OELP Endterm Report
Control Designs for Autonomous Vehicles based on Reinforcement Learning.

Ambati Thrinay Kumar Reddy
Btech Electrical Department
IIT Palakkad
121901003@smail.iitpkd.ac.in

Nalamalapu Venkata Sumanth Reddy
Btech Electrical Department
IIT Palakkad
121901028@smail.iitpkd.ac.in

Mentor: Dr. Shaikshavali Chitraganti
Assistant Professor, Electrical Department
IIT Palakkad
shaik@iitpkd.ac.in

***Abstract*—Developing control designs for autonomous vehicles based on reinforcement learning. Control designs include discrete and continuous action space based designs with emphasis on learning from interaction with an environment in order to achieve goals instead of handcrafted control strategies. Control Designs have to be adaptable to the environment. Training and testing simulations are to be performed in python.**

***Index Terms*—Reinforcement learning, Q-learning, Deep Q-learning**

I. Introduction

With the growth of driver safety features such as Intelligent Driver Assist Systems and Advanced Driver Assist Systems, which strive to assist the driver travel safely, technologies have been increasing enormously. Despite this, countless accidents have occurred in recent decades as a result of improper usage and the stress that drivers experience during lengthy rides. To make driving safer, active safety systems like as Anti-lock Braking Systems, Electronic Stability Programs, Vehicle Stability Programs, and Traction Control have been developed. In recent improvements, active safety has become a top concern, and safety requirements have significantly increased. The addition of sensors like as cameras, LiDAR, and Radars tend to increase the vehicle's safety by recognising things further away and taking the appropriate measures to avoid the obstacles and stay inside the allocated lanes. The capacity of the system to drive autonomously, regulating the steering, acceleration, braking, recognising and avoiding obstacles, would be a significant advancement in the safety technology if proper control Design can take advantage of the add-ons.

For operating these vehicles, traditional PID control-based systems are utilised to regulate the steering angle and speed at each time step with the least amount of cross-track error. Many concerns in control design for autonomous vehicles are not well handled in classical control theory, such as totally changing the course to which it is steered or experiencing new challenges in the place where it is deployed, causing the vehicle to divert from the path and collide. Because this is a decision-making problem that can be solved via experience, a reinforcement learning-based control system can be implemented [1]. They may be trained to comprehend the general principles of driving by activating these control systems in a virtual environment using Python and real-world physics. By granting it access to and information on numerous driving factors such as vehicle size, speed, wheel distance, steering angle, cross-track error, and obstacle position. The control system is programmed to follow a path or travel through checkpoints at a set pace. Designs developed in these manner also can adapt which was not possible before.

II. Reinforcement Learning

Reinforcement learning (RL) is a set of solutions aiming to solve the generic problem of finding optimal solutions to a given problem. The defining features of reinforcement learning from other machine learning methods is that it is active rather than passive and decisions are often sequential i.e future decisions can depend on the earlier ones. These algorithms are built on the fact that humans and animals learn by interacting with our environment as a result training of these methods is based on rewarding desired behaviors and/or punishing undesired ones.

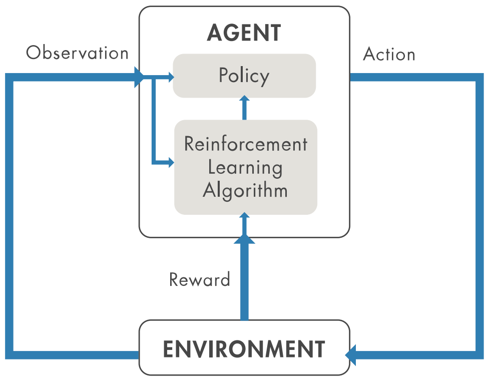

Fig. 1: Agent Environment interaction of Reinforcement Learning

Based on the algorithm, a RL agent learns from consequences of its actions and also by new choices, which is essentially trial and error learning. Interaction loop of reinforcement learning is shown in figure 1 where at each time step t = 1,2,3,.. the agent is in a certain state $s_t \in S$ where S denotes all non-terminal states the agent can end up in, and takes one of the possible available actions $a_t \in A(s)$ where $A(s)$ denotes all possible actions in state s. The action changes the environment and the agent ends up in a new state s_{t+1}. Furthermore, it receives a reward R_{t+1} from the environment that serves as a feed-back about how good it was to take action a_t in state s_t. Two important challenges in RL are the credit assignment problem and exploration vs exploitation trade-off [2]. Exploitation refers to taking actions based only on the present experience because the actions that are known will achieve a high reward. This is not helpful for discovering better rewards in the future which may need few sacrifices at present. Exploration refers to taking actions specifically in random fashion to explore environment better. Credit assignment comes into picture if the actions have to be picked from the sequence of actions which lead to the reward as the actions are helpful for perfecting the algorithms.

A. Standard model of RL

An entity that can explore its surroundings and act on them is called a $agent$ and surroundings the agent is present in called $Environment$ which is assumed to be stochastic in general. Agent state s is defined as a set S of agent observable states which is returned by the environment in the name of observation. An action is defined as a collection of feasible actions in a given state S, i.e moves taken by an agent within the environment. Policy,Reward,Value function and Model represent the standard model of reinforcement learning.

- $Policy$: policy π defines the agent's behaviour and map from agent state to action and tells about the desirability of action given a agent state. Core goal of the RL is to find the optimal policy for given problem which can be Deterministic $\pi(s)$ or Stochastic $\pi(a|s)$.
- $Reward$: Information delivered to the agent on how good and bad is an action and it defines the goal of the problem and here the agent objective is to maximize the total received reward. In fact Reinforcement learning itself is based on the reward hypothesis [3]:
  Any goal can be formalized as the outcome of maximizing a cumulative reward Positive reward is associated with successful decision and negative reward signal is associated with bad decision.
- $Value function$: It is a prediction of the total future rewards, it is used to evaluate the desirability of states and select between actions accordingly [3]. Agents often approximate value functions as they can be used to build a action value of state by following a policy which means with an accurate value function agent can behave optimally for reaching the goal
- $Model$: The model of the environment allow predictions to be made about the behavior of the environment. However, Model is often a optional element of reinforcement learning. Methods that use models and planning are called model-based methods. On the other side is model free methods where the agent does not have a model for the environment. Model-free methods are explicitly trial and error learners [3]

B. Markov Decision Process (MDP)

The $Markov\ Assumption$ assumes an independence of past and future states, which means that the state and the behavior of the environment at time step t are not influenced by the past agent-environment interactions $a_1, a_2, ..., a_{t-1}$. This assumption of environment to be Markov is important as now the agent need not store every state till the present state. If the RL can fulfill the $Markov\ Assumption$, it can be formulated as a set of five-tuple $Markov\ decision\ Process$ $(S, A, P^a_{s,s'}, R^a_{s,s'}, \gamma)$.

- $A = A(s)$ denotes the set of available actions in state s.
- Transition probabilities $P^a_{s,s'}$: $(S \times A \times S)$ denotes the probability of the transition from s to $s^{'}$ when taking a in state s at time step t.

$$P^a_{s,s'} = P(s, a, s^{'}) \approx p\left(S_{t+1} = s^{'} | S_t = s, A_t = a\right) \quad (1)$$

- $R^a_{s,s'}$: $(S \times A \times S)$ denotes the immediate reward the agent receives after the transition from s to $s^{'}$.

$$R^a_{s,s'} = R(s, a, s^{'}) \approx E\left[R_{t+1} | S_t = s, S_{t+1} = s^{'}, A_t = a\right] \quad (2)$$

  The agent can represent the reward value as an expected value since it can deliver various rewards even if the same action is performed in the same state depending on the environment.
- Discount factor $\gamma \in [0, 1]$ for computing the $discounted\ expected$ return.

The Markov Decision Process(MDP) is finite if the set of states S and actions A is finite.

C. Discounted Expected Reward

For training an effective agent, its goal is to maximize the reward in the long run instead of just caring about the immediate return. For example, consider an agent playing chess if the agent cares about just the immediate reward of taking opponent's pieces then there is no incentive for checkmate which gives high reward if the goal of the agent is to win the game as fast a possible. Instead of maximizing the immediate reward the agent can maximize the $discounted\ expected\ return$ where the $return$ denotes the sum of all expected rewards in a given episode. Return is discounted to account for the environment having infinite episodic tasks. In short $discounted\ expected\ return$ (3) is the cumulative sum of possible future rewards. The discount factor $\gamma \in [0, 1]$ rates the future rewards and defines how far in the future rewards are considered. If $\gamma = 1$ all rewards of the future are considered with same weight and if $\gamma = 0$ just the immediate reward is taken into account.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \quad (3)$$

Above formulated return can also be differentiated between $episodic$ and $continuous\ tasks$

- Episodic : The training procedure can be divided into episodes. When the agent reaches a terminal state, the episode will end, the scene will reset and the agent will restart in the next episode. The terminal state can be reached in T finite time steps.
- Continuous : The problem cannot be formulated in episodes and is a continuous ongoing problem so that $T = \infty$.

D. Policy and Value Functions

A policy π tells the agent how to behave. It models a probability distribution $\pi(a|s)$ over the number of available actions $a \in A(s)$ for each state s. i.e $\pi(A_t|S_t$ is the probability of taking action A_t in state S_t. The agent samples its next action from that probability distribution $\pi(a|s)$. The value function $v_\pi(s)$ (4) is an estimate of how good it is for the agent to be in state s and is expected discounted return of state s, if the agent behaves according to policy π (5).

$$v_\pi(s) = E_\pi\left[G_t|S_t = s\right], for\ all\ s \in S \tag{4}$$

$$v_\pi(s) = E_\pi\left[\sum_{k=0}^{\infty}\gamma^k R_{t+k+1}|S_t = s\right], for\ all\ s \in S \tag{5}$$

The value function v_π can be written in recursive from (6) and is called Bellman Equation for v_π. In the Bellman equation, the value of state s is only dependent on the next possible states $s^{'}$ while each state is weighted by the transition probability $P^a_{s,s'}$. Many RL solutions like Q-learning approximate the optimal Bellman equation by approximating the value of the next states.

$$\begin{aligned} v_\pi(s) &= E_\pi\left[G_t|S_t = s\right] \\ &= E_\pi\left[R_{t+1} + \gamma G_{t+1}|S_t = s\right] \\ &= \sum_a \pi(a|s)\sum_{s'} P^a_{s,s'}\left[R^a_{s,s'} + \gamma E_\pi\left[G_{t+1}|S_{t+1} = s^{'}\right]\right] \\ &= \sum_a \pi(a|s)\sum_{s'} P^a_{s,s'}\left[R^a_{s,s'} + \gamma v_\pi(s^{'})\right], for\ all\ s \in S \end{aligned} \tag{6}$$

The action-value function $q_\pi(s,a)$ is an estimate of how good it is to take action a in state s. $q_\pi(s,a)$ (7) is the expected return of taking action a in state s and there after behaving according to policy π.

$$\begin{aligned} v_\pi(s) &= E_\pi\left[G_t|S_t = s, A_t = a\right] \\ &= E_\pi\left[\sum_{k=0}^{\infty}\gamma^k R_{t+k+1}|S_t = s, A_t = a\right], for\ all\ s \in S \end{aligned} \tag{7}$$

RL aims to find an optimal policy π_*. A policy is better than another policy $\pi \geq \pi^{'}$ if the value function of the new policy is better $v_\pi(s) \geq v_{\pi'}(s)$ for all $s \in S$. if the state-value function is optimal, an optimal policy was used by the agent. It is possible that there are multiple policies, that lead to the same optimal state-value function. v_* (8) denote the optimal state-value function.

$$v_*(s) = \max_\pi v_\pi(s),\ for\ all\ s \in S \tag{8}$$

Furthermore, optimal policies result in the optimal action-value function q_* (9).

$$\begin{aligned} q_*(s,a) &= \underset{\pi}{max}\ q_\pi(s,a),\ for\ all\ s \in S\ and\ a \in A(s) \\ &= E\left[R_{t+1} + \gamma v_*(s^{'})|S_{t+1} = s, A_t = a\right] \end{aligned} \tag{9}$$

The optimal state-value function (8),optimal action-value function (9) can be used to describe $Bellman\ optimality\ equation$ (10) rewritten from (5) (7).

$$\begin{aligned} v_*(s) &= \max_{a \in A(s)} q_{\pi_*}(s,a) \\ &= \underset{a}{max}\ E\left[R_{t+1} + \gamma v_*(s^{'})|S_{t+1} = s, A_t = a\right] \\ &= \underset{a}{max}\sum_{s'} P^a_{s,s'}\left[R^a_{s,s'} + \gamma v_*(s^{'})\right] \\ &= \underset{a}{max}\sum_{s'} P^a_{s,s'}\left[R^a_{s,s'} + \gamma \max_{a'} q_{\pi_*}(s^{'},a^{'})\right] \end{aligned} \tag{10}$$

optimal action-value function can be obtained by solving the Bellman optimality equation (10) which can be used to choose optimal action for any state in S to get maximum possible sum of rewards by just choosing the action with maximum value function for each state. Then the resulting optimal policy π^* can written as (11)

$$\pi_*(a|s) = \begin{cases} 1 & \text{if } a = \underset{a \in A(s)}{argmax}\ q_*(s,a) \\ 0 & \text{otherwise} \end{cases} \tag{11}$$

Action-value Methods are class of RL techniques which determine the action-value function to decide the optimal policy. Monte Carlo method, Temporal Difference Methods and Q-learning are examples of Action-value Methods.

E. Monte Carlo,Temporal Difference Methods

The Monte Carlo Method and the Temporal-Difference Learning Methods are both model-free and have no knowledge of MDP transitions or rewards [4]. They learn directly from experiences, therefore the MC return may be approximated by averaging the returns from many rollouts. Unlike Monte Carlo, which learns from complete episodes through sampling, Temporal-Difference learns from partial episodes by sampling and bootstrapping. However, it is possible to get the best of both Monte Carlo method and TD learning as in the TD(λ) algorithm where λ interpolates between temporal Difference (λ=0) and Monte Carlo (λ =1).

F. Q-learning

Q-learning is a popular off-policy Temporal Difference learning method, i.e. the policy π is not used for updating the state-action values also called Q-values and the state-action function is called Q-table. This algorithm involves determining behaviour when the agent does not know how the world works and can learn how to behave from direct experience with world. Q-learning estimates the optimal Q-values of an MDP, which means that behavior can be learned by taking actions greedily with respect to the learned Q-values. The most common way to choose an action in the current world state(s) is to use greedy policy (12), is the ϵ-greedy policy used in Q-learning. ϵ is fraction between 0 and 1 and weighs the relation between exploitation and exploration.

$$\pi_*(a|s) \leftarrow \begin{cases} 1-\epsilon & \text{if } a = \underset{a \in A(s)}{argmax}\ q_*(s,a) \\ \epsilon & \text{otherwise} \end{cases} \tag{12}$$

Based on the policy (12), the agent take a random action with a probability of ϵ to explore the action space or takes a greedy action with a probability of $(1-\epsilon)$ based on the Q-value from the Q-table. The updated rule for Q-learning (13) shows that Q-value is updated by the Q-value of the last state-action pair (s, a) with respect to the observed outcome state $s^{'}$ and direct reward $R(s, a, s^{'})$. The parameter α between 0 and 1 stands for the learning rate.

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[R(s,a,s^{'}) + \gamma \underset{a^{'}}{max} Q(s^{'},a^{'}) - Q(s,a) \right] \tag{13}$$

Q-value of Q-learning algorithm is the sum of rewards and discounted max Q-value of the observed next state, which implies that we only use the states and rewards we happen to get by interacting with the environment. As long as we keep trying random actions on the same state, we could reach all possible states of next. After multiple times of aggregation, we should finally move close to the true Q-value. In order to have guaranteed convergence greedy policy should anneal linearly over a certain training steps, and fixed at the small fraction thereafter. This setting enables the agent to explore more action-state pairs at the beginning of the training, and reduce the randomization when the agent gains more experience. Another technique used is slowly decreasing the learning rate(α) over time to decrease the effect of random actions or incorrect actions on the Q-values. The Q-learning algorithm is summarized below. The basic idea of Q-learning is to estimate the action-value function by using Bellman Equation as an iterative update. In that case, the value function converges to optimal the value function as the iterations tends to infinity. However, updating the state-action values considering only two state-action pairs without any generalization over the whole sequence is impractical for complex environments on top of the fact that Q-learning is model-free learning. Instead,it is common to use a function approximator like neural networks to estimate the action-value function [5]. The other problem is that traditional reinforcement learning algorithms heavily rely on the quality of hand-crafted feature representations, which limits the application scope for complex systems.Based on recent studies, deep learning models have been found to be effective feature extractors over raw high-dimensional data[6].

Algorithm 1 Q-Learning Algorithm

```
Require: learning rate $\alpha \in (0, 1]$
Ensure: small $\epsilon \geq 0$
 1: Initialize $Q(s, a)$, for all $s \in S$, $a \in A(s)$, arbitrarily
    except that $Q(terminal, .) = 0$
 2: for each episode do
 3:     Initialize agent state,environment
 4:     repeat
 5:         For each step of episode
 6:         Choose an action $a$ in current state $s$ using policy
            derived from $Q(s, )$(eg: $\epsilon$ greedy)
 7:         Take action $a$
 8:         Observe the outcome state $s^{'}$
            and reward $R(s, a, s^{'})$
 9:         update $Q(s, a)$ like (13)
10:         $s \leftarrow s^{'}$
11:     until $s$ is terminal
12: end for
```

III. Deep Reinforcement Learning

Deep Learning is one of the most powerful tools we have today for dealing with unstructured environments; it can learn from vast volumes of data and find patterns. Recent breakthroughs in vision and speech applications were possible by training deep neural networks on large datasets to better approximate any nonlinear complex system. The most successful approaches are trained directly from the raw inputs, using lightweight updates based on stochastic gradient descent. By feeding sufficient data into deep neural networks, it is often possible to learn better function representation than handcrafted functions mapping[7].This success motivates the new approach to the reinforcement learning called Deep Q-Networks [8], published by Google deep-mind group.

The optimal state-action value is defines as the sum of direct reward and the discounted state-action of the next step. In this way the action-value function is estimated separately for each sequence, without generalization which may not be practical if the terminal state is difficult to find. Instead, a non-linear function approximater like a neural network can be utilized. This Idea already exists in the reinforcement learning realm in the form of linear function approximator as action-value function.

$$Q(s,a;w) \approx Q(s,a) \approx \underset{s^{'}}{E}[r + \gamma \underset{a^{'}}{max} Q(s^{'},a^{'})] \tag{14}$$

In order to train the neural networks, huber-loss function is used which is combination of mean square error and absolute error. For small errors mean square error is used i.e errors less than 1.0 resulting in arithmetic mean-unbiased error and for large errors absolute error is used resulting in median-unbiased error. Without any hand-engineered features or domain heuristics, agents generate and acquire their own knowledge directly from raw inputs. Deep learning neural networks are used to do this. Neural networks, on the other hand, aren't the perfect solution for every situation. Neural networks, for example, are data-hungry and difficult to analyze. Regardless, neural networks are one of the most powerful approaches accessible right now, and their performance is generally the greatest.

agents must make constant value assessments in order to choose good actions over bad. A Q-network represents this information by estimating the overall reward that an agent can anticipate after doing a certain action.The fundamental concept was to represent the Q-network using deep neural networks and train this Q-network to predict total reward. The Deep Q-Networks (DQN) approach addresses these instabilities by storing all of the agent's experiences and then randomly sampling and replaying them to offer different and decorrelated training data.

A. *Deep Q Learning*

To develop a cheat sheet for agents, Q-learning can be used, a simple yet effective algorithm. This guides the agent in determining which action to take. Consider a world with 10,000 states and 10,000 actions for each state. This would result in a 100 million block table which is a huge cheat sheet that will become very complex. It's evident that we can't deduce the Q-value of new states from those that have already been studied. This causes two issues, the amount of memory necessary to save and update the table grows as the number of states grows, and the amount of time required to explore each state becomes more in order to generate the necessary Q-table which is impractical. So we use machine learning models like a neural network to estimate these Q-values.

As mentioned earlier in deep Q-learning, we use a neural network to approximate the Q-value function. The state is provided as an input, and the output is the Q-value of all possible actions.

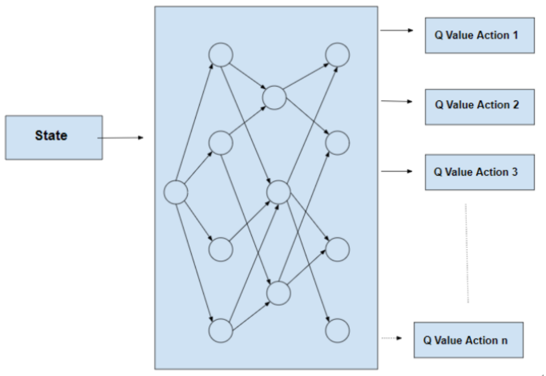

Fig. 2: Q values using DQN

The following are the steps involved in reinforcement learning with deep Q-learning networks (DQNs), the user stores all previous experience in memory, the maximum output of the Q-network determines the next action, and the loss function is mean squared error of the predicted Q-value and the target Q-value Q^-. This is essentially a supervised learning problem. However, because we are working with a reinforcement learning problem, we do not know the target or actual value.

Algorithm 2 Deep Q-Network Algorithm

```
Require: Learning rate $\alpha \in (0, 1]$
Ensure: $\epsilon \geq 0$ for $\epsilon$ greedy policy
 1: Initialize the replay memory to a fixed capacity $N$
 2: Initialize the local neural network with random weight $w$.
 3: for each episode do
 4:     Reset the environment
 5:     repeat
 6:         For each step of episode
 7:         Choose an random action $a$ with probability $\epsilon$
            otherwise select $a = argmaxQ(s, a; w)$
 8:         Execute action $a$
 9:         Observe the outcome state $s^{'}$
            and reward $R(s, a, s^{'})$
10:         Store the transition experience $(s, a, r, s')$ in
            the replay memory
11:         random minibatch of transitions $(s_j, a_j, r_j, s_j^{'})$
    from replay memory
12:
            $$y_j = \begin{cases} r_j, & \text{for terminal } s_j^{'} \\ r_j + \gamma \underset{a^{'}}{max} Q^-(s_j^{'}, a^{'}; w^-), & \\ \quad \text{for non-terminal } s_j^{'} & \end{cases}$$
13:         Perform a gradient descent step on the loss func-
    tion $(y_j - Q(s_j, a_j; w))^2$
14:         Update the weight of target network $Q^- = Q$
            every $N$ steps
15:         $s \leftarrow s^{'}$
16:     until $s$ is terminal
17: end for
```

B. *Target network*

Since the target values are calculated by the same network, there might be significant differences between them. As a result, we may employ two neural networks for learning instead of one. To estimate the goal, we may utilize a different network. The design of this target network is similar to that of the function approximator, but with fixed parameters. The parameters from the prediction network are copied to the target network every N iterations (a hyperparameter). This makes the target function unchanged for some time which makes the training smooth. This type of learning now resembles supervised learning were target parameters to be learned are fixed are fixed for short duration.

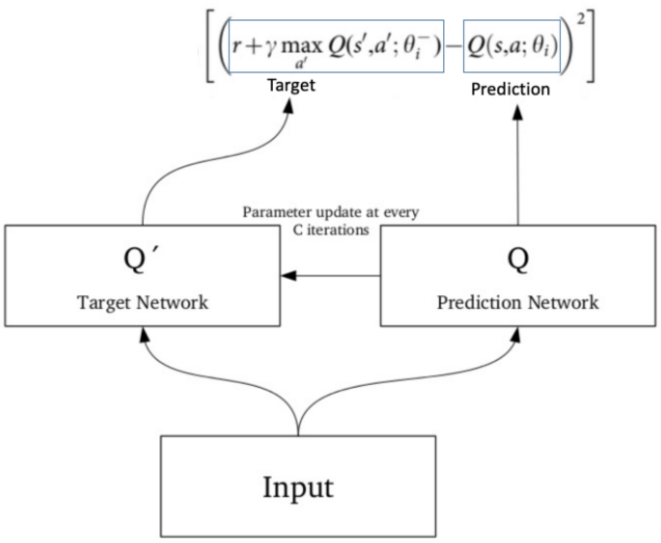
$$\left[\left(\boxed{r+\gamma \max_{a'} Q(s',a';\theta_i^-)}-\boxed{Q(s,a;\theta_i)}\right)^2\right]$$
Target
Prediction
Parameter update at every C iterations
Q′
Target Network
Q
Prediction Network
Input

Fig. 3

C. Experience Replay

We store the agent's experiences instead of executing Q-learning on state/action pairs as they occur during simulation or the actual experience, the system saves the data discovered for [state, action, reward, next state] in a huge table to execute experience replay.

IV. Vehicle Model

Models of mobility of car-like vehicles are widely used in control and motion planning algorithm to approximate a vehicle's behaviour in response to control actions. A high-fidelity model may accurately reflect the response of the vehicle, but the added detail may complicate the planning and control problems. This presents a trade-off between the accuracy of the selected model and the difficulty of the decision problems. $Kinematic\ Single-Track\ Model$ have been used in this project for modelling the dynamics of the motion of the car which is the simplest model available consisting of two wheels connected by a rigid link and is restricted to move in a plane [9],[10].

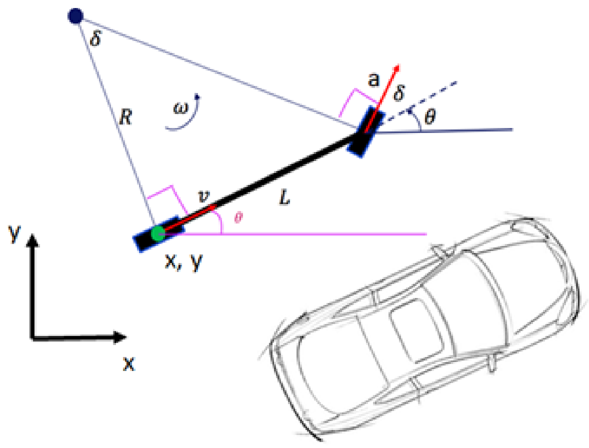

Fig. 4: Kinematics of the single track model

It is assumed that the wheels do not slip at their contact point with the ground, but can rotate freely about their axes of rotation. The front wheel has an added degree of freedom where it is allowed to rotate about an axis normal to the plane of motion. This is to model steering. These two modeling features reflect the experience most passengers have where the car is unable to make lateral displacement without simultaneously moving forward. In reference to the figure 3, the vehicle rear wheel is positioned at (x, y), with distance L from the front wheel and heading angle θ. In fact, vehicle current state at certain time t can be described with a triplet (x_t, y_t, θ_t). Vehicle model is controlled with only two inputs namely acceleration a and steering angle δ at any time t to simplify the control algorithm. Controlling the vehicle direction and its speed implies updating the vehicle's state to a new position such that $(x_{t+1}, y_{t+1}, \theta_{t+1})$ converges toward the desired location. vehicle's state is updated to new position according the differential equations :

$$\dot{x} = v\ *\ cos(\theta) \tag{15}$$
$$\dot{y} = v\ *\ sin(\theta) \tag{16}$$
$$\dot{\theta} = \frac{v}{L}\ *\ tan(\delta) = \omega \tag{17}$$
$$\dot{v} = a \tag{18}$$

v,a denotes the velocity and acceleration of vehicle along the body and along the front wheels respectively. ω denotes the angular velocity with R as the turning radius of the vehicle. The model does not take into account the lateral motion of the vehicle which does exist during situations and is very much during situations like drift in reality. The vehicle is supposed to follow a reference path and the control process is based on calculating and minimizing the error between the vehicle current position at time t with respect to the reference path using (δ, a). To measure the amount of error between the vehicle current trajectory and reference trajectory, a metric named Cross-track error(XTE) is often used and is defined as the minimum distance of the vehicle current position to the reference trajectory. XTE can be used to model the vehicle travelling through a road as exceeding the a certain XTE threshold can be considered as vehicle leaving the road. Note that the task of control the vehicle to reaches the checkpoint in the least amount of time is not trivial and need a lot of trails and errors to finding the best fit and becomes even more difficult is the reference curve changes. so Q-learning an RL algorithm is used which can generalizing the control strategy of the vehicle [11].

V. Simulation

Simulations are written in Python and Pygame library [12] is used for rendering 2D simulation to the screen. Reference trajectory is primarily made from points and then spline functions are used to generate a path passing through the points. No RL library has been used so as to have a full control over simulations and the code in written in OOP(Object Oriented Programming) [13] for modularity.

A. Path Editor

Reinforcement learning agents under consideration are simplest of the form and do solve complex environments. Even the shaping corner of track can throw off the agent which is different to doing using spline implementation from python libraries as they do not provide much control over the path curvature. A custom implementation of Catmull-Rom spline is used to make the path from the control point given by the user. This implementation sacrifices controllability for ease of use with smooth curvatures. Spline point distribution is another area where the library implementation fails and here a uniform point distribution is generated by first generating the spline for path length and then a uniform distribution is generated by dividing the path into equal segments. Minimum possible point distribution is generated as a number of points directly related to the environment performance during the simulation. Upon running the script path.py(contains the path editor) the keyboard commands for using the editor are printed and few parameters like the spline resolution, path width and PPU(pixel per unit) are shown on the screen. Path itself is divided into outer and inner regions corresponding to the rewards -3 and 0 in the environment and are of Quarter of total path width. The line at the end represents the finish line for which the agent gets 1000 reward for crossing. Paths made in the path editor are saved in json format for latter use in the environment with needing to do the spline computation.

B. Car Model

Car Model is based on $single\ track\ model$ and the updates of the position (x, y), velocity v, heading angle (θ) and steering angle (δ) are based on (14)-(17) and also are subjected to mechanical limits i.e all have a maximum and minimum value about which they are allowed. All the angles are measured with respect to x-axis If the values is about to increase in simulations they will be capped to their nearest limits. Following are the attributes of the car as written in the code :

- $position$: (x, y) co-ordinates of the car at time t in the episode
- $velocity$: forward velocity at time t in the episode
- $angle$: heading angle(θ) at time t in the episode
- $length$: distance between the front and rear wheel (L)
- $max\_acceleration$: maximum acceleration(a) allowed in forward and reverse direction
- $max\_steering$: maximum steering(δ) allowed in both directions
- $brake\_deacceleration$: amount of de-acceleration in case of braking
- $free\_deacceleration$: de-acceleration to model the friction
- $steering\_speed$: how fast the steering angle increments
- $acceleration\_speed$: how fast the acceleration increments
- $acceleration$: acceleration of the car at time t in the episode
- $steering$: steering angle of the car at time t in the episode

All the above values are initialized at the start of the training process based on the starting state of the agent. To convert the problem to finite state MDP, the action space has been discretized to 7 action from the continuous action space. The allowed actions by the agent are (as written the code):

- $pedal\_gas$: increments acceleration by $acceleration\_speed$
- $pedal\_brake$: apply brake
- $pedal\_none$: no action on the accelerator and friction will be applied
- $pedal\_reverse$: decrements acceleration by $acceleration\_speed$
- $steer\_right$: decrements acceleration by $steering\_speed$
- $steer\_left$: increments acceleration by $steering\_speed$
- $steer\_none$: no change in the steering angle(δ)

The update model for control inputs (a, δ) is Algorithm 2 and update model for the car is Algorithm 3 based on (14)-(17).

Algorithm 3 Control Inputs Update Model

```
Require: action from the agent
 1: if action is $pedal\_gas$ then
 2:     increment $a$ by $acceleration\_speed$
 3: else if action is $pedal\_brake$ then
 4:     decrement $a$ by $brake\_deacceleration$ if $v$ is non zero
        else $v$ becomes zero
 5: else if action is $pedal\_reverse$ then
 6:     decrement $a$ by $acceleration\_speed$
 7: else if action is $pedal\_none$ then
 8:     apply friction $(free\_deacceleration)$ if $|v|$ is non zero
        else $v$ becomes zero
 9: else if action is $steer\_right$ then
10:     decrement $\delta$ by $steering\_speed$
11: else if action is $steer\_left$ then
12:     increment $\delta$ by $steering\_speed$
13: else if action is $steer\_none$ then
14:     leave $\delta$ as it is
15: end if
Ensure: $a \in [-max\_acceleration, max\_acceleration]$
Ensure: $\delta \in [-max\_steering, max\_steering]$
```

In algorithm 4, R represents the turning radius of the vehicle and ω represents the angular velocity of the vehicle about the rotation point about which the turning radius id calculated.

C. Environment Setup

Environment consists of car model and path which can be loaded from the paths.json file which contains all the saved paths and can stimulate and render the environment using pygame. OpenCV package is used to capture and save the simulation as MP4 video file [14]. XTE value is approximated by calculating the minimum distance between car position and points on the path and hold good for path with sufficient spline

Algorithm 4 Car attributes Update Model

```
Require: action from the agent and time increment $dt$
 1: Update $(a, \delta)$ by following Algorithm 2
 2: $x\ position \leftarrow x\ position + v * cos(\theta) * dt$
 3: $y\ position \leftarrow y\ position + v * sin(\theta) * dt$
 4: $v \leftarrow a * dt$
Ensure: $v \in [-max\_velocity, max\_velocity]$
 5: if $\delta \neq 0$ then
 6:     $R \leftarrow \frac{L}{sin(\delta)}$
 7:     $\omega \leftarrow \frac{v}{R}$
 8: else
 9:     $\omega \leftarrow 0$
10: end if
11: $\theta \leftarrow \omega * dt$
```

resolution. Car reaching the finish line is also approximated as minimum distance between the finish line and the car position less than a threshold value instead of checking the collision which is computationally expense in the simulation. Update rate dt of the simulation is set as 1/60 seconds for a 60 FPS frame rate during render and can be decreased for more accurate simulation. Update rate of the simulation and rendering rate(60 FPS) can be in controlled individually so that the rendering does not take up all the machines resources and simulation update can run faster. The actual position of the car in meters is scaled on the screen during rendering by a factor $pixel\ per\ unit(ppu)$ to have more visual movement on the screen. Environment is also responsible for ending the simulation if the car has reached the finish line or if car crossed the map containing the path, providing the observations and rewards for the agent. Environment also renders grey vertical and horizontal lines dividing the screen into equal squares where each square represent a discrete state from the perceptive of the agent.

D. RL Model Configuration and Training

Q-learning agent and Deep Q-learning agent are used for optimizing the vehicle speed and steering direction according to the path. state space, action space are discrete as follows so that the Q-table is of smaller size in case of Q-learning and avoids high correlation in case of Deep Q-learning. Large Q table requires more memory and computation resources during training which is not feasible, high correlation shows negative impact on the neural networks used in Deep Q-learning during the training phase.

- **State Space** : 100, 60 equally spaced samples are used to discretize x, y co-ordinates which are received as observation from the environment for 591m X 341m track. XTE is also included in the observation space and is sampled at 2 points to have 3 possible states representing the center(thick yellow color), edge(light yellow color) and outer(grey color) regions. In more details, there are 100 samples for x axis (from 0 to 591 m), 60 samples in y axis (from 0 to 341 m) and 3 samples for the XTE 0 to 10 m,10 to 20 m,$\geq$ 20 m. This will result in a total of 18000 states $(100 \times 60 \times 3)$.
- **Action Space** : Only 4 actions are allowed to be taken by agent to encourage the agent to reach the finish line faster. $pedal\_brake$, $pedal\_none$, $steer\_none$ are not included in the action space.
- **Reward Model** : Agents will receive +1000 reward for reaching the finish line and -1000 for going out of the map and for every other step, agent reward will be a sum of xte reward , progress reward and frame reward. Xte reward is -10 for going out of the path, -3 for traveling on the edge and 0 for traveling in the center. Progress reward is +25 for traveling every 9m and 0 otherwise and finally frame regards is kept as -1 always to encourage agents to complete the path as fast as possible.

Q-Learning episodic approach with learning rate (α) = 0.01, discount factor (γ) = 0.9 and exploratory rate of 0.01 is used for training. ϵ for the $\epsilon - greedy\ policy$ is decremented from 1 to Exploratory rate within first half of the training process so that agent can initially explore in the fist half of episodes and exploit in second of the episodes with less exploration. Q-table is 4 dimensional tensor with x, y, XTE, $action$ along each dimension. For Deep Q-learning the same exploration rate of 0.01 and discount factor of 0.9 is used with memory size of 10,000 for experience replay buffer, learning rate of 0.001 for gradient descent with batch size of 32 is used to training the local neural network and Huber-loss is used as loss function. Adam optimizer is used to adjust the weights of local neural network using the back-propagation algorithm. weights of local network are copied to target network after every 500 steps taken by the agent. ϵ is also annealed from 1 to exploration rate linearly for first half of the episodes so that next half of the episodes can be used to exploit by the agent. Both the target and local neural network are of same topology having two 128 neuron hidden layers and fixed seed value of 333(random fixed number) is used to initialize the random weights for both neural networks. All the neural networks, state transitions and replay buffer are converted to Pytorch tensors and sent to the GPU if available or CPU for efficient matrix computations. Pseudo-code for training the RL agent is Algorithm 5. Deep Q-learning is an extension of Q learning by using a Neural network as a Q function instead of Q table and the pseudo code is also similar and is implemented by modifying algorithm 5 according to the algorithm 2.

E. simulation Results

Straight path with little curve fig[5] and highly curved path fig[6] is used for testing the performance of Q-learning and Deep Q-learning by training them on both for 30,000 and 40,000 episodes respectively and Deep Q-learning was found to perform well. Both Q-learning agent and Deep Q-learning agent were able to complete the straight path but the latter was able to complete with more net positive reward as the agent almost drove in the center region at full speed. In the highly curved path Q-learning agent was unable to complete the first sharp turn in the path let alone the next two and on the other hand Deep Q-learning agent successfully took the first two

Algorithm 5 Q-learning agent training

```
Require: step size $\alpha \in (0, 1]$
Ensure: small $\epsilon \geq 0$
 1: Initialize the environment with path data Initialize the
    $epsilon\_decay\_rate$ Initialize $Q(s, a)$, for all $s \in S$,
    $a \in A(s)$, arbitrarily except that $Q(terminal, .) = 0$
 2: for each episode do
 3:     Reset the environment and agent state
 4:     repeat
 5:         For each step of episode
 6:         Discretize the observation
 7:         Choose an action $a$ in current state $s$ using
            $\epsilon$-greedy policy derived from $Q(s, a)$ table
 8:         Take action $a$
 9:         Observe the outcome state $s^{'}$
            and reward $R(s, a, s^{'})$
10:         update $Q(s, a)$ like (13)
11:         $s \leftarrow s^{'}$
12:         Render and save the episode for every few episodes
13:     until $s$ is terminal or simulation time
        for the episode exceeds 2.5 minutes.
14:     save the Q-table for every few episodes
15:     save the average, minimum and maximum rewards for
    every few episodes to see the agent progress
16:     Decrement the $\epsilon$ by $epsilon\_decay\_rate$ if
        epsilon is less than $Exploratory\_rate$
17: end for
```

sharp turns but failed at the last smooth which is due to the overfitting of the small neural local and target neural networks due to their small size. Large networks are not an option for this setup as these two 256,256 hidden layer networks took 1 day to train due to the OOP's execution bottleneck in the current simulation pipeline. Based on fig[7], fig[8] Deep Q-learning agent scores better total reward in straight path and also converges faster and flatten out than Q-learning agent. Both the agents do not cross the zero rewards and the maximum reward is still much higher, showing that still there is some room for improvement, also Deep Q-learning agent average rewards is closer to maximum rewards. On curved path Q-learning agent showed little to no progress whereas Deep Q-learning agent almost completed the path failing at the last small stretch Episode 40,000 with bump in the average reward representing agent reaching the finish line through exploration in fig[9]

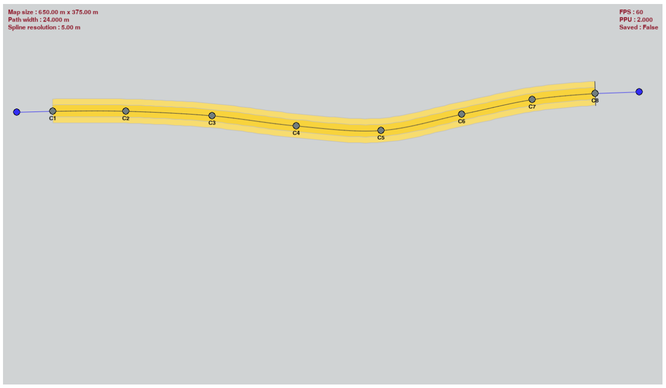

Fig. 5: Straight path made using the path editor tool. Grey points named C1,C2 .. represents the control points for shaping the track and blue control points are used to set the orientation of start and end of the track.

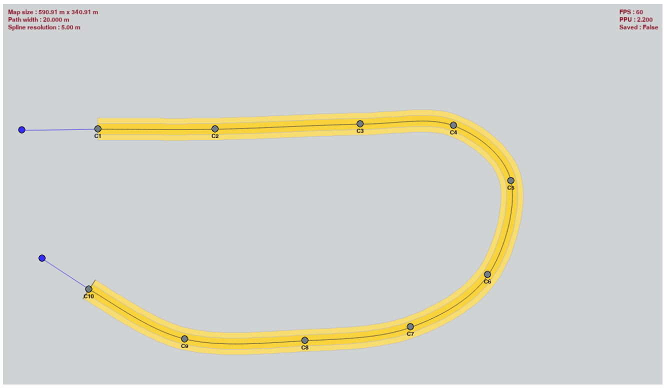

Fig. 6: Curved path made using the path editor tool. Grey points named C1,C2 .. represents the control points for shaping the track and blue control points are used to set the orientation of start and end of the track.

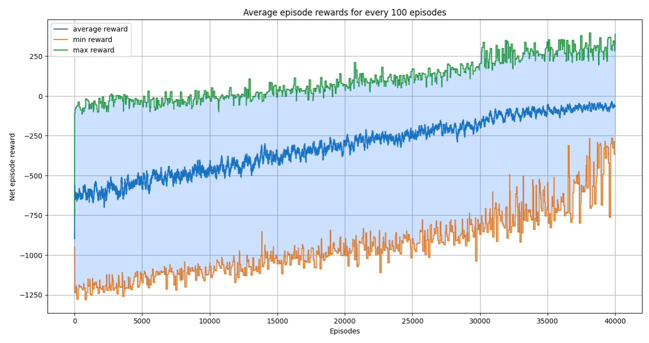

Fig. 7: Average episode rewards of Q-learning agent on straight path. Blue shaded region represents total episode reward and green, orange, blue represent minimum, maximum and average episode reward for every 100 episodes for 30,000 episodes.

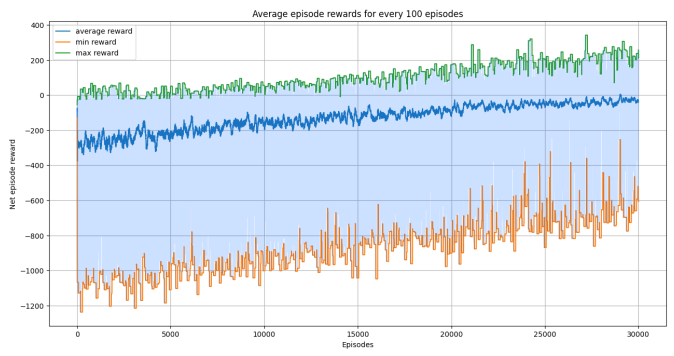

Fig. 8: Average episode rewards of Deep Q-learning agent on straight path. Blue shaded region represents total episode reward and green, orange, blue represent minimum, maximum and average episode reward for every 100 episodes for 30,000 episodes.

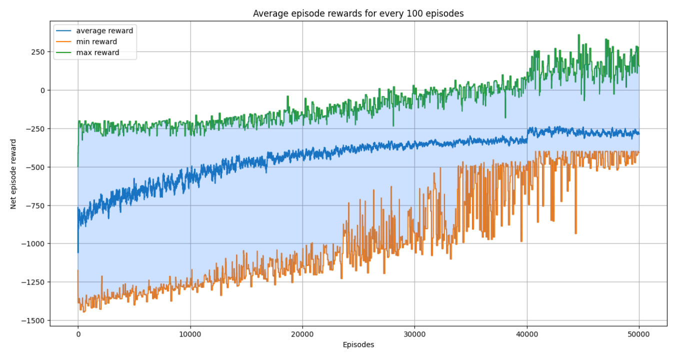

Fig. 9: Average episode rewards of Deep Q-learning agent on curved path. Blue shaded region represents total episode reward and green, orange, blue represent minimum, maximum and average episode reward for every 100 episodes for 30,000 episodes.

Environment Recording of final episodes for both agents are in Final episode recordings and the Code files are in the OELP GitHub repository.

VI. Conclusion and Future Works

The Environment setup need for training and testing a RL model has been developed successfully developed and optimized to perform the simulations need in reasonable amount of time. This also shows that computational power plays significant role in driving progress in reinforcement learning research. Naive Q-learning approach is updated using Neural Networks as value function approximator leading to better solution to the self driving problem under consideration.Reward model plays a crucial role in the agent performance and is little handcrafted showing that observation model used in incomplete for optimization, a better model model would use sensors in front and side of the car for assigning rewards. XTE can not provide any information about the orientation of the car given a state without having knowledge about the previous state which can be solved using a sensor model. It can also relieve the dependency on full observability of the environment. Oscillations in the episode rewards show signs of over estimating Q values . Discrete Observation space can be converted to continuous space with frame stacking [8] and multiple gradient descent iterations for each agent step for reliable neural network training. Randomizing the path during the training has not been considered. we also have to experiment with others forms of Q-learning like double Q-learning. [15] to improve the performance. Action space has to be changed from discrete to continuous which is more the use full as a control strategy for real world simulations. OELP repo has the codes files, requirement file for anaconda python environment and saved paths, hyperparamters for Q-learning, Deep Q-learning json files.

References

[1] J. Kober, J. A. Bagnell, and J. Peters, "Reinforcement learning in robotics: A survey," *The International Journal of Robotics Research*, vol. 32, no. 11, pp. 1238–1274, 2013.
[2] [Online]. Available: https://deepmind.com/learning-resources/-introduction-reinforcement-learning-david-silver
[3] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.
[4] J. A. Boyan, "Technical update: Least-squares temporal difference learning," *Machine learning*, vol. 49, no. 2, pp. 233–246, 2002.
[5] D. Silver, "Deep reinforcement learning," 2016.
[6] N. Vithayathil Varghese and Q. H. Mahmoud, "A survey of multi-task deep reinforcement learning," *Electronics*, vol. 9, no. 9, p. 1363, 2020.
[7] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, 2012.
[8] 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.
[9] A. D. Luca, G. Oriolo, and C. Samson, "Feedback control of a nonholonomic car-like robot," in *Robot motion planning and control*. Springer, 1998, pp. 171–253.
[10] T. Fraichard and R. Mermond, "Path planning with uncertainty for car-like robots," in *Proceedings. 1998 IEEE International Conference on Robotics and Automation (Cat. No. 98CH36146)*, vol. 1. IEEE, 1998, pp. 27–32.
[11] [Online]. Available: https://ksp-windmill-itn.eu/research/autonomous-vehicle-control-using-reinforcement-learning/
[12] [Online]. Available: https://github.com/pygame/pygame
[13] [Online]. Available: https://www.geeksforgeeks.org/python-oops-concepts/
[14] [Online]. Available: https://github.com/tdrmk/pygame_recorder
[15] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 30, no. 1, 2016.