2D Simulation of a Self-driving Vehicle through Deep Reinforcement Learning

Ghani Haider
Department of Computer Science
Habib University
Karachi, Pakistan

Sarrah
Department of Computer Science
Habib University
Karachi, Pakistan

***Abstract*—A self-driving vehicle, also known as an autonomous vehicle, is one that can sense its surroundings and operate without the need for human intervention. The aim of the paper is to simulate a 2D environment using Deep Reinforcement Learning to teach a vehicle to avoid obstacles and drive independently. With no prior information about the route, the vehicle learns to navigate through a system of rewards and penalties. The results are shown in a live simulation using Python where the unintelligent vehicle gradually finds its way around a path. To test a balanced complexity, the path will not only be straight, but it will also contain curves and sharp edges. The resulting simulation shows boundary detection and depicts a smart, self-driving vehicle.**

***Index Terms*—reinforcement learning, deep Q-network, neural networks, self-driving vehicle**

I. Introduction

Self-driving vehicle, also known as an autonomous vehicle, is a vehicle that is capable of sensing its environment and operating without human involvement. These vehicles combine a variety of sensors to perceive their surroundings. They have then advanced control systems which interpret sensory information to identify appropriate navigation paths, as well as obstacles in their environment.

A widespread approach of AI for self-driving vehicles is the Supervised Learning approach, in which the main limitation is that we need to consider that the vehicle is driving in an open context environment, and therefore we need to train the model with all possible scenes and scenarios in the real world by collecting more and more data and validating system operations based on the collected data.

In order to solve this issue, we can use Reinforcement Learning (RL). In Reinforcement Learning, an agent gathers the environmental information and switches from one state to the next state, based on a defined policy to maximize the rewards. A self-driving vehicle is very similar to an agent in a reinforcement learning approach. Therefore, the objective of this paper is to simulate a 2D environment in which we train a vehicle to autonomously drive in an established environment through Reinforcement Learning.

II. Background and Related Work

Reinforcement Learning is an unsupervised machine learning technique that uses an agent to learn in an interactive environment through trial and error. Given the knowledge of the agent's current state in the environment at a time, the agent takes actions that are judged and then rewarded or punished.

In the case of the agent being the self-driven vehicle, it can take three actions, namely acceleration, deceleration, and steering. We define a policy and a reward function for the actions executed by the self-driving vehicle to move around in the environment.

1) Reward Function: A reward function basically provides feedback when the agent performs an action that can be both positive and negative. For the reward function in the case of self-driving vehicles, we can look at various parameters such as the best way of navigation to get to the final state faster, completing the right manoeuvre etc. For negative rewards, a penalty can be assigned when the vehicle departs from the lane. The rewards can be both short-term and long-term. The self-driving vehicle also needs to realize which new states receive maximum reward and take actions accordingly. Here, the learning is based on rewards instead of labels, where the goal is to maximize reward, accumulated over time.

2) Policy: Policy is the mapping of states of the environment to the actions that should be performed when in those states. For selecting an optimised policy in order to take the next action, one solution is to mimic the vehicles manually driven and the associated distribution as a reference and try to reproduce this behaviour through the agent.

3) Value Function: Another component of Reinforcement Learning involves the value function, which takes the current state as an input and returns the expected cumulative reward that could be collected over iterations.

Since reinforcement learning works by trial and error in which reward is accumulated over iterations, the primary goal is to get the maximum possible value. The higher the value, the better the system performs. Hence, the value function is designed such that a state gives the best possible value over time.

In value iteration, the value of a state is not determined by its own reward, but is a sum of its current and future possible rewards from its next states.

$$G_t = R_{t+1} + R_{t+2} + R_{t+3}...$$

$$G_t = \sum_{k=0}^{T} R_{t+k+1}$$

Consequently, the immediate state rewards have a higher priority because of its predictability over the rewards received in later states. This difference in priorities is where a discounting factor is applied to account for the uncertainties in later rewards.

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1},$$
$$where\ \gamma \in [0, 1)$$

The discounting factor in the above equation is denoted by γ whose value lies between 0 and 1. A greater value of γ means more influence of future rewards and a smaller value of γ means more influence on short term rewards.

A. Q-learning

Q-learning is a reinforcement learning algorithm that is model-free. It means that the agent in Q-learning works by sorting good actions from bad actions without understanding the world it is present in. It simply works by estimation of values of state-action pairs. Instead of having a value for each state, the algorithm now has a value for each state-action pair. After the values have been learned, it follows the rule of obtaining the maximum value. The optimal action from any state is the one with the highest Q-value.

Since Q-learning maps values to state-action pairs, it requires a huge table called Q-table to store all the data. Iterating over all values gets extensive. This makes the algorithm's space-time performance inefficient. The algorithm becomes exhaustive for a big environment as the set of state-action pairs grows and the overall system becomes costly. In order to resolve this issue, deep Q-learning is proposed which combines deep neural network and Q-learning.

The deep Q-learning algorithm has been used before to implement complex problems. One such problem is "An adaptive deep Q-learning strategy for handwritten digit recognition" [1] in which deep Q-learning is used to recognise digits that are handwritten. Another famous application is to play Atari games [2]. The algorithm proves to be superior to previous approaches and human experts.

III. Implementation

The Deep RL algorithm implemented in this paper is the Deep Q-Learning/Deep Q-Network (DQN) [2] with greedy action selection policy. DQN is just a variation of Q-Learning where it makes a neural network act like the Q-table in Q-Learning, thus avoiding creating an unrealistic huge Q-table containing Q-values for every state and action. The Q-values (expected reward) obtained by taking a specific action during a specific state is calculated using the formulae

$$Q(s, a) = r(s, a) + \gamma(maxQ(s', A)$$

where s is the current and s' is the next future state, a is the particular action and A is the action space, $Q(s, a)$ is the Q-value given by taking the action a during the state s, $r(s, a)$ is the reward given by taking the action a during the state s, $maxQ(s', A)$ is the maximum Q-value given by taking any action in the action space A during the state s', and γ is the discount rate that will discount the future Q-values.

So the Q-value given the state s and the a action $Q(s, a)$ is the sum of the rewards given the state s and the a action $r(s, a)$ and the maximum Q-value given any action in the action space and the next state s' $maxQ(s', A)$ multiplied by the discount rate γ. Therefore, the objective is to always choose the action with the highest Q-value to maximize the rewards.

A. DQN Architecture

Instead of parameterizing Q-table using neural networks where it maps state-action pairs to their Q-values, the DQN uses an architecture in which the input to the neural network is only the state representation and there is a separate output unit for each possible action. The output values correspond to the predicted Q-values of the individual actions for the input state. This gives the advantage of calculating Q-values for all possible actions in a given state with only a single forward pass through the network [2].

For implementation, a CNN architecture is utilized to represent the Deep Q-Network. Figure 1 represents the brief structure of the DQN model. It takes 3 consecutive top views of the current state ($96 \times 96 \times 3$ image) of the environment, which in our case is the car-racing environment by OpenAI Gym [3] as the input. The first hidden layer convolves 6 8×8 filters with stride 4 with the input image and applies a reLU-nonlinear function. A max pooling layer of 2×2 pool size is then applied to reduce the spatial size of the convolved features. It is then followed by a second hidden layer that convolves 12 4×4 filters with stride 1, a reLU-nonlinear function and max pooling layer of 2×2 pool size. The final hidden layer is a fully-connected layer with 216 neurons. The output layer is a fully-connected linear layer that represents Q-value of the 12 actions. Each actions consist of one of the 3 states of the steering wheel (left, straight, right), one of the 2 states of the speed (full speed, no speed), and one of the 2 states of the break (applying 20% break, no break).

B. Learning Procedure

In order to approximate Q-values for all the different states and action, the learning process begins with the initialization of main Q and target Q' networks for the agent. If the same network is used to calculate the predicted and the target value, there could be a lot of divergence between these two. Therefore, a separate network is used to estimate the target. This target network has the same architecture as the main network but with frozen parameters and after specific iterations c (a hyper-parameter), the parameters from the main network are copied to the target network. This leads to more stable training and helps the algorithm to learn more effectively, because it keeps the target function fixed for a while. For the implementation of this paper, the main network weights replace the target network weights every 5 steps.

Instead of running Q-learning on state-action pairs as they occur during simulation of the environment, experience replay

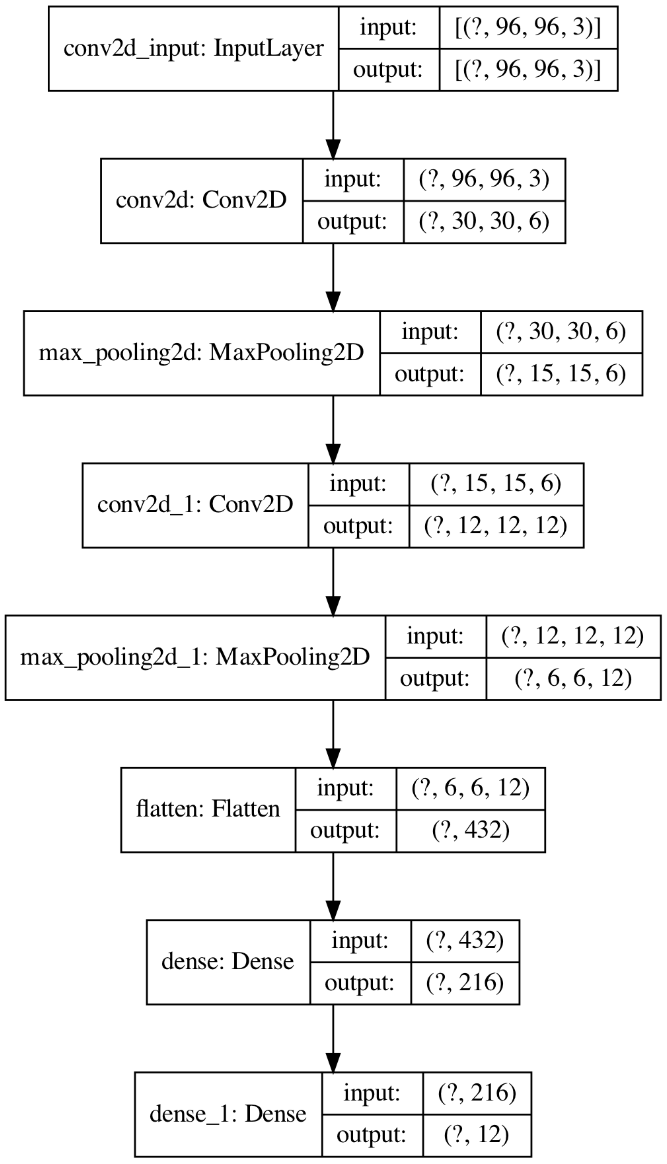

Fig. 1. The DQN Structure

memory is used to store the data discovered for (state, action, reward, next state) in a large table. Experience replay allows updation of the model's parameters using saved and stored information from previously taken actions. In this paper, experience replay is used to train on small batches once every 64 steps rather than every single step to help speed up the training process.

The ϵ-greedy exploration strategy is used to choose an action, where the agent chooses a random action with probability ϵ and exploits the best known action with probability $1 - \epsilon$. After choosing an action, the agent performs the action and updates the main and target networks according to the given equation above. Deep Q-Learning agent uses experience replay to learn about the environment and update the main and target networks.

To summarize, the main network samples and trains on a batch of past experiences every 64 steps. The main network weights are then copied to the target network weights every 5 steps. Figure 2 represents the pseudocode for the discussed solution.

```
Initialize network $Q$
Initialize target network $\hat{Q}$
Initialize experience replay memory $D$
Initialize the $Agent$ to interact with the Environment
while $not\ converged$ do
    /* Sample phase
    $\epsilon \leftarrow$ setting new epsilon with $\epsilon$-decay
    Choose an action $a$ from state $s$ using policy $\epsilon$-greedy($Q$)
    $Agent$ takes action $a$, observe reward $r$, and next state $s'$
    Store transition $(s, a, r, s', done)$ in the experience replay memory $D$
    if $enough\ experiences\ in\ D$ then
        /* Learn phase
        Sample a random $minibatch$ of $N$ transitions from $D$
        for $every\ transition$ $(s_i, a_i, r_i, s'_i, done_i)$ $in$ $minibatch$ do
            if $done_i$ then
                $y_i = r_i$
            else
                $y_i = r_i + \gamma \max_{a' \in \mathcal{A}} \hat{Q}(s'_i, a')$
            end
        end
        Calculate the loss $\mathcal{L}$ =1/N $\sum_{i=0}^{N-1}(Q(s_i, a_i) - y_i)^2$
        Update $Q$ using the SGD algorithm by minimizing the loss $\mathcal{L}$
        Every $C$ steps, copy weights from $Q$ to $\hat{Q}$
    end
end
```

Fig. 2. Pseudocode for DQN

IV. Experiments and Results

We ran our experiment on Car-Racing environment by OpenAI Gym [3]. The agent was trained using the environment for 500 episodes where each episode represented a complete simulation of 1000 frames or until termination criteria was met based on negative rewards. Due to lack of computational power, we also used a frame skipping technique where the agent selects actions after third frames to allow playing 3 times more game without significantly increasing the runtime. Table I shows the hyper-parameter values for the agent.

TABLE I
DQN-Agent Parameters

Frame stack num	3
Memory size	5000
Discount rate γ	0.95
Exploration rate ϵ	1.0
ϵ_{min}	0.1
ϵ_{decay}	0.9999
Learning rate	0.001

Since our metric of evaluation is the total reward the agent collects in an episode or over a number of episodes, we are computing the total reward during the training phase. Figure 3 indicate how the total reward evolved during the training on the environment. We can observe that the graph is very noisy giving the impression that the learning algorithm is not progressing steadily. However, it can be observed from the trend line that the total reward increases over episodes and converges when it is reaching episode 500. The noise in the graph can be attributed to the fact that small changes to the weights of a policy can lead to large changes in the distribution of states the policy visits. Figure 4 shows the performance of the agent with different training episodes. We can clearly see that the model's performance increases with the increase in training episodes.

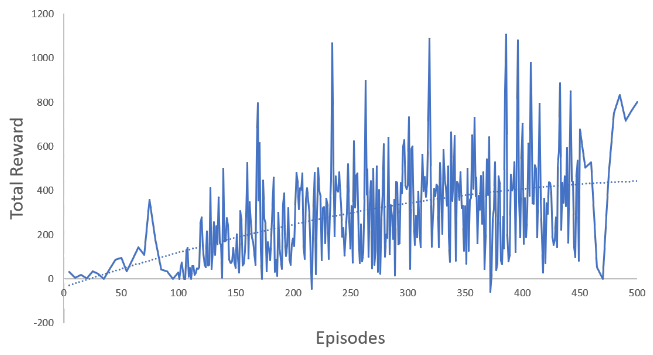

Fig. 3. Total reward against training episode

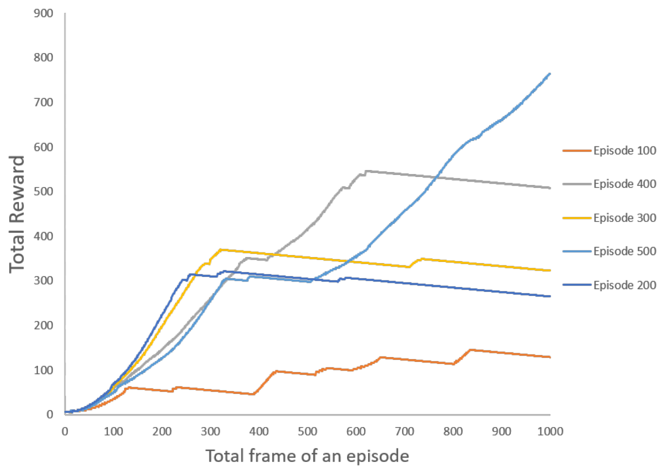

Fig. 4. Agent's simulation on different episodes

V. Conclusion

The paper implemented a reinforcement learning algorithm called Deep Q-Learning Q-Network (DQN) for simulating a self driving vehicle using OpenAI Gym car racing environment. DQN is a variant of Q-learning which combined experience replay memory and stochastic mini-batch update for training the neural network for this task. We were able to successfully train the agent to learn the optimal policies within a few episodes and to visualize the simulation with the trained agent.

References

[1] J. Qiao, G. Wang, W. Li, and M. Chen, "An adaptive deep q-learning strategy for handwritten digit recognition," *Neural Networks*, vol. 107, pp. 61–71, 2018, special issue on deep reinforcement learning. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0893608018300492

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, "Playing atari with deep reinforcement learning," *CoRR*, vol. abs/1312.5602, 2013. [Online]. Available: http://arxiv.org/abs/1312.5602

[3] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.