# Autonomous Vehicle Navigation Using Reinforcement Learning Techniques

Aditi Jain
*University Of Southern California*
amjain@usc.edu

Aditya Jain
*University Of Southern California*
jainadit@usc.edu

Adwaita Jadhav
*University Of Southern California*
ajjadhav@usc.edu

Devanshi Desai
*University Of Southern California*
dhdesai@usc.edu

Harshita Bhorshetti
*University Of Southern California*
bhorshet@usc.edu

Isha Chaudhari
*University Of Southern California*
ichaudha@usc.edu

Saichand Duggirala
*University Of Southern California*
sduggira@usc.edu

***Abstract*—Over the last few years, autonomous driving has attracted the interest of researchers, institutes, and recent private companies. Starting with the introduction of a radio controlled car called ”Linriccan Wonder”, to developing self-driving cars, the research in this field seems to have come a long way, while still holding a lot of potential to develop useful products for the future. The usage of this technology is also visible in the gaming industry where various games demonstrating autonomous driving such as Formula-1, Grand Theft Auto (GTA) etc are rapidly developed and seems to provide the virtual platform to test various approaches for the targeted ”driver-less” real world. Recent advancements in machine learning, deep learning and reinforcement learning have helped in driving autonomous navigation technology forward. The main objective of our thesis is to comprehensively analyze current methods in reinforcement learning for training autonomous vehicle navigation. Our major contribution can be seen in the form of an agent which learns utilizing algorithms like Dueling DDQN with PER and Proximal Policy Optimization (PPO) to drive in the urban driving simulator, CARLA. Furthermore, we will present the results of our experiments using variational autoencoder and cropped observation as a preprocessing step for training to enhance performance of our agent. For Carla environment, we will provide a PPO based model that can autonomously drive on a predefined trajectory.**

***Keywords – Reinforcement Learning (RL), Deep Q Network (DQN), Dueling Deep Q Network with Prioritized Experience Replay (DDQN with PER), Proximal Policy Optimization (PPO)***

## I. Introduction

An autonomous vehicle is a technological advancement to traditional vehicles where a vehicle can travel to its destination without human intervention. In order to achieve this, the system needs to be able to properly navigate in the surroundings by avoiding obstacles and re-routing its path whenever needed. Autonomous vehicles help mitigate the risks caused due to weather conditions such as risk of accidents, long driving hours, rash driving amongst others. Additionally, it helps in relieving the driver of mundane tasks like parking. Many Companies like Tesla, Nvidia, Waymo by Google are investing a lot of time and money in this effort where the main aim is to create a risk free and efficient driverless car.

Research in autonomous driving vehicles however, present some of the major challenges like reliability, safety, cost. With the recent advancements in state-of-the-art software and sensing, the reliability and safety has achieved a level befitting its implementation and trials in the real world. Various machine learning and deep learning methods have been proposed and tried, but couldn’t produce effective results. The research is now oriented towards reinforcement learning methods by using past experiences based on state, action and rewards. The vehicle, also called the Reinforcement Learning(RL) agent, learns to take correct actions by learning from the rewards it receives.

With such new methods constantly being developed, there comes a need to be able to test these without much investment. Simulation platforms thus, have become important to test such systems and a platform that can calculate the efficiency in terms of rewards or points becomes an easy choice. Testing various such methodologies in a gaming environment not only allows to test the reliability but also helps compare the various methods in terms of the agent that performed better.

Deploying an RL agent in the game environment where it is allowed to explore, rewarding the agent for correct actions and punishing it for taking wrong ones can help the train the RL agent. Our aim is to develop a model for autonomous vehicle navigation where the car could perform correct actions such as taking proper turns and avoid crashing with static and dynamic obstacles. We have tried to implement this game by training the autonomous vehicle in CARLA environment using various advanced techniques related to Reinforcement Learning such as DQN, Double duelling with PER, PPO and Imitation Learning with and without PPO.

## II. Related Work

Research in the field of autonomous driving has been done for a long time from around 1900. The foremost idea was introduced by General Motors. However the research has evolved from being most advanced in recent areas with the rise of Artificial Intelligence, and various ways and technologies involving Computer Vision, Light Detection And Ranging (LiDAR) etc.

The research done around 1989 by Pomerleau, was one of the earliest attempts of using Neural Networks for Autonomous Driving. Typically he built "Autonomous Land Vehicle in Neural Network" (ALVINN). Various images were given as input to the neural networks and depending on the pixel along with various car driving scenes the network predicted different actions. The research grabbed the attention of numerous scientists as it proved the potential of Neural Networks specifically in the domain of Autonomous driving.

Mariusz Bojarski et al. [13] trained a convolutional neural network (CNN) to map raw pixels from a single front-facing camera directly to steering commands. This end-to-end approach proved surprisingly powerful. With minimum training data from humans the system learns to drive in traffic on local roads with or without lane markings and on highways. It also operates in areas with unclear visual guidance such as in parking lots and on unpaved roads.

John Schulman et al. [1] used Deep Q-Network i.e. DQN to perform the task of autonomous car driving using information from various sensory inputs. To implement the DQN, John Schulman et al. have 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. John Schulman et al. 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.

Another attempt made in the area of autonomous driving was by Comma.ai [2] who tried to state the problem in terms of videos and their frames. The problem was framed to predict the next frame of the video.They implemented a Variational Autoencoder along with Generative Adversarial Network (GAN) . The implementation of variational autoencoders along with previous cost functions using GAN for developing embeddings of various road frames. The main idea was to predict the next frame and continue doing so till it looks quite realistic. Later a transition model was learned in the embedded space using various action conditioned Recurrent Neural Networks. The model could continuously predict realistic looking video for several frames and thus eventually the next actions to be performed specifically the next steering actions.

Many authors have worked on using Deep Imitation Learning for Autonomous Vehicles Based on Convolutional Neural Networks. Lei Qin et al. [11] 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 were developed in this manner. On the other hand, Jinyun Zhou et al. [12] focused 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 = [$x_t$, $y_t$, $\phi_t$, $v_t$] T , vehicle length l, and width w, our rasterization function G rasterizes images as:

$$G_{i,j}(s_t) = \max_{k=1,2,3}(N(\mu^{(k)}, \Sigma^{(k)})) \tag{1}$$

where each cell (i, j) in G is denoted as $G_{i,j}$ . 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.

## III. Data and Environments

We define the following elements of the system:

### A. 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. For a more realistic driving environment, we are using an open source simulator - Carla [5], featuring common urban driving scenarios. It is especially 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.

Interacting with the 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.

### *B. 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 the system also provides other actors like pedestrians, traffic signs, traffic lights.

### *C. 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.

### *D. Action*

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

- Left turn
- Right turn
- Go straight

### *E. 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.

## IV. Methods

### *A. Reinforcement Learning*

In reinforcement learning, the autonomous agent learns to improve its performance by obtaining rewards on performing desired behaviours and punishing the agent for undesired ones. The feedback in terms of positive and negative is numerically given to the agent as rewards, which in turn helps the agent to learn on its own without needing an expert. For every state, the agent chooses an action and receives the reward value generated by a reward function that helps it to determine how useful the decision of taking that action was. The goal of the agent is to maximize the cumulative rewards achieved over its lifetime. Hence, the agent can learn from the previously learnt knowledge about the expected utility(i.e. discounted sum of expected future rewards) of the various state-action pairs. This helps the agent in increasing its long term reward.

One of the major challenges while designing an RL agent is the ability to strike a balance between the exploration and exploitation. In order to achieve the maximum cumulative reward, the agent must utilize its knowledge to choose actions that produce maximum rewards. This means that the agent is limited to choosing the optimal action and hence would not be able to explore any new actions. On the other hand, in order to discover such beneficial actions, it has to take the risk of trying new actions which may lead to higher rewards than the current best-valued actions for each system state. To balance this trade-off, various $\epsilon$-greedy and softmax approaches have been proposed. With the $\epsilon$-greedy strategy, the agent has the option of either selecting an action at random with probability $0 << 1$, or greedily select the highest valued action for the current state with the remaining probability $1\epsilon$. We have also used varied levels of exploration, with more exploration being done initially when little is known about the problem environment. As the training progresses, the agent gradually conducts more exploitation than exploration.

In many real-world application domains, it is not possible for an agent to observe all features of the environment state. In such cases the decision-making problem is formulated as a Partially-Observable Markov Decision Process (POMDP). Solving a reinforcement learning task means finding a policy $\pi$ that maximises the expected discounted sum of rewards over trajectories in the state space. An RL agent can learn value function estimates, policies and/or environment models directly. This forms the basis of our RL agent implementation.

### *B. Deep Q Network*

Reinforcement learning works good 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, the car agent has to learn continuous states and continuous variables, rather than discrete ones as used in the Deep RL networks [10]. Deep Q Networks or DQNs allows the agent to acquire knowledge by exploring any unstructured environment over a time.

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 states' by performing some action $a$ and receiving a reward $r$. We restricted the definition to include only one action: to steer in discrete space with values as $-1, 0, 1$. We also used the epsilon greedy strategy by starting with $\epsilon = 1$ and decreasing by 0.99 for each episode.

The TD error was calculated using the formula in Equation 2:

$$Q(S,A) = Q(S,A) + \alpha(R + \gamma \max_{a'} Q(S', A') - Q(S,A)) \quad (2)$$

Where:

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

## C. Dueling Architecture with PER

The dueling architecture was presented by Wang et al [8]. A Dueling Deep Q-Network was also presented by Peng. B. Syn et al. [3] that helps in making the agent learn about lane keeping. In scenarios like self-driving cars, it is not needed that we know the value associated with each action at every timestamp. Thus we can modify the architecture by dividing the representation of state values and state dependent actions into two sub networks. By separating into 2 separate estimators, the dueling 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 in Equation 3 and 4, respectively:

$$Q^\pi(s,a) = E[R_t | s_t = sa_t = a, \pi] \quad (3)$$

$$V^\pi(s,a) = E_{a\ \pi(s)}[Q^\pi(s,a)] \quad (4)$$

The same Q function could be written as in Equation 5:

$$Q^p i(s,a) = E[r + [Q^\pi(s',a')] | s_t = s, s_t = s, \pi] \quad (5)$$

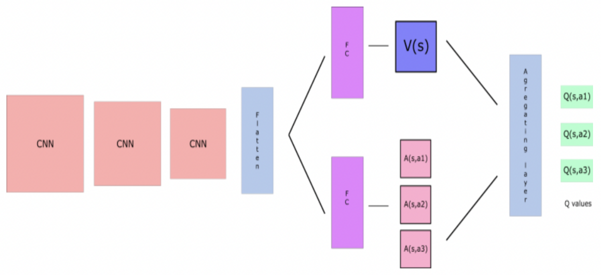

Fig. 1. Dueling Architecture

The advantage can be calculated by subtracting Q-value by V value as in Equation 6:

$$A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s) \quad (6)$$

Consider that the specific action at a given state is shown by Q value. The advantage can be calculated by subtracting Q value from V value. The given formula can, thus, be interpreted as:

- V value is the value of a state irrespective of an action taken.
- Advantage depicts how beneficial choosing a particular action is compared to other states.

*1) Architecture:* Similar to a DQN, we have convolutional layers which take input as game frames. We divide the final layer into 2 parts for finding state value and find corresponding states advantages. The final layer merges the advantages and state value.

The final output is a combination of a given state's value and states advantage with normalization in Equation 7:

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + A(s,a;\theta,\beta) \quad (7)$$

Issues associated with this method are as follows:

1) Simply adding two values in order to estimate the state-value and action advantages is a naive approach, and can lead to issues.
2) Simply the sum is "unidentifiable," with just given Q value its not possible to retrieve the V and A. This inability to retrieve V and A leads to performance issues.

The identifiability issue can be solved, by doing a forward mapping which forces Qvalue to maximize action to equal V as in Equation 8:

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + (A(s,a;\theta,\alpha) - \max_{a' \in |A|} A(s,a;\theta,\alpha)) \quad (8)$$

The training error used is the TD error. The training process is exactly the same as the vanilla DQN architecture. We define the loss as mean squared error as seen in Equation 9.

$$L(\theta) = \frac{1}{n} \sum_{i \in N} (Q_\theta(s_i,a_i) - Q'_\theta(s_i,a_i))^2 \quad (9)$$

where $Q_\theta^{'} = R(s_t, a_t) + \gamma \max a_i^{'} Q_\theta(s_i^{'}, a_i^{'})$. (TD Error)

*2) Implementation:* We used the same architecture as DQN except for the change in second last layer which was divided into two parts to calculate values and advantages of a particular action which was later combined to find the next course of action. 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.

### *D. Proximal Policy Optimization*

Proximal Policy Optimization(PPO) [7] is a model-free, policy gradient based reinforcement learning algorithm and can be used for either discrete or continuous action spaces.

PPO [6] 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 in Equation 10:

$$(s_t, a_t, r_t, d_t, V(s_t.\theta_v)) \tag{10}$$

$d_t$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($G_t$) as seen in Equation 11:

$$G_t = R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + ..... = \sum_{k=0}^{\infty} \gamma^k R_{t+k} \tag{11}$$

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 $\hat{A}_t$ uses the more accurate generalized advantage estimation (GAE) calculation instead. After we have obtained a trajectory, we calculate advantage estimates with this GAE equation in 12:

$$\hat{A}_t = \sigma_t + (\gamma.\lambda)\sigma_{t+1} + + (\gamma.\lambda)^{T-t+1}\sigma_{T-1} \tag{12}$$

$\lambda$ is an interpolation factor that serves as a trade-off between bias and variance in the advantage estimates.
$\gamma$ is the reward discount factor.

Similar to the Trust Region Policy Optimization objective function, we define the probability ratio between the new policy and old policy and call it as r($\theta$) as seen in Equation 13.

$$r(\theta) = \frac{\pi_\theta(a|s)}{\pi_{\theta old}(a|s)} \tag{13}$$

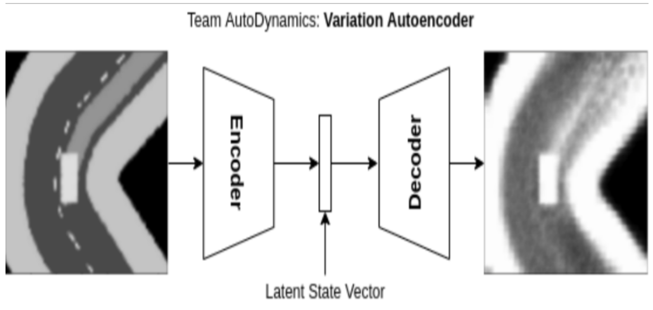

Fig. 2. Variational Autoencoder

The actor network is updated by an Adam optimizer with a learning rate of $3e^{-4}$. For each optimization step, we updated the policy by maximizing the PPO-Clip objective defined in Equation 14:

$$\theta_{k+1} = \arg\max_\theta \frac{1}{|\mathcal{D}_k|T} \sum_{\tau\in\mathcal{D}_k} \sum_{t=0}^{T} \min\left( \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t),\ \ g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t)) \right), \tag{14}$$

The critic network is updated by Adam optimizer with a learning rate of $1e^{-3}$. We use the following mean-squared error to optimize the critic network as seen in Equation 15:

$$\phi_{k+1} = \arg\min_\phi \frac{1}{|\mathcal{D}_k|T} \sum_{\tau\in\mathcal{D}_k} \sum_{t=0}^{T} \left( V_\phi(s_t) - \hat{R}_t \right)^2 \tag{15}$$

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:
First approach is variational autoencoder [9] to reduce state dimension. It compressing 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 : As shown in Fig 2. The encoding network turns input samples into 2 parameters in latent space - $\mu$ 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. In equation 10, $s_t$ corresponds to the latent space vector z produced by the encoder.
- Decoding Method : As shown in Fig 2. 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

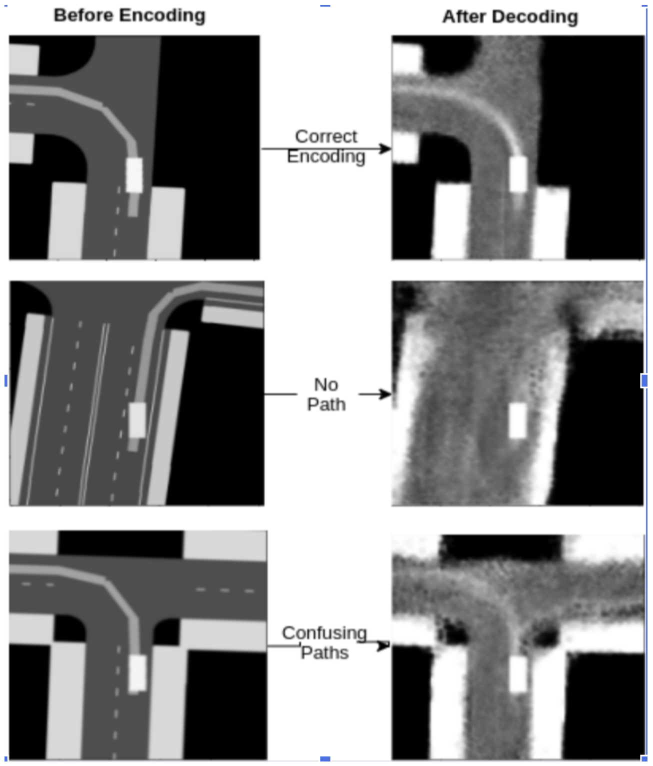

Fig. 3. Encoding Decoding Process

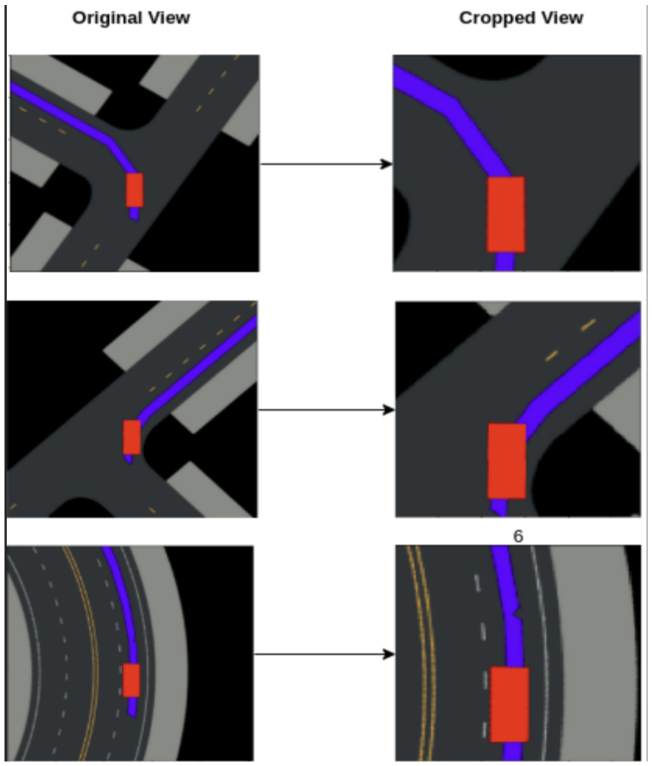

Fig. 4. Cropped View in CARLA

is based on binary cross entropy and forces the decoded samples to match initial inputs as seen in Equation 16.

$$reconstruction\ loss = \frac{1}{hw}\sum_{i=0}^{h}\sum_{j=0}^{w} -ylog(\hat{y}) \tag{16}$$

The Kullback Leibler divergence also known as relative entropy measures how one probability distribution is different from another. In the VAE model, KL divergence acts as a regularization term between the learned latent distribution and previous distribution as seen in Equation 17.

$$kl\ loss = \sum_{i=1}^{n} \sigma_i^2 + \mu_i^2 - log(\sigma_i) - 1 \tag{17}$$

Challenges faced in approach 1 : The agent was working with training images but was not able to follow policy for new unseen states. So, we were facing over-fitting. Also, the decoding was not perfect for some rare observations. As seen in Fig 3, there were times when a correct path was not provided by the decoder or when there were multiple turning options, the path to be followed was confusing. Hence we tried the next approach to overcome these problems.

Second approach is cropping the state view to use only interesting areas in image for training. We removed irrelevant areas like pixels behind the car, pixels on the footpath and pixels far ahead of car from our observations. As seen in Fig 4, we converted 512 x 512 x 3 images into 128 x 128 x 3 images to get cropped observations. 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 number of parameters are to be fit.

## V. Imitation Learning

Imitation Learning algorithms [14] 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.

In order to train the model using imitation learning, we introduced other actors like vehicles and pedestrians in the environment that required us to introduce one more action i.e. brake and accelerate. There are 6 possible combinations of actions that the agent can take:

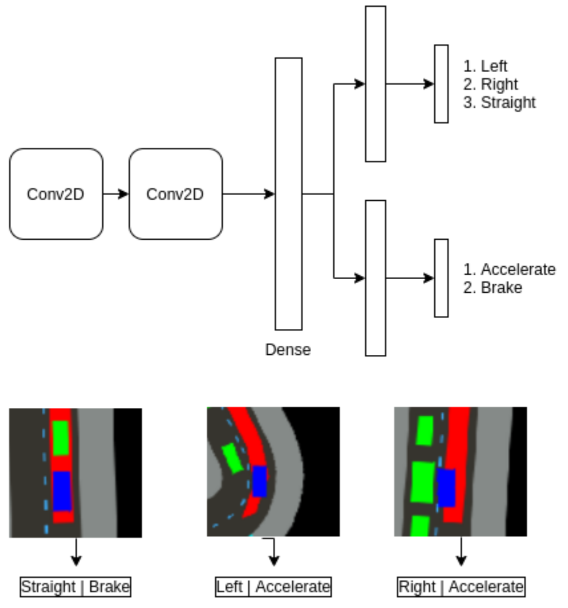

Fig. 5. Model and Training Configuration

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

We used the same training approach we used while developing the previous models: 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. 5 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.

## VI. 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. Hence we used the imitation learning model as the baseline model for the PPO training. 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:

The episode length and the reward that our agent receives 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.

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. The equation is defined as seen in Equation 18:

$$acceleration(x) = (1 - x) * 1.75 + 1.25 \tag{18}$$

where x is the moving average.

## VII. Results and Analysis

To begin with, the agent was trained on DQN Network to properly navigate the steering of the vehicle on straight road to avoid bumping on the sides. However, the agent was not able to steer the vehicle to take an accurate turn on roundabouts, even after training for 1000 episodes. To overcome this, the agent was implemented using Dueling Architecture with PER. The agent generalized across the learning actions and led to a better policy evaluation. We used the values of hyperparameter for training as seen in Table I:

We used the epsilon greedy strategy, in which we used epsilon 1 initially and then decreased by 0.995 after each episode. The agent not only able to navigate properly on straight roads but also managed to take a few left and right turns. However, due to policy overfitting, the agent starts taking straight steer action after a period of time and with decreasing epsilon it fails to learn new actions as seen in Fig.

TABLE I
HYPERPARAMETER FOR DDQN WITH PER

| Hyperparameter | Value |
|---|---|
| Minimum Replay Memory Size | 500 |
| Training Batch Size | 32 |
| Discount | 0.99 |
| Maximum episode length | 2000 |
| Epsilon | 1 |
| Epsilon Decay | 0.99 |
| Minimum Epsilon | 0.001 |
| Minimum Rewards | -200 |
| Replay Memory | 3000 |
| Episodes | 200 |

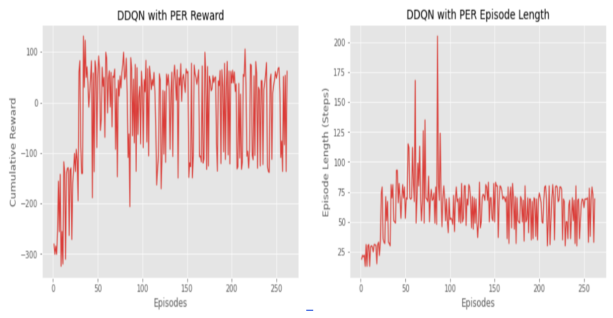

Fig. 6. DDQN with PER

5. Hence after around 280 episodes, the cumulative reward becomes drastically negative and the car crashes with the sides.

As a result, our next step was to try the proximal policy optimization approach since PPO provides the benefit of trust region policy optimization. PPO adds a soft constraint that can be optimized by a first-order optimizer. Hyperparameters used by PPO can be seen in Table II:

TABLE II
HYPERPARAMETER FOR PPO

| Hyperparameter | Value |
|---|---|
| Steps per Epoch | 1000 |
| Maximum Episode Steps | 2000 |
| Epochs | 200 |
| Gamma | 0.99 |
| Clip Ratio | 0.2 |
| Policy Learning Rate | $3e-4$ |
| Value Function Learning Rate | $1e-3$ |
| Train Policy Iterations | 20 |
| Train Value Iterations | 20 |

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. 8 initially the reward increases very slightly, but after training the model for around 100 episodes the cumulative reward gain increases exponentially. In fig. 7 the episode length increases consistently after training for 100 episodes.

We evaluated the three approaches by computing the average reward and average episode length for different numbers

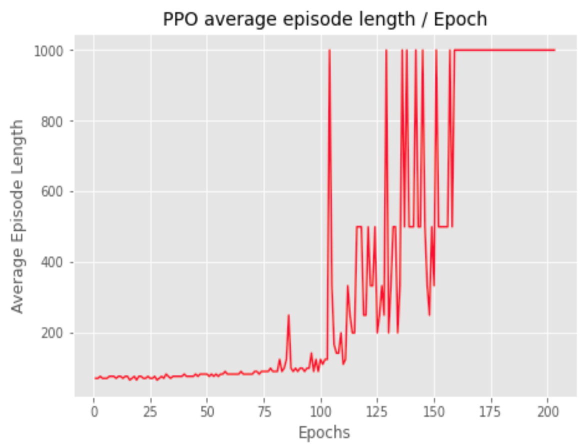

Fig. 7. PPO - Average Episode Length per epoch

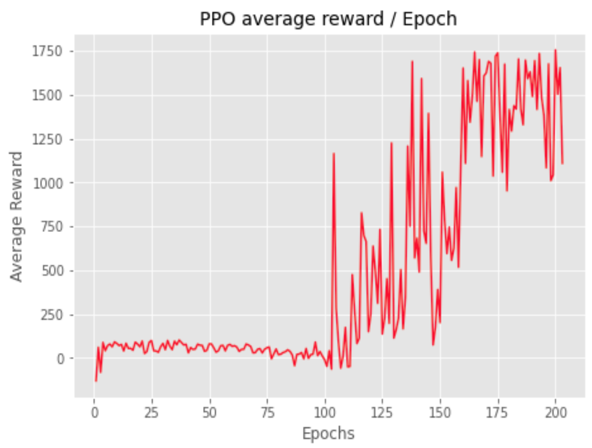

Fig. 8. PPO - Average Reward per epoch

of episodes. The results obtained after training the model on the three methods for 1000 episodes are depicted in table III.

TABLE III
RESULTS

| Method | Average Reward | Average Episode Length |
|---|---|---|
| DQN | -13.04 | 50.99 |
| DDQN with PER | -11.02 | 60.89 |
| PPO | 1356.63 | 1000 |

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 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 table IV.

To use PPO, we segregated our actor model into 2 separate model, one for throttle and another one for steer. The epochs required during fine-tuning of imitation learning model were only 100 as compared to 200 which was previously required by

TABLE IV
TRAINING METRICS

| 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 |

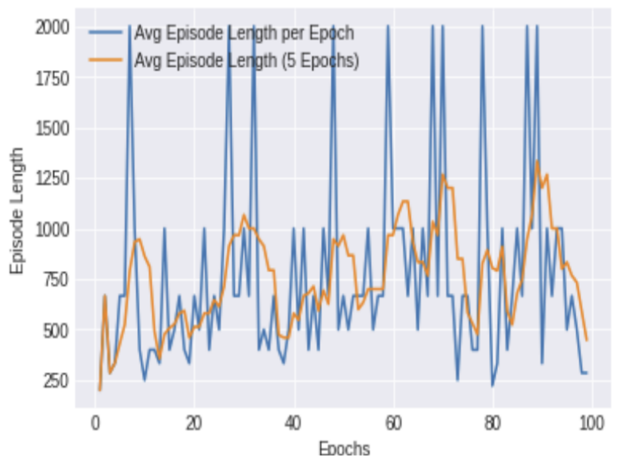

Fig. 9. PPO with Imitation - Average Episode Length per epoch

PPO. We were able to improve average reward from around 500/epoch for baseline model to around to 2000/epoch for fine-tuned model as seen in Fig. 9.

As far as the rewards are concerned, we can see in Fig. 10 how they increase with 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. The results obtained after training the model on the two methods are depicted in table V.

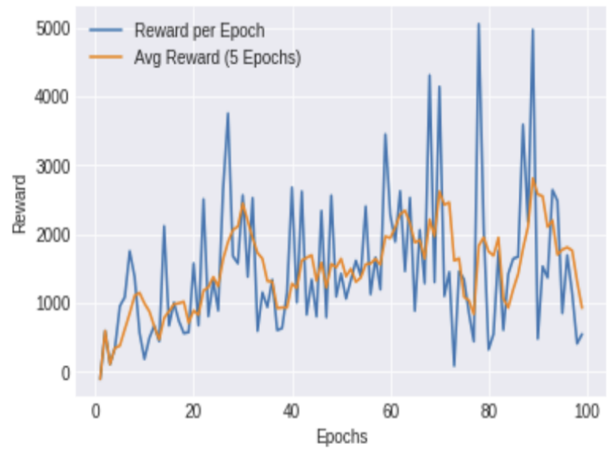

Fig. 10. PPO with Imitation - Average Rewards

TABLE V
RESULTS WITH OBSTACLES

| Method | Average Reward | Avg Episode Length |
|---|---|---|
| Imitation Learning | 787.07 | 578 |
| PPO with Imitation Learning | 1661.46 | 808.12 |

## VIII. LIMITATIONS, CONCLUSION AND FUTURE WORK

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. After getting satisfactory results using PPO, we fine-tuned the model to run in a dynamic environment with multiple actors. 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 PPO 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 pedestrian crossing the road.
As part of our future work, we can train the model using front-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. We can also use computer vision techniques like YOLOv3 to perform image recognition while training using front view.

## REFERENCES

[1] 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.
[2] Santana, Eder, and George Hotz. "Learning a driving simulator." arXiv preprint arXiv:1608.01230 (2016).
[3] 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).

[4] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, Vladlen Koltun Proceedings of the 1st Annual Conference on Robot Learning, PMLR 78:1-16, 2017.

[5] Dosovitskiy, Alexey, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. "CARLA: An open urban driving simulator." In Conference on robot learning, pp. 1-16. PMLR, 2017.

[6] Wong, I. K. (2021). A Study of Proximal Policy Optimization (PPO) Algorithm in Carla (OAPS)). Retrieved from University of Macau, Outstanding Academic Papers by Students Repository.

[7] Schulman J, Wolski F, Dhariwal P, Radford A, Klimov O. Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347. 2017 Jul 20.

[8] Wang, Ziyu, et al. "Dueling network architectures for deep reinforcement learning." International conference on machine learning. PMLR, 2016.

[9] 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, pp. 568-575, doi: 10.1109/IROS.2018.8594386.

[10] Kiran, B. Ravi, Ibrahim Sobh, Victor Talpaert, Patrick Mannion, Ahmad A. Al Sallab, Senthil Yogamani, and Patrick Pérez. "Deep reinforcement learning for autonomous driving: A survey." IEEE Transactions on Intelligent Transportation Systems (2021).

[11] 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

[12] 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

[13] Bojarski, Mariusz, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel et al. "End to end learning for self-driving cars." arXiv preprint arXiv:1604.07316 (2016).

[14] Zhu, Zeyu, and Huijing Zhao. "A Survey of Deep RL and IL for Autonomous Driving Policy Learning." arXiv preprint arXiv:2101.01993 (2021).