*, pp. 6369–6374.IEEE, Piscataway, NJ (2017)
20. Al-Qizwini, M., et al.: Deep learning algorithm for autonomous driving using GoogLeNet. In: *2017 IEEE Intelligent Vehicles Symposium*, pp. 89–96.IEEE, Piscataway, NJ (2017)
21. Rausch, V., et al.: Learning a deep neural net policy for end-to-end control of autonomous vehicles. In: *2017 American Control Conference*, pp. 4914–4919.IEEE, Piscataway, NJ (2017)
22. Yang, Z., et al.: End-to-end multi-modal multi-task vehicle control for self-driving cars with visual perceptions. In: *2018 24th International Conference on Pattern Recognition*, pp. 2289–2294.IEEE, Piscataway, NJ (2018)
23. Qiao, J., Hou, Z., Ruan, X.: Q-learning based on neural network in learning action selection of mobile robot. In: *2007 IEEE International Conference on Automation and Logistics*, pp. 263–267.IEEE, Piscataway, NJ (2007)
24. Xia, W., Li, H., Li, B.: A control strategy of autonomous vehicles based on deep reinforcement learning. In: *2016 9th International Symposium on Computational Intelligence and Design*, pp. 198–201.IEEE, Piscataway, NJ (2016)
25. Fayjie, A.R., et al.: Driverless car: Autonomous driving using deep reinforcement learning in urban environment. In: *2018 15th International Conference on Ubiquitous Robots*, pp. 896–901.IEEE, Piscataway, NJ (2018)
26. Tai, L., Liu, M.: A robot exploration strategy based on Q-learning network. In: *2016 IEEE International Conference on Real-time Computing and Robotic*, pp. 57–62.IEEE, Piscataway, NJ (2016)
27. Mnih, V., et al.: Human-level control through deep reinforcement learning. Nature 518, 529, (2015)
28. El Sallab, A., et al.: Deep reinforcement learning framework for autonomous driving. arXiv:1704.02532, (2017)
29. Okuyama, T., Gonsalves, T., Upadhay, J.: Autonomous driving system based on deep Q learning. In: *2018 International Conference on Intelligent Autonomous Systems*, pp. 201–205.Academic Press, Waltham, MA (2018)
30. Ronecker, M.P., Zhu, Y.: Deep Q-network based decision making for autonomous driving. In: *2019 3rd International Conference on Robotics and Automation Sciences*, pp. 154–160.IEEE, Piscataway, NJ (2019)
31. Kashihara, K.: Deep Q learning for traffic simulation in autonomous driving at a highway junction. In: *2017 IEEE International Conference on Systems, Man, and Cybernetics*, pp. 984–988.IEEE, Piscataway, NJ (2017)
32. Lee, P.G.C.-Y., Tu, Z.C.: Generalizing pooling functions in CNNs: Mixed, gated and tree. IEEE Trans. Pattern Anal. Mach. Intell. 40, 464–472, (2017)

How to cite this article: Quek, Y.T., et al.: Deep Q-network implementation for simulated autonomous vehicle control. IET Intell. Transp. Syst. 15, 875–885 (2021). https://doi.org/10.1049/itr2.12067