Autonomous Vehicle Navigation Using Reinforcement Learning Techniques

Team Members:

Aditya Jain, jainadit@usc.edu
Isha Chaudhari, ichaudha@usc.edu
Aditi Jain, amjain@usc.edu
Adwaita Jadhav, ajjadhav@usc.edu
Devanshi Desai, dhdesai@usc.edu
Harshita Bhorshetti, bhorshet@usc.edu
Saichand Duggirala, sduggira@usc.edu

Table of Contents

1. Goal

The goal of our project is:

- To train an autonomous vehicle to navigate in a particular environment in the presence of static and dynamic obstacles like pedestrians and vehicles.
- To predict the movement of the vehicle taking into account the environment.
- To predict and track the maximum distance and time the car can drive autonomously without crashing.

2. Game Overview

2.1. Background

The goal of this project is to build an agent to perform autonomous driving in a simulated environment and tackle all the obstacles present along its way. Autonomous driving has the potential to drastically improve our lives. An autonomous vehicle is capable of sensing its environment and operating without human involvement. A human passenger is not required to take control of the vehicle at any time, nor is a human passenger required to be present in the vehicle at all. An autonomous car can go anywhere a traditional car goes and do everything that an experienced human driver does. Simulation is pivotal in self-driving research because we can test customized situations without facing real-world consequences in less time.

2.2. Overview

We have used CARLA[3] as the simulation environment as it is an open source simulator specially developed for autonomous driving research which supports development, training and validation of autonomous systems. The main agent is a car which is trained to navigate autonomously in the environment such that it follows the correct path and takes the appropriate turns. In order to navigate the car through the environment at the time of training, users can use the keyboard's left and right arrow keys to go left or right respectively. Depending upon the action that the car takes, the reinforcement learning assigns a reward or a penalty to the agent which helps it to further learn the correct actions. The game / model is considered to be successful if it manages to traverse the entire path without crashing, going out of lane, or stopping suddenly. The other agents on the environment are pedestrians that are walking on the road and other vehicles.

3. Prior Research

3.1 CARMA: A Deep Reinforcement Learning Approach to Autonomous Driving[12]

This work uses Deep Q-Network i.e. DQN to perform the task of autonomous car driving using information from various sensory inputs. To implement the DQN, the paper has used a combination of CNN and RNN networks. The paper revolved around designing the hand-crafted car controller to safely navigate around a track in a simulated environment. They experimented with various agents to understand which agent will be perfect for their use case. The agents they experimented with included:

a. Discrete Q-learning agent, where the agent could vary 4 modes of car speed, 3 angles from the center of the track and 4 distance from the center of the track.
b. Deep Q-learning agents, which again had 3 types:
    1. CNN agent, where the images were passed through multiple CNN layers to calculate the Q-values. The drawback of this approach was that the agent was not able to make complex turns.
    2. RNN agent, where the racing features were passed through LSTM(Long Short-term network) to calculate the Q-values.Here, the agent could drive smoothly, but the drawback of the approach was that it was slow as compared to CNN.
    3. Hybrid CNN-RNN agent, which was a combination of CNN and RNN where results of CNN and RNN were concatenated to calculate the Q-values.

After finalizing the Hybrid CNN-RNN approach, the authors compared their Hybrid CNN-RNN agent against their discrete Q-learning agent, hand-crafted car controller, and a greedy agent to maximize its immediate reward at each timestep. Here is their comparison,

	Average Reward	**Average Speed**	**Max Speed**
Hand Crafted Controller	0.542	17.65 m/s	18.77 m/s
Greedy Agent	1.416	17.53 m/s	18.16 m/s
Discrete Agent	0.635	18.71 m/s	20.21 m/s
DQN Agent	0.915	17.57 m/s	24.71 m/s

Fig 3.1 Performance comparison between different agents

Looking at the comparison, DQN agent exhibited more unstable behavior than the traditional discrete agent. The authors were successful in training a DQN to reliably drive around the race track. Both the methods definitely outperformed the hand designed controller in both their average cumulative rewards and maximum driving speeds.

3.2 Driverless Car: Autonomous Driving Using Deep Reinforcement Learning In Urban Environment[8]

In this study, the author outlines how the deep reinforcement learning algorithms can be used in complex control and navigation tasks. They used Deep Q network for car navigation and obstacle avoidance in an urban environment.

In this work, a deep fully connected convolutional neural network was used to approximate the Q-function based on the image input received from the vision based camera and laser sensor. Then, experienced replay based reinforcement learning is applied using a reward function to train the agent. The network consisted of 3 convolutional layers and 4 dense layers. The image data is fed into the convolutional neural network for each data type. Two data types are fused using a flatten layer, preceded by a dense layer and a pooling layer (used for data resize) for each data type. They trained their network to accept 5 actions with defined reward.

They fed the neural network with 4 consecutive frames of the camera (front, size 80x80) images. The lidar data are plotted as map frames, and 4 consecutive frames are fed into the network. Both the data types are fused using a flatten layer preceded by a pooling layer pooling from last convolution and first dense layer. The simulation tests showed that the trained neural network was capable of driving the car autonomously by taking proper decisions (choose an action from action set) in the environment.

3.3 DeepRacer: Educational Autonomous Racing Platform for Experimentation with Sim2Real Reinforcement Learning

This study describes the renowned AWS DeepRacer learning platform which AWS has deployed in a game-like form to help understand how reinforcement learning works in the physical and virtual environment. This employs a $1/18^{th}$ scale car in the physical environment which has a fisheye lens camera mounted on it live streaming camera view over WiFi and a GPU to execute the DRL network. The virtual environment has a simulated robot model with simulations for multiple racing tracks as well.

DeepRacer supports DRL models implemented on AWS SageMaker, simulations with the OpenAI Gym interface and integration with AWS cloud services coupled with distributed rollouts.

The environment, the agent, the observation space and the potential actions that the agent can take at each point of time are all described as shown. The agent does not have knowledge of the full state which in this case, is the track layout, hence, this becomes a Partially Observed Markovian Decision Process problem. The incentivization process is to keep the agent as close to the center line as possible, rewarding the agent if it follows this policy at each step.

The Proximal Policy Optimization[4] (PPO) algorithm is used as the policy gradient algorithm. The policy network interacts with the simulated environment to collect test data. This dataset is then further used to update the policy and value networks as per the loss function of the algorithm. This updated policy is then used to interact with the environment to collect more data and this cycle will continue till a stipulated time limit, with the loss function maximizing the actions giving higher rewards on average along with a clipped importance sampling weight as the policy collecting the dataset is an older version of the policy being updated. Loss function uses the mean squared error between the value predicted and value observed. The default implementation utilizes 3 conv layers and 2 fully connected layers for both networks with a new policy being trained every 20 episodes.

3.4 Deep Imitation Learning for Autonomous Vehicles Based on Convolutional Neural Networks[13][15]

In the case of autonomous vehicles, CNN models are utilized for providing them with a visual sense and perception of their environment. The authors began the training with a basic CNN architecture of 3 convolutional layers, 4 filters of dimensions 3 x 3. Eventually they have introduced additional multiple nodes in which the filter sizes change after the max-pooling layer, 7*7 for the 1st layer, 5*5 after the 1st max-pooling layer, and 3*3 for the final max-pooling layer. Together 96 CNN models are developed. The following steps are followed to train the model using imitation learning:

a) Collection of images of the car while it is being driven.
b) Train the CNN model and use the images to update the parameters of the CNN model. The model controls policy by imitation, same as the idea of pixel to action.
c) To prevent overfitting due to an imbalanced dataset, the generator is forced to augment images in real-time, while the models are being trained.
d) Models are being tested on 20% of the data. Once all models have been trained and tested, they are evaluated on an autonomous vehicle for efficiency analyses.

3.5 Exploring Imitation Learning for Autonomous Driving with Feedback Synthesizer and Differentiable Rasterization[14]

This paper focuses on developing a learning-based planner that aims to robustly drive a vehicle by mimicking human drivers' driving behavior. It makes use of a mid-to-mid approach that allows manipulating the input to our imitation learning network freely. It further proposes a novel feedback synthesizer for data augmentation. It allows the agent to gain more driving experience in various previously unseen environments that are likely to encounter, thus improving overall performance. The model consists of three parts, a CNN backbone with a branch of spatial attention module that is stemmed from intermediate features of the CNN, an LSTM decoder taking the feature embeddings from the CNN as inputs and outputting planning trajectories in the time domain, a differentiable rasterizer module that is appended to the LSTM decoder during training rasterizing output trajectories into images and finally feeding them to our loss functions.

During training time, the output trajectory points are rasterized into N images.
Given the vehicle waypoints st = [xt, yt, φt, vt] T , vehicle length l, and width w, our rasterization function G rasterizes images as:

$$\mathcal{G}_{i,j}(\mathbf{s}_t) = \max_{k=1,2,3}(\mathcal{N}(\mu^{(k)}, \mathbf{\Sigma}^{(k)})),$$

where each cell (i, j) in G is denoted as Gi,j . Let $x_t^k = [x_t^k , y_t^k]$ T denote the center of the k-th Gaussian kernel of our vehicle representation. Besides the imitation loss, four task losses are introduced to prevent our vehicle from undesirable behaviors, such as obstacle collision, off-route, off-road, or traffic signal violation.
Benefiting from the differentiable rasterizer, their learning-based planner runs inference in real-time, yielding output either directly used by a downstream control module or further fed into an optional post-processing planner. Task losses and spatial attention are introduced to help their system reason critical traffic participants on the road. Therefore, the full exploitation of the above modules enables the system to avoid collisions or off-road, smoothly drive through intersections with traffic signals, and even overtake slow vehicles by dynamically interacting with them.

4.Methodology

4.1.Data Collection

For data collection, we are using the CARLA environment's images. The environment offers three different types of views: LIDAR view, front view and bird eye view. We took 10000 screenshots of the bird eye view specifically in the states where the lanes had a right turn, a left turn or straight in the presence of obstacles in the form of other cars and pedestrians. In order to navigate the vehicle through the environment during data collection, we used the arrow key left for action = 1, arrow key right for action = -1, else action = 0.

4.2. Data Preprocessing

The dataset consists of 10,000 images, initially at the time of training the model the system would go out of memory because of the huge dataset size. Hence we decided to encode the images through a variational encoder and then feed the dataset to the reinforcement learning algorithms. An encoder is a neural network that takes a high dimensional data point as input and converts to a lower dimensional feature vector and then later reconstructs the original input data sample just utilizing the latent vector representation without losing valuable information. Variational AutoEncoders[7](VAEs) calculate the mean and variance of the latent vectors (instead of directly learning latent features) for each sample and force them to follow a standard normal distribution. As a result the input images of resolution 128 * 128 were converted to a latent encoding vector of size 1024.

4.3 Reinforcement Learning for Autonomous Vehicle Navigation

Reinforcement learning[10] also called RL is a machine learning technique that learns based on rewarding all the desired behaviors and giving penalties to undesired behaviors. It tries to learn through the agent's interactions with the environment. This algorithm is an interaction based algorithm which is derived from human behaviours. During the training phase of this algorithm, the agent adjusts its behavior to maximize the cumulative rewards it can achieve from the entire control task according to the reciprocal action it receives from the environment. That means the car is given high rewards for good driving and given punishments for bad driving, in a well constrained reward function the car should exhibit the behavior of good driving.

4.3.1. Elements of Reinforcement Learning

4.3.1.1. Environment

Finding an appropriate environment for training the navigation model was the first step. In selecting our environment, tradeoffs between installation requirements, the complexity of the environment, and features offered were important factors influencing our decision. We decided to go with CARLA as the simulation environment as CARLA is an open source simulator specially developed for autonomous driving research which supports development, training and validation of autonomous systems. The simulator is controlled through Python and C++ APIs. The simulator has a client-server architecture where the server is responsible for the simulation and the client is responsible for controlling the logic of the actors on scene and setting world conditions. The CARLA environment consists of eight towns and each town can have either Layered or Non Layered maps. Layered maps include components like buildings, foliage, walls, street lights etc. We can set different parameters while initializing the environment like number of cars, number of pedestrians, desired speed, time interval between two frames [3].

Interacting with environment: Action is represented by a float vector of size two which denotes [acceleration, steer]. In

order to collect the dataset, we set the environment parameters as continuous acceleration range = [-3.0 (brake), 3.0 (accelerate)], continuous steering angle range = [-3.0 (right), 3.0 (left)], desired speed 8 m/s. Initially the car always throttles i.e. acceleration = 1.

4.3.1.2. Agent

The agent perceives the environment through sensors (cameras in our case) and acts upon the environment using actuators. The main agent in the system is the vehicle / car whose goal is to navigate through the environment, take appropriate left or right turns on the lanes, and maximize its cumulative reward by taking the optimal actions. The vehicle has different controls which helps to simulate the physics of wheeled vehicles: speed, throttle, brake, direction etc. Apart from this, we have introduced other actors in the system like static and dynamic obstacles in the form of moving vehicles and pedestrians. The pedestrians / walkers can be moved around in the environment with a certain direction and speed. The system also provides other actors like traffic signs, traffic lights etc.

4.3.1.3. State

State is the observation that the agent does on the environment after performing an action. If the current state is that the car is present on a straight lane, and it performs an action of taking a right turn or left turn then the environment goes into a different state which then becomes the current state.

4.3.1.4. Action

The agent can perform any of the four actions depending on its current state:

1) Left turn
2) Right turn
3) Go straight - accelerate
4) Brake

4.3.1.5. Reward

Reward is the feedback that the agent receives based on the action it performed. If the feedback is positive the agent receives a reward else it receives a punishment. For every *(state, action)* pair, if the agent takes the correct action i.e. making steer += 1 when the lane has a right turn, making steer -= 1 when the lane has a right turn or not making any change in the steer at all, the algorithm gives a positive reward to the agent, else it gives a negative reward. We built a reward function which would allow the agent to navigate the roundabout, this reward function was made up of multiple factors. A high negative reward is given to the agent if it collides with any other vehicle or pedestrian in order to discourage it from doing it more often.

Reward Type	**Cost**
collision	-200
too fast driving	-10
out of lane	-1
steering (opposite direction)	-5
steering (expected direction)	+5

4.3.2. Deep Q Network (DQN)

Reinforcement learning[5][9] works well in all environments where all the achievable states can be stored in the standard computer RAM memory. However, when the number of states overwhelms the RAM memory of standard computers, Deep RL networks cannot be used. Also, in practical scenarios, our car agent has to learn continuous states and continuous variables, rather than discrete ones as used in the Deep RL networks. Deep Q Networks or DQNs allows the agent to acquire knowledge by exploring any unstructured environment over a time.

The steps followed for training the Deep Q networks are:

1. All the past experience is stored by the user in memory
2. The next action is determined by the maximum output of the Q-network
3. The loss function here is mean squared error of the predicted Q-value and the target Q-value – Q*. This is basically a regression problem. However, we do not know the target or actual value here as we are dealing with a reinforcement learning problem.

Our DQN model is based on VGG16 architecture for feature extraction and has the Global Average MaxPooling Layer (GAP Layer) along with a Dense Layer, having a learning rate of 0.001 and Adam optimizer. Our model is trained with a replay memory of maximum length 3000 consisting of previous steps, more specifically, the current state, action, reward and new state for every episode. We have defined the loss function as the squared difference between the target and predicted value, and we have also tried to minimize the loss by updating the weights (assuming that the Agent performs a transition from one state s to the next state s' by performing some action a and receiving a reward r). We are limited to defining only one action: to steer in discrete space with values as −1, 0, 1. We also used the epsilon greedy strategy by starting with = 1 and decreasing by 0.99 for each episode.

The TD error was calculated using the following formula:

$$Q(S,\ A)\ =\ Q(S,A)\ +\ \alpha(R\ +\ \gamma max_{a'}Q(S',\ A')\ -\ Q(S,\ A))$$

Where,
TD target is $(R\ +\ \gamma max_{a'}Q(S',\ A'))$
TD error is $(R\ +\ \gamma max_{a'}Q(S',\ A')\ -\ Q(S,\ A))$

4.3.3. Prioritized Experience Replay (PER)

The key idea is that an RL agent can learn more effectively from some transitions than from others. Transitions may be more or less surprising, redundant, or task-relevant. Some transitions may not be immediately useful to the agent, but might become so when the agent competence increases. Experience replay liberates online learning agents from processing transitions in the exact order they are experienced. Prioritized replay further liberates agents from considering transitions with the same frequency that they are experienced. In particular, replaying transitions frequently with high expected learning progress, as measured by the magnitude of their temporal-difference (TD) error. This prioritization can lead to a loss of diversity, which we alleviate with stochastic prioritization, and introduce bias, which we correct with importance sampling. Prioritized Experience Replay (PER) is one of the most important and conceptually straightforward improvements for the Deep Q-Network (DQN) algorithm. It is built on top of *experience replay buffers*, which allow a

reinforcement learning (RL) agent to store experiences in the form of transition tuples, usually denoted as (s_t, a_t, r_t, s_t+1) with states, actions, rewards, and successor states at some time index *t*.

4.3.4. Double Duelling DQN with PER

The duelling architecture[6] separates the representation of state values and state dependent action advantages via two separate streams. The main reason behind this architecture is that in some situations/games it is not necessary to know the value of each action at every timestep. By dividing explicitly two separate estimators, the duelling[2] architecture is able to differentiate which steps are valuable without going through the learning of the effect of each action for each state. Given the agent's policy pi the action value and state values are defined as, respectively:

$$Q^{\pi}(s, a) = E|R_t|s_t = sa_t = a, \pi$$

$$V^{\pi}(s, a) = E_{a\,\pi(s)}|Q^{\pi}(s, a)|$$

The same Q function could be written as:

$$Q^{P}i(s, a) = E[r + [Q^{\pi}(s', a')]|st = s, st = s, \pi]$$

The advantage can be calculated by subtracting value Q-value by V-value:

$$A^{\pi}(s, a) = Q^{\pi}(s, a) - V^{\pi}(s)$$

The Q value represents the value of choosing a specific action at a particular state. The V value represents the value of the given state regardless of the action taken. So basically advantage value shows how advantageous selecting an action is relative to others at a given state.

4.3.4.1. Architecture

Like standard DQN architecture we have convolutional layers to process game-play frames. From there we split the network into 2 different parts one for estimating the state value and other for estimating state dependent action advantages. After the split of two streams the last layer of the network combines the state value and advantage outputs. The final output is combination of two value with some normalization:

$$Q(s, a, \theta, \alpha, \beta) = V(s, \theta, \beta) + A(s, a, \theta, \beta)$$

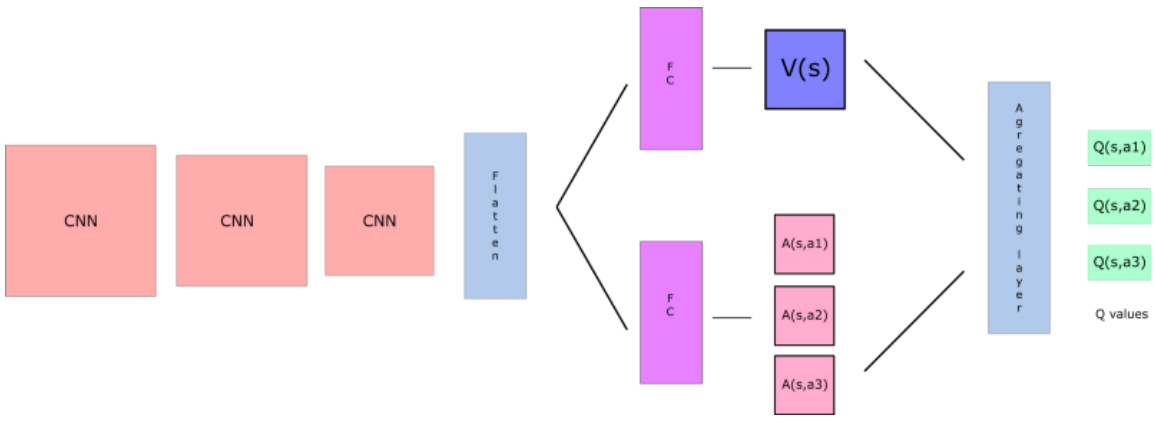

Fig. 4.3.1 Double Duelling DQN Architecture

There are 2 issues associated with this method as follows: 1) It is problematic to assume that and gives reasonable estimates of the state-value and action advantages, respectively. Naively adding these two values can, therefore, be problematic. 2) The naive sum of the two is "unidentifiable," in that given the Q value, we cannot recover the V and A uniquely. It is empirically shown in Wang et al. that this lack of identifiability leads to poor practical performance To resolve these issues we need to perform forward mapping.This will force the Q value to maximize action to equal V and further solve the identifiability issue.

$$Q(s, a, \theta, \alpha, \beta) = V(s, \theta, \beta) + A(s, a, \theta, \beta) - max(s, a, \theta, \alpha)$$

4.3.4.2. Implementation

We used the same DQN model as developed before with the convolutional layer splitting up. The main issue with using DQN along with PER was that the policy was very slow to converge. Additionally, the policy would overfit to take straight steer action after a period of time due to a constant decrease in epsilon over every episode as a result of which, it fails to learn new policies like taking a turn.

4.3.5. Proximal Policy Optimization Algorithm (PPO)

Proximal Policy Optimization(PPO) is a model-free, policy gradient based reinforcement learning algorithm and can be used for either discrete or continuous action spaces [1]. PPO trains a stochastic policy in an on-policy way and is based on actor-critic architecture. The actor maps the observation to an action and the critic gives an expectation of the rewards for the given observation. We switch between sampling data from the policy and performing several epochs of optimization on the sampled data, for achieving optimization. Each experience, stores the transition state in the form of a tuple shown below:

$$(s_{t,} a_{t,} r_{t,} d_{t,} V(s_t \bullet \theta_v))$$

d_t is a variable that represents whether or not this state is a terminal state, which is used only when we calculate the advantage estimates.

For every epoch, initially we collected a set of trajectories by sampling from the current stochastic policy. We calculated discounted cumulative sums of vectors for computing rewards-to-go G_t and advantage estimates A_t.

We calculate rewards-to-go(Gt) as follows:

$$G_t = R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} \ldots = \sum_{k=0}^{\infty} \gamma k \, R_{t+k}$$

The standard policy gradient loss function is replaced with the clipped loss function, there is a loop repeating the gradient update on random minibatch samples over K epochs, and advantage estimate Aˆ t uses the more accurate generalized advantage estimation (GAE) calculation instead. After we have obtained a trajectory, we calculate advantage estimates with the GAE equation. While implementing PPO, we faced some issues due to memory limitations as a result of which we were unable to create prolonged experiences. Since convolutions are time consuming, it was slowing down the policy convergence as well as the time to predict next best action.

To resolve these problems, we tried 2 approaches:

4.3.5.1. Variational Autoencoder

First approach is a variational autoencoder to reduce state dimension. It compresses the original input into a lower

dimensional ”code” and reconstructs the output from this code. This ”code” is a compact representation or the bottleneck in the network. To build an autoencoder, we took following 3 parts:

- *Encoding Method* : The encoding network turns input samples into 2 parameters in latent space - μ and $log(\sigma)$. Then we randomly sample similar z points via $z = \mu + exp(log(\sigma)) * \epsilon$ where epsilon is random normal tensor from the latent normal distribution.
- *Decoding Method* : The decoder network converts the latent space points back to original data.
- *Loss Functions*: The 2 loss functions using which the parameters of our model are trained are reconstruction loss function and KL divergence.

  The reconstruction loss is based on binary cross entropy and forces the decoded samples to match initial inputs.

$$reconstruction\ loss\ =\ \frac{1}{hw}\sum_{i=0}^{h}\sum_{j=0}^{w} -\ ylog(y)$$

  The Kullback Leibler divergence also known as relative entropy measures how one probability distribution is different from another. In VAE model, KL divergence acts as a regularization term between the learned latent distribution and previous distribution.

$$kl\ loss\ =\ \sum_{i=1}^{n} \sigma_i^2 + \mu_i^2 - log(\sigma_i) - 1$$

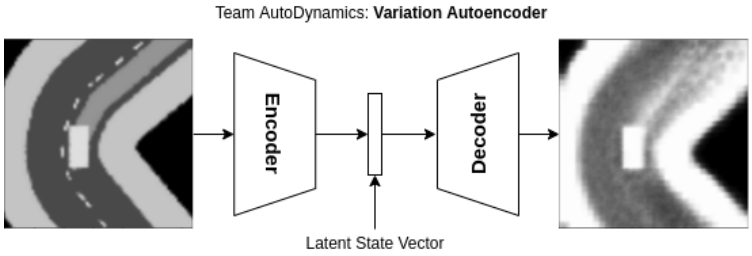

Fig. 4.3.2 Encoding and decoding image using variational autoencoder

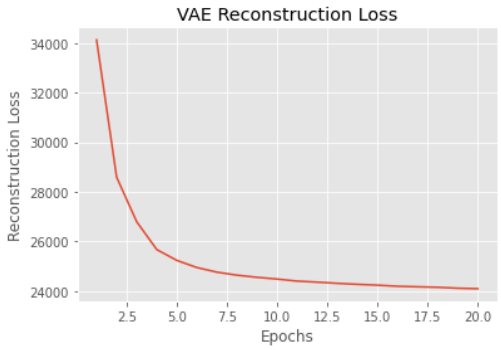

Fig. 4.3.3 VAE Reconstruction Loss

However despite solving the problem of latent space irregularity, we were facing some performance issues in this approach. As we can see in the Fig. 4.3.3 The reconstruction loss reduces exponentially in the initial few epochs and then reduces gradually over time. However, the VAE was overfitting for straight roads and common turns. Also as shown in Fig. 4.3.4, we were getting the wrong state vector for some input states. As a result PPO was getting confused and failing on corners.

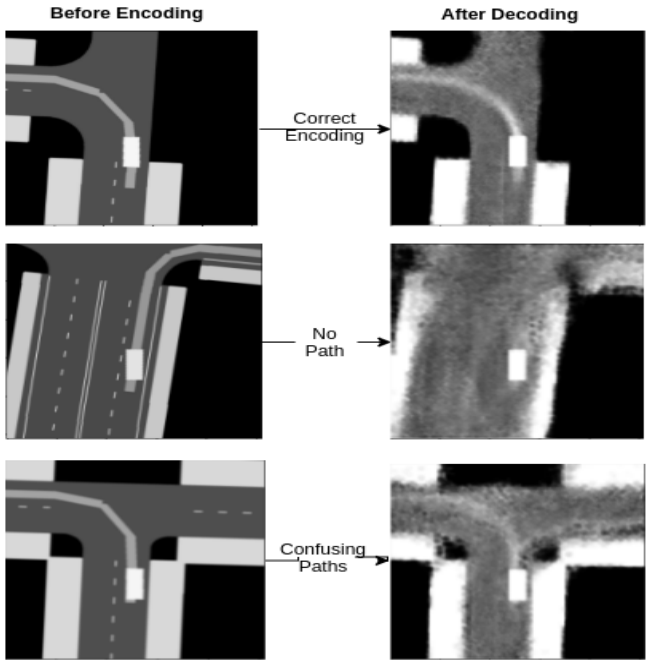

Fig. 4.3.4 VAE output for different states

4.3.5.2. Cropping State View

Second approach is cropping the state view to use only interesting areas in image for training. In our state, following areas in observation are not relevant at a given point of time.

a. Pixels behind the car
b. Pixels on footpath
c. Pixels far ahead of car

We converted a (512 x 512 x 3) image into (128 x 128 x 3) image by removing the irrelevant areas as shown in Fig. 4.3.5. This approach enabled us to deal with a smaller number of convolution layers along with longer experience buffers. It also improved the time for policy convergence as fewer parameters are to be fit. We used Adam optimizer on policy as well as critic.

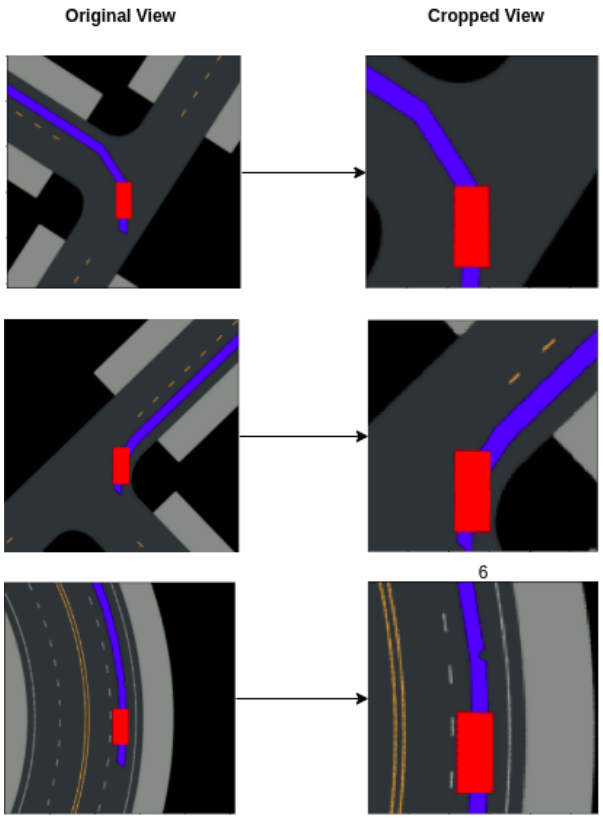

Fig. 4.3.5 Cropped view of the environment for different states

4.3.6. Imitation Learning

Imitation Learning algorithms learn a policy from demonstrations of expert behavior. Usually, demonstrations are presented in the form of state-action trajectories, with each pair indicating the action to take at the state being visited. In order to learn the behavior policy, the demonstrated actions are usually utilized in two ways. The first is Behavior Cloning which treats the action as the target label for each state, and then learns a generalized mapping from states to actions in a supervised manner. The second is Inverse Reinforcement Learning (IRL) which views the demonstrated actions as a sequence of decisions, and aims at finding a reward/cost function under which the demonstrated decisions are optimal. Compared to other learning approaches, imitation learning is easier and more efficient in terms of computations and the amount of expert knowledge required for the training process.

4.3.6.1. Implementation

Introducing static and dynamic obstacles was one of our major project goals and for training the model using imitation learning[11] we introduced other actors like vehicles and pedestrians in the environment. In order to deal with the newly made changes we introduced one more action i.e. brake and accelerate, the car will accelerate ahead if there is no obstacle in front of it, else it will take the brake action meanwhile also turning left or right depending on the environment. There are 6 possible combinations of actions that the agent can take:

1) accelerate
2) brake
3) accelerate - left
4) accelerate - right
5) brake - left
6) brake - right

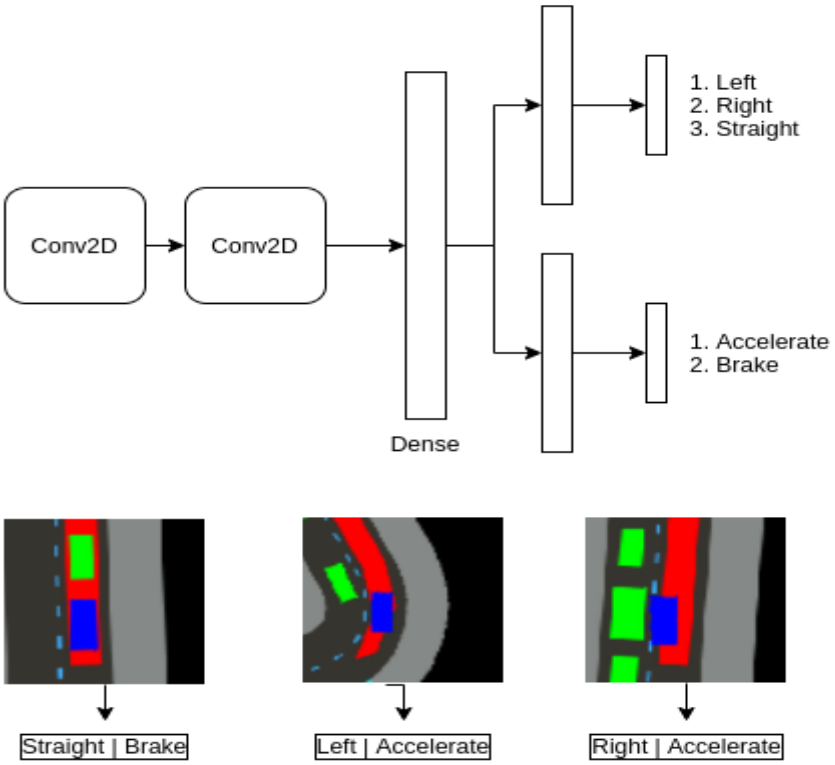

Fig. 4.3.6 Model and Training Configuration

In order to train the model, we collected around 5000 images of the car along with other vehicles and pedestrians in situations where the car is performing the above mentioned actions. As seen in Fig. 4.3.6 in the bird eye view, the blue rectangle is the main actor and the other green rectangles are the vehicles introduced as obstacles. We trained a Convolution Neural Network based model to replicate behaviour of human players in the CARLA environment. 80% of the images have been used for the training and 20% for the test. Adam optimizer and cross entropy loss is used for both the actions. There are two CNN layers with 32 and 8 filters respectively in each of them. Training was performed for 5 epochs with 0.001 learning rate. The CNN model has to predict the next possible action that the car will perform depending on the current state of the image. The CNN parameters are being updated by gradient descent algorithm and the model learns the control policy by imitation. However after the initial training, the car movements were not quite stable around the corners or the turns. Even after training the model using an increased number of images for the input dataset, the navigation was not giving a satisfactory performance. Hence we integrated the imitation learning model with the already developed PPO model.

4.3.6.2. Training Metrics

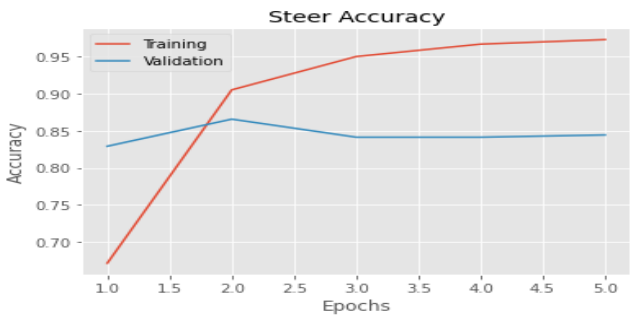

Fig 4.3.6.2 a) epochs vs steer accuracy

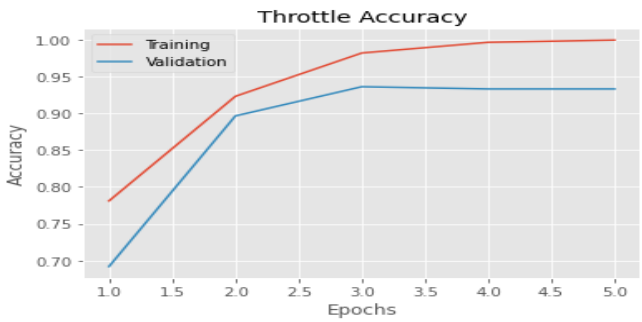

Fig 4.3.6.2 b) epochs vs throttle accuracy

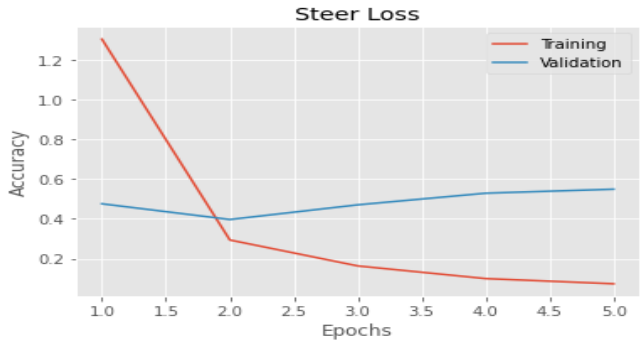

Fig 4.3.6.2 c) epochs vs steer loss

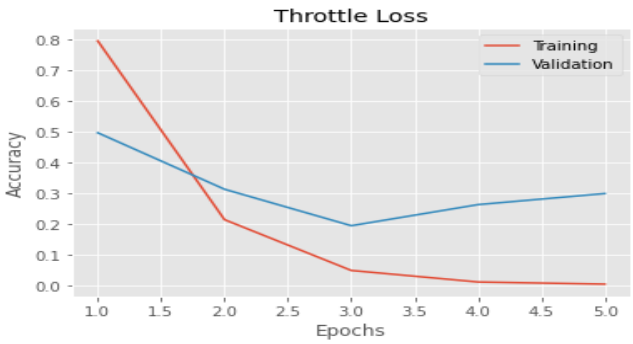

Fig 4.3.6.2 d) epochs vs accuracy

As seen in the above graphs, the steer accuracy is 95% for training and 85% for validation. Throttle accuracy is almost 100% for training and 93% for validation. Best steer loss obtained is 0.4 and best throttle loss is 0.2. However, even if the accuracy is good, the car was not able to navigate smoothly in the environment with other obstacles, it was taking many abrupt turns and the acceleration was also static.

4.3.7. Integrating Imitation Learning with PPO

In transfer learning, a pre-trained model developed for a task is reused as the starting point for a model on a second task. Sometimes reinforcement learning algorithms can take a very long time to converge and we were facing this same issue in training the PPO based model. Also, the model was only learning from the correct demonstrations and did not learn what wrong actions were. Hence we used the imitation learning model as the baseline model for the PPO training.
The source domain i.e. imitation learning is the place where the prior knowledge comes from, and the target domain i.e. PPO is where the knowledge is transferred to. To use PPO, we segregated our actor model into two separate models: one for throttle and another one for steer. The training time of this model was increased upto 100 epochs. This helped to greatly improve the average reward from around 500/epoch for baseline model to 2000/epoch for the fine tuned model. As a result, our actor is much more stable now and can drive longer without crashing. The training logs are shown below:

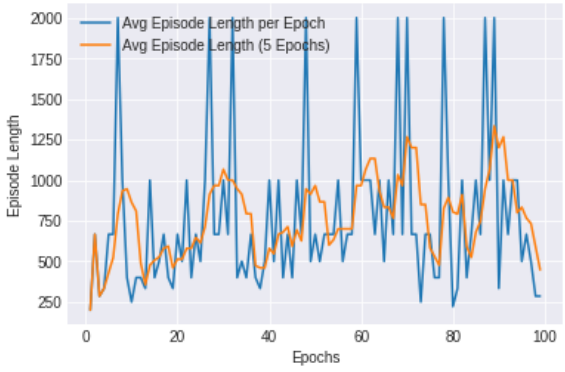

Fig 4.3.7.1 Epochs vs Episode Length

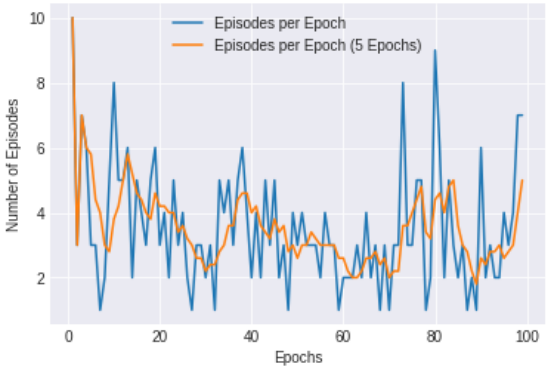

Fig 4.3.7.2 Epochs vs No. of Episodes

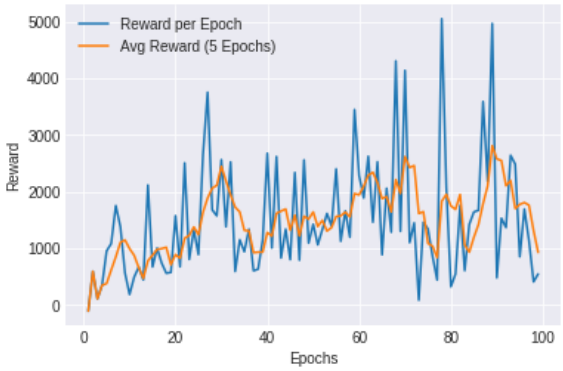

Fig 4.3.7.3 Epochs vs Reward

As seen in Fig 4.3.7.1 and 4.3.7.3 the episode length and the reward gradually increases as the number of epochs increases. The average reward gained over the 100 epochs is around 1500 and the average episode length over 100 epochs is around 700. However if we compare the behaviour of reward and episode length per epoch is quite varied for every epoch whereas the average reward over 5 epochs is more stable. The number of episodes has stayed almost the same over 100 epochs.

4.3.7.1. Dynamic Acceleration

In the initial phases of training, the actor was moving at a constant acceleration i.e. 1.25 when model outputs acceleration as 1 and -1 when model outputs 0. To make it more dynamic, we have used an acceleration moving average of window size 20. If the moving average is smaller, then we have provided higher acceleration at the current step. If the moving average is higher, then we have provided smaller acceleration at the current step.

acceleration(x) = (1-x)*1.75 + 1.25, when the model outputs 1 and x is moving average.

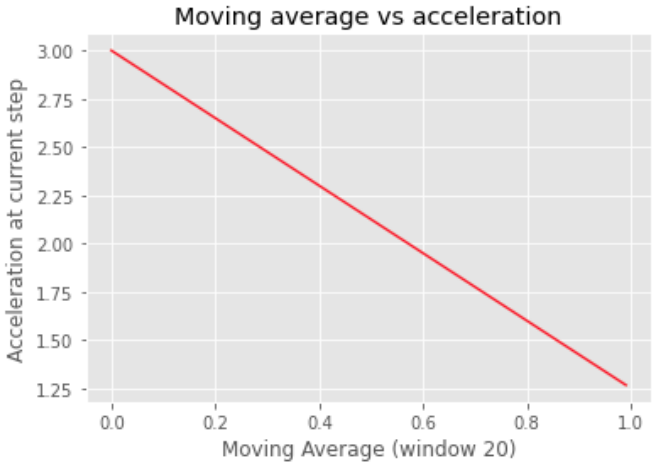

Fig 4.3.7.1 moving average vs acceleration

5. Results and Evaluation

When using the DQN Network, our agent was able to properly navigate the steering on the straight road to avoid bumping on the sides. However, when it reached a roundabout, it was not able to steer the vehicle to take the turn even after training for around 1000 episodes. During training, we also observed that in some episodes, the agent would earn only negative rewards and would not learn to maximize the cumulative reward at all.

We then tried to implement the Dueling Architecture with PER. The duelling architecture can learn which states are valuable and which are not without having to learn the effect of each action for each state. The main benefit that this architecture provided us is that it generalized across the learning actions and led to a better policy evaluation. Our agent was not only able to navigate properly on straight roads but also managed to take a few left and right turns, but eventually ended up crashing on the sides. However, policy overfits to take straight steer action after a period of time and with decreasing epsilon it fails to learn new actions. Hence after around 280 episodes, the cumulative reward becomes drastically negative.

As a result, our next step was to try the proximal policy optimization approach as we were aware the PPO could provide us the benefit of trust region policy optimization. PPO adds a soft constraint that can be optimized by a first-order optimizer. We may make some bad decisions once a while but it strikes a good balance on the speed of the optimization. This balance has helped us to get better results as compared to the other approaches. As we can see in the fig. 5.2 initially the reward increases very slightly, but after training the model for around 100 episodes the cumulative reward gain increases exponentially. In fig. 5.3 the episode length increases consistently after training for 100 episodes.

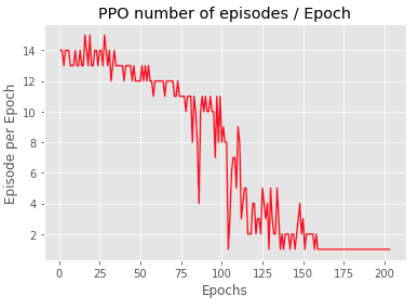

Fig. 5.1 Epochs vs Episode per Epoch

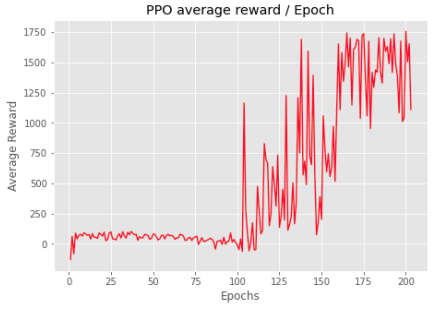

Fig. 5.2 Epochs vs Avg Reward

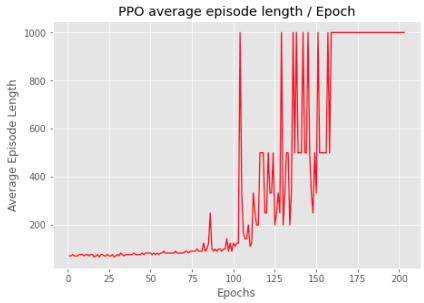

Fig. 5.3. Epochs vs Avg. Episode Length

We evaluated the three approaches by computing the average reward and average episode length for different numbers of episodes. Following are the results obtained after training the model on three different methods for 1000 episodes:

	Average Reward	Average Episode Length
DQN	-13.04	50.99
DDQN with PER	-11.02	60.89
PPO	1356.63	1000
Imitation Learning	787.07	578
Imitation Learning with PPO	1661.46	808.12

Fig. 5.4 Comparison of three different approaches for 1000 episodes

After getting significantly better results with PPO, the agent was then trained using imitation learning to drive in a dynamic environment without colliding with other actors. We also introduced one more action i.e brake/accelerate. We increased the speed of our actor by 25%, so that it can catch up with other drivers. The training metrics of the imitation learning model can be seen in Fig 5.5. We can see in Fig. 5.4 how they increase with the number of epochs and after around 80 epochs, we got the best performing model. We also evaluated the two approaches used for avoiding collision with other actors by computing the average reward and average episode length for different numbers of episodes.

Action	Training	Validation
Steer Accuracy	0.97	0.86
Throttle Accuracy	1.00	0.93
Steer Loss	0.07	0.4
Throttle Loss	0.01	0.2

Fig. 5.5 Imitation learning training logs results

6. Conclusion

The main objective of the project was to train an autonomous driving agent in a simulation environment to follow the path and take appropriate paths. This objective was achieved through a variety of state of the art reinforcement learning methods that allow control in different environments. In this work, we presented a reinforcement-learning based approach with Deep Q Network implemented in autonomous driving. We defined the reward function and agent (car) actions and trained the neural network accordingly to maximise the positive reward. Since the performance of the DQN was not fulfilling the project requirements, we implemented the duelling architecture with PER where the agent was able to take a few turns but still failed to properly navigate through the environment. We have then introduced proximal policy optimization(PPO) where the simulation tests showed that the trained neural network was capable of driving the car autonomously by taking most of the turns. We used imitation learning to avoid collision with the other actors by introducing brake/acceleration as an action whenever another actor stops/accelerates in the lane followed by our model. We then performed transfer learning to integrate imitation learning results with the existing PPO model we had already developed which increased the average rewards by 18%. One limitation on the model is that we have currently trained the model only using bird-eye view and bird-eye view is not available in real-life. Hence, even though the agent performs well in the game, it will not perform well in real-life scenarios. Another limitation on the game is that it doesn't support pedestrians crossing the road.

7. Future Work

As part of our future work, we can train the model using front-view instead of the currently used bird eye view so that it can perform well in scenarios like following traffic light or stop boards. Also, we can modify the game engine to support pedestrians crossing the road. Use computer vision techniques like Mask R-CNN and YOLO v3 for object detection when training the model using front view.

8. References

[1] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, Oleg Klimov, "Proximal Policy Optimization Algorithms" , Aug 2017, https://arxiv.org/pdf/1707.06347.pdf

[2] Peng, B., Sun, Q., Li, S.E. et al. "End-to-End Autonomous Driving Through Dueling Double Deep Q-Network." Automot. Innov. 4, 328–337 (2021), https://link.springer.com/content/pdf/10.1007/s42154-021-00151-3.pdf

[3] Dosovitskiy, Alexey, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. "CARLA: An open urban driving simulator.", https://arxiv.org/abs/1711.03938

[4] Wong, I. K. (2021). "A Study of Proximal Policy Optimization (PPO) Algorithm in Carla (OAPS))." Retrieved from University of Macau, http://oaps.umac.mo/handle/10692.1/247

[5] Li, G., Li, S., Li, S. *et al.* Deep Reinforcement Learning Enabled Decision-Making for Autonomous Driving at Intersections. *Automot. Innov.* 3, 374–385 (2020). https://doi.org/10.1007/s42154-020-00113-1

[6] Wang, Ziyu, et al. "Dueling network architectures for deep reinforcement learning." International conference on machine learning. PMLR, 2016, https://arxiv.org/abs/1511.06581

[7] A. Amini, W. Schwarting, G. Rosman, B. Araki, S. Karaman and D. Rus, "Variational Autoencoder for End-to-End Control of Autonomous Driving with Novelty Detection and Training De-biasing" 2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 2018,

https://dspace.mit.edu/handle/1721.1/118139
[8] Aria, M. “A Survey of Self-driving Urban Vehicles Development.” In IOP Conference Series: Materials Science and Engineering, vol. 662, no. 4, p. 042006. IOP Publishing, 2019 https://iopscience.iop.org/article/10.1088/1757-899X/662/4/042006
[9] Kiran, B. Ravi, Ibrahim Sobh, Victor Talpaert, Patrick Mannion, Ahmad A. Al Sallab, Senthil Yogamani, and Patrick Perez. “Deep reinforcement learning for autonomous driving: A survey.” IEEE Transactions on Intelligent Transportation Systems (2021), https://arxiv.org/abs/2002.00444
[10] Vergara, Marcus Loo. “Accelerating Training of Deep Reinforcement Learning-based Autonomous Driving Agents Through Comparative Study of Agent and Environment Designs.” Master’s thesis, NTNU, 2019, https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/2625841
[11] Zhu, Zeyu, and Huijing Zhao. “A Survey of Deep RL and IL for Autonomous Driving Policy Learning.”, https://arxiv.org/abs/2101.01993
[12] Matt Vitelli, Aran Nayebi, “CARMA: A deep reinforcement learning approach to Autonomous Driving”, https://web.stanford.edu/~anayebi/projects/CS_239_Final_Project_Writeup.pdf
[13] L. Qin, Z. Huang, C. Zhang, H. Guo, M. Ang and D. Rus, "Deep Imitation Learning for Autonomous Navigation in Dynamic Pedestrian Environments," *2021 IEEE International Conference on Robotics and Automation(ICRA)*,2021
[14] Jinyun Zhou, Rui Wang, Xu Liu, Yifei Jiang, Shu Jiang, Jiaming Tao, Jinghao Miao, Shiyu Song, “Exploring Imitation Learning for Autonomous Driving with Feedback Synthesizer and Differentiable Rasterization”, https://arxiv.org/abs/2103.01882
[15] P. M. Kebria, A. Khosravi, S. M. Salaken and S. Nahavandi, "Deep imitation learning for autonomous vehicles based on convolutional neural networks," in *IEEE/CAA Journal of Automatica Sinica,* https://ieeexplore.ieee.org/document/8945486
[16] Bharathan Balaji, Sunil Mallya, Sahika Genc, Saurabh Gupta, Leo Dirac, Vineet Khare, Gourav Roy, Tao Sun, Yunzhe Tao, Brian Townsend, Eddie Calleja, Sunil Muralidhara, Dhanasekar Karuppasamy, “DeepRacer: Educational Autonomous Racing Platform for Experimentation with Sim2Real Reinforcement Learning”. https://arxiv.org/abs/1911.01562