Simulation-based Self-driving Car using Reinforcement Learning

1st Amit Raj Shakya
Department of Computer and Electronics Engineering
Kantipur Engineering College (TU)
Lalitpur, Nepal
a7shky@gmail.com

2nd Kushal Sagar Shrestha
Department of Computer and Electronics Engineering
Kantipur Engineering College (TU)
Lalitpur, Nepal
scrucial98@gmail.com

3rd Manish Shrestha
Department of Computer and Electronics Engineering
Kantipur Engineering College (TU)
Lalitpur, Nepal
shresmanish1999@gmail.com

4th Nikesh Subedi
Department of Computer and Electronics Engineering
Kantipur Engineering College (TU)
Lalitpur, Nepal
subedi.nikesh@hotmail.com

***Abstract*—Reinforcement learning is an essential machine learning paradigm that is used to develop an AI through interaction with the environment and self-improvement by learning from their mistakes and perceiving the rewards obtained while doing appropriate actions. Its implementation in autonomous driving has not yet been widely applied. But it should be noted that it is difficult to implement autonomous driving as a supervised learning problem. As such, our objective here is to simulate vehicle driving in a virtual environment using Reinforcement Learning to accomplish one of the first steps in self-driving car problems i.e., navigating through a predefined track. The simulated vehicle is only given distances to the walls of the track as input and based on the policy it develops, has to navigate the track without collision. We used a variant of Q-Learning (a Reinforcement Learning algorithm) called Deep Q-Network (DQN) to develop the aforementioned policy. DQN uses neural networks to approximate the policy function. The environment, as well as the vehicle simulation, is done in Unreal Engine 4.**

***Index Terms*—Reinforcement Learning, Self-driving, Simulation**

I. Introduction

Reinforcement learning is a machine learning paradigm where an agent learns behavior through trial-and-error interactions with an environment agent taking an action within an environment (i.e., machines can learn similar to humans where we learn from our action and its consequence). The environment returns two types of information to the agent — reward and state. The quantitative feedback on the action that the agent took can be positive or negative termed as a reward. The agent's goal is to obtain the maximum reward possible in an episode. An episode is the duration between the first state and the last or terminal state within the environment.

Our objective here was to create a simulated self-driving car to navigate through a track without colliding with the boundary of the track (walls). To achieve this, we used an RL algorithm called Deep Q-Network(DQN). For the simulation of the environment, we used Unreal Engine 4, a game development engine developed by Epic Games. Unreal Engine gives a visual representation of our agent's action with the environment. The only input the car receives is the values of distances to the walls of the track as measured by the sensors present in the car. The DQN comprises neural networks with multiple hidden layers as action-value functions and an experience replay mechanism. The goal of the car (agent) is to interact with the simulation (environment) in a way that maximizes future rewards.

II. Related Work

One of the best implementations of reinforcement learning is The DQN Atari project which involved training an agent using reinforcement learning to play 7 different games in the Atari 2600 console in an arcade learning environment using only raw pixels as input which was able to outperform all the previous approaches in 6 of the 7 games and was also able to surpass human experts in 3 of the 7 games. In the project, the network was trained with a variant of the Q-learning Algorithm called Deep Q-Network using Stochastic Gradient Descent for updating the weights. Experience Replay was implemented to sample data from previous transitions resulting in smooth training distribution over several past behaviors [1]

Deep Deterministic Policy Gradient (DDPG) algorithm having the capacity to handle complex state, action pair in continuous simulations has been used in a simulation of a fully autonomous vehicle in environments where action spaces were continuous. In the DDPG Reinforcement Learning for Autonomous Driving Car project, The Open Racing Car Simulation (TORCS) was used which would continuously provide sensor information to the algorithm and based on these inputs, a reward function inside TORCS would

encourage the agent to run fast without hitting the other cars while driving in the middle of the road. To fit in the TORCS environment, a network architecture was designed for both actor and critic used in the DDPG algorithm [2].

Another example is the Gran Turismo Sport Deep Reinforcement Learning project that makes use of the PlayStation 4 game Gran Turismo Sport as the environment allowing usage of various tracks and cars. It was not only able to beat the built-in NPCs (Non-Player Characters) but also was able to obtain the fastest time in the dataset of best lap time records. They were able to create an autonomous agent that was able to match or outperform human experts in time trials without relying on any forms of path planning, while also leading to trajectories that are qualitatively similar to those chosen by the best human players [3].

III. Reinforcement Learning

Reinforcement Learning (RL) deals with agents that observe and act upon their environment to learn the optimal action to take in any given state. As such, in our case, the agent interacts with the environment in the simulation performing a sequence of actions, observes state, and receives feedback in the form of rewards. Depending on whether the agent received positive or negative feedback, the agent is motivated to learn. This model of learning in essence represents a Markov Decision Process (MDP). MDP is fundamental to RL as it provides the basis for agents to make decisions based on environment, state, action, and reward. The goal of MDP is to maximize some performance criterium [4]. Regarding RL, that performance criterium is the reward. Thus, we can apply standard RL methods for MDPs using a complete sequence of state s_t (state at time t) [1].

We take into account not only the current reward obtained but immediate future rewards which need to be discounted by a factor of γ per timestep. The discounted return/reward is defined as,

$$R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'} \tag{1}$$

We define an action-value function $Q(s, a)$ that represents the discounted future reward obtained when performing action a in a state s. The goal is to learn a policy π that maps sequences to actions that maximize the discounted future reward.

$$\pi = argmax_a Q(s, a) \tag{2}$$

The optimal action-value function $Q^*(s', a')$ for next state s' (state at next timestep), and all possible actions a' obeys the *Bellman equation* [1],

$$Q^*(s, a) = r + \gamma max_{a'} Q^*(s', a') \tag{3}$$

We use equation 3 to estimate the action-value function in an iterative way. In RL, it is essential to find balance between *exploration* (selecting sub-optimal action) and *exploitation* (seleting the best known action i.e., policy action) [5]. This can be achieved using ϵ-greedy strategy i.e., selecting a policy action with probability $1 - \epsilon$ and selecting a random action with probablity ϵ.

A. Deep Reinforcement Learning

For problems with continuous state-space, the action-value function can be approximated using neural networks. Along with this, we can also utilize the *experience replay* mechanism where we store the agent's *experiences* at each timestep. An experience is a tuple of state, action, next state, and reward i.e., for each timestep t, experience is given as $e_t = (s_t, a_t, s_{t+1}, r_t)$. These experiences are stored in a memory buffer called *replay memory*. This is later used to sample inputs for the neural network [1]. This approach is termed as *Deep Q-Learning* and the algorithm is presented in Algorithm 1. We only store the most recent N experiences and uniformly sample a random minibatch of experiences. The neural network inputs states and outputs Q-values for each action. In RL, the training is unstable because both the input and the target change constantly during the process [6]. Also, the agent depends on the action-value function to sample actions but this function changes frequently as the agent learns about the environment. So we implement two action-value functions – policy action-value function (Q) and target action-value function ($\hat{Q}$) [7].

IV. Experiments

We created a simple 3D world inside *Unreal Engine 4*, which is a game engine developed by Epic Games. This world is composed of an agent (Car), and a track. We added 7 sensors to the agent. These sensors measure distances to the walls of the track. The measurements were normalized in the range of 0 to 1. These measurements represent a state. At any given time, the agent is capable of performing one of three actions — move forward, turn left, turn right. The reward obtained is determined by the agent's distance from the wall as well as whether the agent has collided with the wall or not. The agent receives a reward of -200 if it collides with the wall. If it is critically close to the wall i.e., any one of its sensors outputs a value less than 0.1, the agent receives a reward of -20. For the agent to receive a positive reward, we have added reward gates at various positions of the track. Essentially, the agent receives an additional reward of $+50$ every time it passes one of these reward gates. We consider the episode to end when the car collides with the wall. These reward values were chosen experimentally, we essentially wanted to give a higher negative reward if the agent collided with the walls and a relatively lower negative reward if it got critically close to the walls.

Algorithm 1: Deep Q-Learning with experience replay [7]

Initialize replay memory D to capacity N
Initialize action-value funtion Q with random weights θ
Initialize target action-value function $\hat{Q}$ with with weights $\theta^- = \theta$
for $episode = 1,\ M$ **do**
  Initialize state s_1
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = argmax_a Q(s_t, a; \theta)$
    Execute action a_t and observe reward r_t and next state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in D
    Sample random minibatch of transitions (s_t, a_t, r_t, s_{t+1}) from D
    Set $y_j = \begin{cases} r_j & \text{if episode ends at } j+1 \\ r_j + \gamma max_{a'} \hat{Q}(s_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
    Perform gradient descent step on $(y_j - Q(s_j, a_j, \theta))^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  end
end

We used a neural network with three hidden layers for both the action-value functions Q and $\hat{Q}$. The input layer and the hidden layers use $ReLU$ activation function and the output layer is a $Linear$ function. We used $Mean\ Squared\ Error\ (MSE)$ loss function and $Adam$ optimizer to update the weights using back-propagation. For the $\epsilon-$greedy strategy, we decreased the value of ϵ exponentially as follows,

$$\epsilon = \epsilon_{min} + (\epsilon_{max} - \epsilon_{min}) * e^{-\eta * t} \tag{4}$$

A complete list of parameters and hyperparameters used is given in Table I. They were tuned experimentally through trial and error. The number of hidden layers in the neural network turned out to drastically improve the policy. We initially used a neural network with just one hidden layer but it turned out to be insufficient to develop a proper policy to effectively navigate through the track even after long hours of training. The learning rate turned out to be another crucial hyperparameter to tune. Training with a very small learning rate increased the time required for convergence, while a greater learning rate made the training unstable. The other hyperparameters like the batch size didn't affect the learning all that much.

TABLE I: Parameters and Hyperparameters of DQN

Parameters and Hyperparameters in DQN	
State-space size	7
Action-space size	3
Hidden layer 1 size	16
Hidden layer 2 size	16
Hidden layer 3 size	8
Discount factor (γ)	0.95
Learning rate	1×10^{-3}
Batch size	128
Capacity (N)	131072
Update Step size (C)	36000
ϵ_{max}	0.99
ϵ_{min}	0.01
Decay rate (η)	1×10^{-4}

The flow of our system is as shown in Figure 1. The simulation is reset every time the agent collides with the walls of the track. The agent performs actions based on the policy action-value – which takes the current state as input – and as a result, gets a reward and transitions to a new state. The policy action-value function is trained by sampling a random minibatch from the replay memory (as described in the Algorithm 1) and the target action-value function is updated every 'C' timesteps.

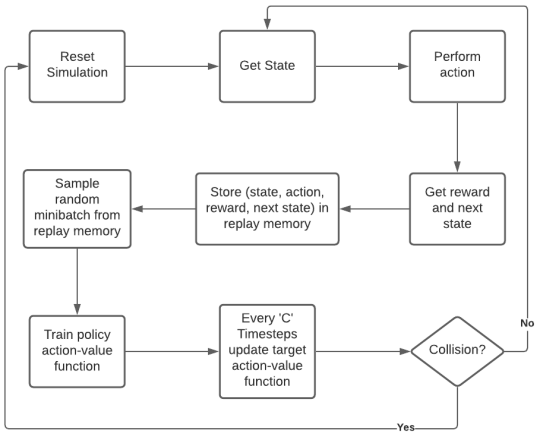
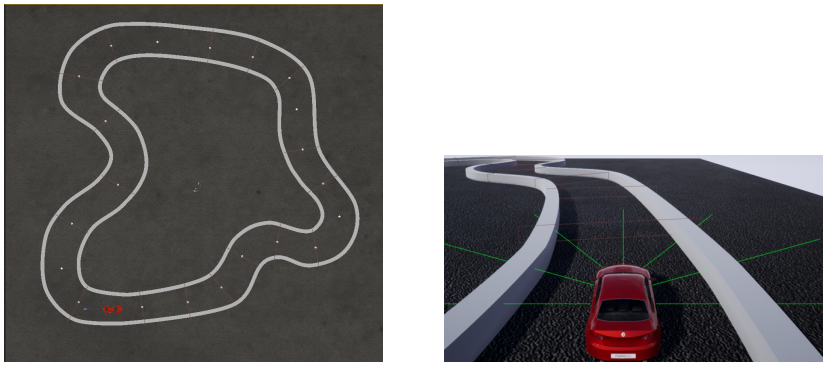


Fig. 1: System Flow Diagram

V. Results

After training the agent for about 3000 episodes, it was able to successfully navigate the track without colliding into the walls. We used a standard computer with *Intel i5-9300H* CPU with a clock speed of 2.40GHz, 8GB memory and a *GeForce GTX 1050* GPU to run the simulation. This hardware configuration was limiting and we were not able to train the



(a) Top view of the environment (b) Car showing sensors

Fig. 2: **Left:** Shows the track, car and positions of the reward gates (the white dots). **Right:** Green lines represent sensors of the car.

agent beyond 3000 episodes. Thus, the agent navigated the track in a rather unconventional motion, using the combination of 'turn left' and 'turn right' actions more often as opposed to the 'move forward' action. This problem can be solved with sufficient training.

We cannot simply use the MSE loss function as the evaluation metric for RL problems. This is because the loss function tends to be very unstable during the training, due to the fact that the target action-value function has to be updated every certain timestep. As such, our evaluation metric is the total reward the agent obtains in an episode averaged over a certain number of episodes. This metric (the average total reward metric), tends to be very noisy because small changes to the weights of the policy action-value function can cause large changes within the distribution of states the policy visits [1]. The graph shown in Figure 3 shows total reward obtained by the agent averaged over every 20 episodes. We can see a gradual increase in the average reward obtained from the 1000^{th} episode. The rewards are dominant on the negative side because of the reward function we designed (there is a higher magnitude of negative rewards).

We evaluated the agent in the same track that it was trained in. During the evaluation, the agent could finish a lap around the track with a total reward of 380.

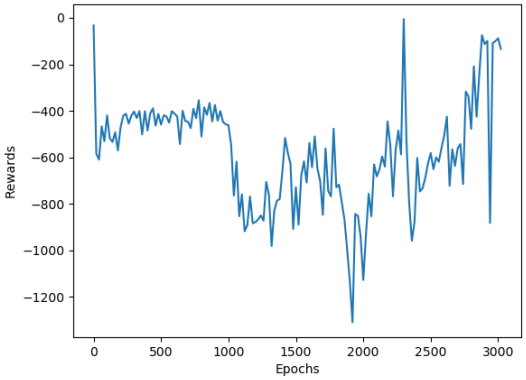


Fig. 3: Graph showing average reward per 20 episodes (during training)

VI. Conclusion

In this paper, we present an approach to simulate training of a car to drive without colliding into the walls in the environment. The results show that with enough training the car was at least able to navigate though the track without colliding with the wall. However, due to hardware limitations, we weren't able to train the agent for long hours and thus, the agent wasn't able to develop a proper policy to navigate the track in a smooth manner (i.e., using 'move forward' action more often as opposed to using a combination of 'turn left' and 'turn right' actions). Still, this shows us that with enough training it is possible to have it fully go through the track in a much smoother manner. We focused on training the model with different parameters and hyperparameters. This allowed us to fine-tune and see which ones would produce better results faster.

References

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[2] S. Wang, D. Jia, and X. Weng, "Deep reinforcement learning for autonomous driving," *arXiv preprint arXiv:1811.11329*, 2018.

[3] F. Fuchs, Y. Song, E. Kaufmann, D. Scaramuzza, and P. Duerr, "Super-human performance in gran turismo sport using deep reinforcement learning," *arXiv preprint arXiv:2008.07971*, 2020.

[4] M. van Otterlo and M. Wiering, "Reinforcement learning and markov decision processes," in *Reinforcement Learning*. Springer, 2012, pp. 3–42.

[5] M. Coggan, "Exploration and exploitation in reinforcement learning," *Research supervised by Prof. Doina Precup, CRA-W DMP Project at McGill University*, 2004.

[6] J. Hui, "Rl - dqn deep q-network," Mar 2019. [Online]. Available: https://jonathan-hui.medium.com/rl-dqn-deep-q-network-e207751f7ae4

[7] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.