Autonomous Driving using Deep Reinforcement Learning on MetaDrive

Puru Upadhyay, Shreyansh Goyal
e-mail: pupadhyay@wpi.edu, sgoyal2@wpi.edu

Abstract – **Safe-Driving requires the knowledge of the environment around and taking the actions accordingly. Autonomous driving can be considered reliable when it is able to do so in the real-time. In this paper, we have implemented value based and policy gradient based Reinforcemnt Learning algorithms for making a vehicle learn to drive on road using discrete and continuous actions space. The Metadrive vehicle simulator is used to implement RL algorithms like DQN, Double DQN, Dueling DQN which are values based reinforcement learning algorithms for an optimal policy to drive on road. PPO, a gradient based method is used for discrete as well as continuous action space learning.**

Keywords – **DQN, PPO, MetaDrive**

NOMENCLATURE

DQN	Deep Q-Learning
PPO	Proximal Policy Optimization

I. INTRODUCTION

Significant progress has been made in reinforcement learning (RL), ranging from super-human Go[1] playing to delicate dexterous in-hand manipulation. Recent progress shows the promise to apply Reinforcement Learning to real-world applications, such as autonomous driving. Autonomous driving has garnered the interest of researchers, governments, and private companies as of late, as such technologies promise to solve several problems that are prominent in modern society e.g. wasting a lot of time in the traffic or getting into road-accident. Reinforcement learning has gained a lot of attention in the field of self-driving lately due to its success in playing video games and beating humans by a high margin. With the development of deep representation learning, the domain of reinforcement learning (RL) has become a powerful learning framework now capable of learning complex policies in high dimensional environments. Autonomous driving systems constitute of multiple perception level tasks that have now achieved high precision on account of deep learning architectures.Besides the perception, autonomous driving systems constitute of multiple tasks where classical supervised learning methods are no more applicable.

Q-Learning algorithm creates an exact matrix for the agent to maximize its reward in the long run.The goal of Q-Learning is to maximize the Q-value trough iteration policy, which tuns a loop between policy evaluation and policy improvement. But if the number of states increases the Q-matrix dramatically increases and it becomes difficult to manage.Deep Q-Learning manages this issue by introducing a Neural Network. So, DQN estimates Q-values by using it in a learning process, where the state is the input of the Net, and the output is the corresponding Q-value for each action.

However, DQN is applied for the discrete state space and therefore, is not considered best for the tasks like autonomous driving. On the other hand, policy gradient method is applied for the continuous state space and fits better for the self-driving task. Proximal Policy Optimization (PPO) by Schulman 2017 at OpenAI is currently considered the best baseline for reinforcement learning research. PPO is a policy gradient is an optimized policy gradient method which can be used for environments that have either discrete or continuous action space. The method utilizes the Actor critic method in which the actor is responsible for observing the environment and choosing an optimal action and the critic gives an expectation of the rewards that the agent can get for the given observation.It has much better convergence properties than previous reinforcement learning approaches, due to a clever combination of clipping the policy loss, and calculating the loss in terms of a probability ratio instead of optimizing the policy's log likelihood directly. In this paper we implement DQN and it's variants and PPO and compare the results of their behaviour.

II. RELATED WORK

Autonomous driving is one of the most researched field of application of RL as it allows to solve for the complex policies. In the RL paradigm an autonomous agent learns to improve its performance at an assigned task by interacting with its environment. Russel and Norvig[2] define an agent as "anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators" RL gained attention after it's success in beating humans by a significant scores in the complex games[3]. Kuutti et al.[4] discussed DL methods for controlling AVs and its promising performance in complex scenarios. Authors presented the strengths and limitations of existing DL methods applied to autonomous vehicle control. Huang et al.[5] proposed longitudinal control of autonomous land vehicles using parametric reinforcement learning. This approach used the parameterized batch actor-critic algorithm to get optimal control policies which adaptively tune the fuel control signals for tracking speeds. Nevertheless, they did not give some simulating criteria to evaluate the effectiveness of autonomous driving.

Min et al.[6] defined highways driving policy using the reinforcement learning method. They also proposed a supervisor agent using deep distributional reinforcement learning to enhance the driver assistant systems. The supervisor agent is trained using end-to-end approach that directly maps both a camera image and LIDAR data into action plan. Hitesh Arora [7] explore the use of an off-policy RL algorithm, Deep Q-Learning, to learn goal-directed navigation in a simulated urban driving environment and demonstrate promising results on the CoRL2017 and NoCrash benchmarks on CARLA. Oleguer Canal and Federico Taschin[8] tackle the completion of an obstacle maze in Unity by a self-driving vehicle where a fully self-trained agent using PPO (Proximal Policy Optimization) controls the vehicle and show that the agent can learn to follow a path that does not take into account their dynamics by exploiting Deep Reinforcement Learning. Tanmay Agarwal[9] propose two works that formulate the urban driving tasks using reinforcement learning and learn optimal control policies primarily using waypoints and low-dimensional representations and demonstrate that the agents when trained from scratch learn the tasks of lane-following and driving around intersections as well as learn to stop in front of other actors or traffic lights even in the dense traffic setting.

III. ENVIRONMENT

A. Simulator

MetaDrive is an efficient and compositional driving simulator which has the capability of generating infinite scenes with various road maps and traffic settings for research of generalize RL. It is very easy to install and has the capability to achieve 300 FPS on a standard PC. The simulator has accurate physics with multiple sensory input including Lidar, RGB images, top-down semantic map and first-person view images.

B. Observation Space

MetaDrive provides various kinds of sensory inputs. For low-level sensors, RGB cameras, depth cameras and Lidar can be placed anywhere in the scene with adjustable parameters such as view field and the laser number. Meanwhile, the high-level scene information including the road information and nearby vehicles' information like velocity and heading can also be provided as the observation. Three types of existing Observation are:

- State Vector.
- Top-down Semantic Maps
- First-View Images

C. Action Space

MetaDrive receives normalized action as input to control each target vehicle. At each environmental time step, MetaDrive converts the normalized action into the steering (degree), acceleration (hp) and brake signal (hp). MetaDrive provides the functionality of adding more dimensions in the action space. This allow the user to write environment wrapper that introduce more input action dimensions.

D. Rewards

The default reward function in MetaDrive only contains a dense driving reward and a sparse terminal reward. The dense reward is the longitudinal movement toward destination in Frenet coordinates. The sparse reward +20 is given when the agent arrives the destination. MetaDrive calculates a complex reward function that enables user to customize their reward functions from config dict. The complete reward function is composed of four parts as follows:

$$\mathrm{R} = \mathrm{c}_1 * R_{driving} + C_2 * R_{speed} + R_{termination}$$

The driving reward denote the longitudinal coordinates of the target vehicle in the current lane of two consecutive time steps, providing dense reward to encourage agent to move forward.

Speed reward incentives agent to drive fast. The termination reward contains a set of sparse rewards. At the end of episode, other dense rewards will be disabled and only one sparse reward will be given to the agent at the end of the episode according to its termination state.

It also implements success reward,out of the road penalty and crash vehicle penalty.

IV. IMPLEMENTATION

Generalization remains one of the fundamental challenges in RL for its real-world applications. Even for the common driving task, an agent that has learned to drive in one town often fails to drive in another town. There is the critical issue of model overfitting due to the lack of diversity in the existing RL environments which is why we decided to go with the MetaDrive simulator. MetaDrive is highly compositional, which can generate an infinite number of diverse driving scenarios from both the procedural generation and the real data import which allows generalizability and safe exploration.

In the architecture of the model, the input to the neural network are the 19 dimensional observations which consists of ego states and navigation information. The first hidden layer is the fully connected layer with 256 nodes and takes these higher dimensional inputs and applies rectifier non-linearity. The second fully connected hidden layer outputs 256 nodes and again applies rectifier non-linearity. The output layer is the fully connected linear layer with output for each of the valid action.

A. DQN Family

Firstly, we have implemented DQN and it's improved versions-Double DQN and Dueling DQN for navigation the car in the simulator. The states, actions and rewards for the deep Q-network are defined below.

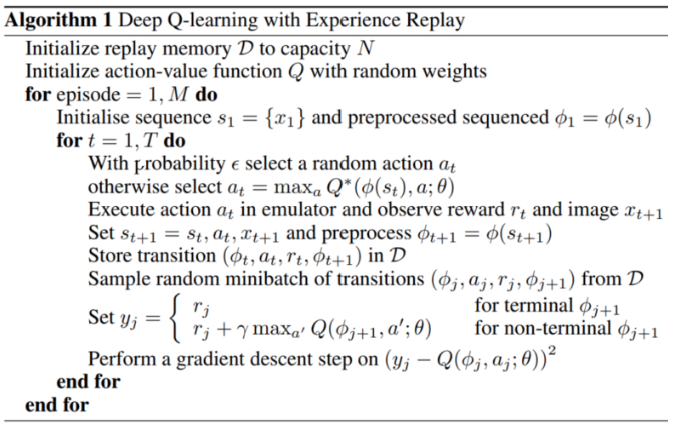
1) States: State space consists of vehicle's state information and Navigation information.Vehicle's state consists of information like distance from the side walk, distance from the left yellow line,current speed, steering etc. While navigation info includes distance between vehicle and checkpoint etc. Total dimension of the state space is 10+9 =19.

2) Action: Action space is discrete and is an array of steering and acceleration.It's dimension is 3x3 and each of the steering and acceleration discrete values are -0.5, 0 and +0.5.

3) Reward: MetaDrive actually prepares a complex reward function that enables user to customize their reward functions from config dict directly. For all the algorithms implemented, rewards used are the same as described above.

4) Algorithm: In this section we will discuss the algorithms and their implementation for the autonomous driving in detail.

5) DQN: Deep Q-learning we take advantage of experience replay, which is when an agent learns from a batch of experience by selecting the small batch of tuple randomly and learn from it using a gradient descent update step.The goal is to update our neural nets weights to reduce the error. Hyperparameters used are batch size= 32,Gamma=0.9, epsilon=0.1, target update frequency=1000 and memory size= 20000.



```
Algorithm 1 Deep Q-learning with Experience Replay
Initialize replay memory $\mathcal{D}$ to capacity $N$
Initialize action-value function $Q$ with random weights
for episode $= 1, M$ do
    Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
    for $t = 1, T$ do
        With probability $\epsilon$ select a random action $a_t$
        otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
        Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
        Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$
    end for
end for
```

6) Double DQN: The max operator in standard Q-learning and DQN uses the same values both to select and to evaluate an action. This makes it more likely to select overestimated values, resulting in overoptimistic value estimates. To prevent this, we can decouple the selection from the evaluation.

$$\text{Double DQN target} = \mathrm{R}_{t+1} + Q(S_{t+1}, argmax_a Q(S_{t+1}, a; \theta); \theta')$$

For the implementation, hyperparameters used are Epsilon = 0.1, Batch size = 64, Gamma = 0.9, Learning rate = 2e-4 and target update frequency= 1000.

```
Algorithm 1: Double DQN Algorithm.
input  : $\mathcal{D}$ – empty replay buffer; $\theta$ – initial network parameters, $\theta^-$ – copy of $\theta$
input  : $N_r$ – replay buffer maximum size; $N_b$ – training batch size; $N^-$ – target network replacement freq.
for episode $e \in \{1, 2, \ldots, M\}$ do
    Initialize frame sequence $\mathbf{x} \leftarrow ()$
    for $t \in \{0, 1, \ldots\}$ do
        Set state $s \leftarrow \mathbf{x}$, sample action $a \sim \pi_{\mathcal{B}}$
        Sample next frame $x^t$ from environment $\mathcal{E}$ given $(s, a)$ and receive reward $r$, and append $x^t$ to $\mathbf{x}$
        if $|\mathbf{x}| > N_f$ then delete oldest frame $x_{t_{min}}$ from $\mathbf{x}$ end
        Set $s' \leftarrow \mathbf{x}$, and add transition tuple $(s, a, r, s')$ to $\mathcal{D}$,
            replacing the oldest tuple if $|\mathcal{D}| \geq N_r$
        Sample a minibatch of $N_b$ tuples $(s, a, r, s') \sim \text{Unif}(\mathcal{D})$
        Construct target values, one for each of the $N_b$ tuples:
        Define $a^{\max}(s'; \theta) = \arg\max_{a'} Q(s', a'; \theta)$
        $y_j = \begin{cases} r & \text{if } s' \text{ is terminal} \\ r + \gamma Q(s', a^{\max}(s'; \theta); \theta^-), & \text{otherwise.} \end{cases}$
        Do a gradient descent step with loss $\|y_j - Q(s, a; \theta)\|^2$
        Replace target parameters $\theta^- \leftarrow \theta$ every $N^-$ steps
    end
end
```

7) Dueling DQN: Dueling network represents two separate estimators: one for the state value function and one for the state-dependent action advantage function. We apply prioritized experience replay here.The idea is that some experiences may be more important than others for our training, but might occur less frequently. We define the probability of sampling transition i as P(i) = $\mathrm{p}_i^\alpha \frac{}{\sum_k p_k^\alpha}$ here Pi is the priority of each transition and exponent α etermines how much prioritization is used. Formula for the decomposition of Q-value:

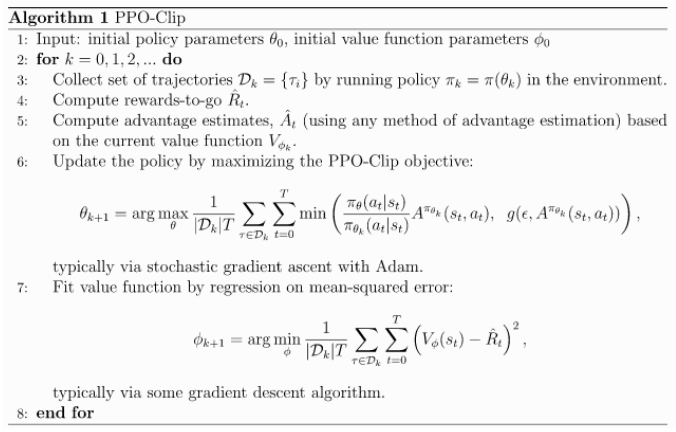
$$\mathrm{Q(s,a;}\theta,\alpha,\beta) = V(s;\theta,\beta) + (A(s,a;\theta,\alpha) - \frac{1}{|\mathscr{A}|}\sum_{a'} A(s,a';\theta,\alpha))$$

For the implementation, hyperparameters used are Epsilon=0.2, Gamma= 0.9, target update frequency= 1000 and learning rate= 2e-4.

```
Algorithm 1 Double DQN with proportional prioritization
 1: Input: minibatch $k$, step-size $\eta$, replay period $K$ and size $N$, exponents $\alpha$ and $\beta$, budget $T$.
 2: Initialize replay memory $\mathcal{H} = \emptyset$, $\Delta = 0$, $p_1 = 1$
 3: Observe $S_0$ and choose $A_0 \sim \pi_\theta(S_0)$
 4: for $t = 1$ to $T$ do
 5:     Observe $S_t, R_t, \gamma_t$
 6:     Store transition $(S_{t-1}, A_{t-1}, R_t, \gamma_t, S_t)$ in $\mathcal{H}$ with maximal priority $p_t = \max_{i<t} p_i$
 7:     if $t \equiv 0 \mod K$ then
 8:         for $j = 1$ to $k$ do
 9:             Sample transition $j \sim P(j) = p_j^\alpha / \sum_i p_i^\alpha$
10:             Compute importance-sampling weight $w_j = (N \cdot P(j))^{-\beta} / \max_i w_i$
11:             Compute TD-error $\delta_j = R_j + \gamma_j Q_{\text{target}}(S_j, \arg\max_a Q(S_j, a)) - Q(S_{j-1}, A_{j-1})$
12:             Update transition priority $p_j \leftarrow |\delta_j|$
13:             Accumulate weight-change $\Delta \leftarrow \Delta + w_j \cdot \delta_j \cdot \nabla_\theta Q(S_{j-1}, A_{j-1})$
14:         end for
15:         Update weights $\theta \leftarrow \theta + \eta \cdot \Delta$, reset $\Delta = 0$
16:         From time to time copy weights into target network $\theta_{\text{target}} \leftarrow \theta$
17:     end if
18:     Choose action $A_t \sim \pi_\theta(S_t)$
19: end for
```

B. PPO

PPO is a pseudo off-policy algorithm in which a slightly older policy is used to sample episodes for each epoch. The rewards and advantage functions are computed in order to update the policy. The policy is updated via a stochastic gradient ascent optimizer, while the value function is fitted via gradient descent method. The procedure is followed for a number of training epochs.



```
Algorithm 1 PPO-Clip
1: Input: initial policy parameters $\theta_0$, initial value function parameters $\phi_0$
2: for $k = 0, 1, 2, \ldots$ do
3:     Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
4:     Compute rewards-to-go $\hat{R}_t$.
5:     Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based
       on the current value function $V_{\phi_k}$.
6:     Update the policy by maximizing the PPO-Clip objective:
           $\theta_{k+1} = \arg\max_\theta \frac{1}{|\mathcal{D}_k| T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min\left( \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t), \ g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t)) \right),$
       typically via stochastic gradient ascent with Adam.
7:     Fit value function by regression on mean-squared error:
           $\phi_{k+1} = \arg\min_\phi \frac{1}{|\mathcal{D}_k| T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left( V_\phi(s_t) - \hat{R}_t \right)^2,$
       typically via some gradient descent algorithm.
8: end for
```

There are two primary variants of PPO: PPO-Penalty and PPO-Clip. PPO-Penalty approximately solves a KL-constrained update like TRPO, but penalizes the KL-divergence in the objective function instead of making it a hard constraint, and automatically adjusts the penalty coefficient over the course of training so that it's scaled appropriately. PPO-Clip relies on specialized clipping in the objective function to remove incentives for the new policy to get far from the old policy.PPO-clip updates policies via typically taking multiple steps of (usually minibatch) SGD to maximize the objective.

PPO trains a stochastic policy in an on-policy way. This means that it explores by sampling actions according to the latest version of its stochastic policy. The amount of randomness in action selection depends on both initial conditions and the training procedure. Over the course of training, the policy typically becomes progressively less random, as the update rule encourages it to exploit rewards that it has already found. This may cause the policy to get trapped in local optima.

V. RESULTS

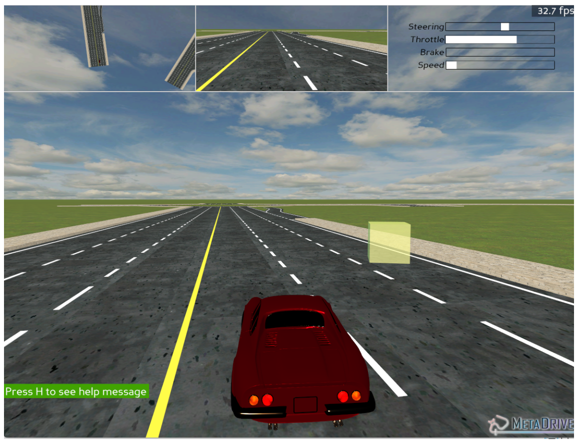


Fig.: Ego vehicle at start of each episode

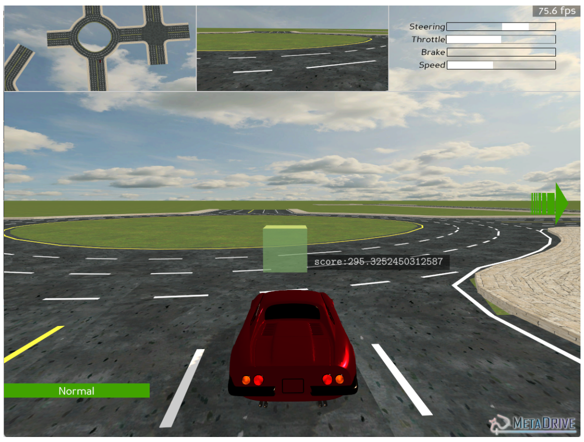


Fig.: Ego vehicle approaching roundabout

A. DQN

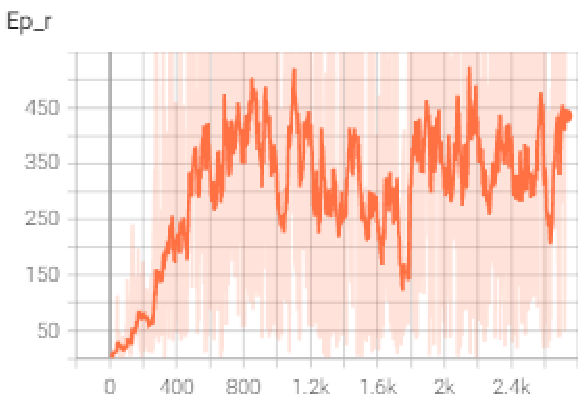


Fig.: Reward per episode

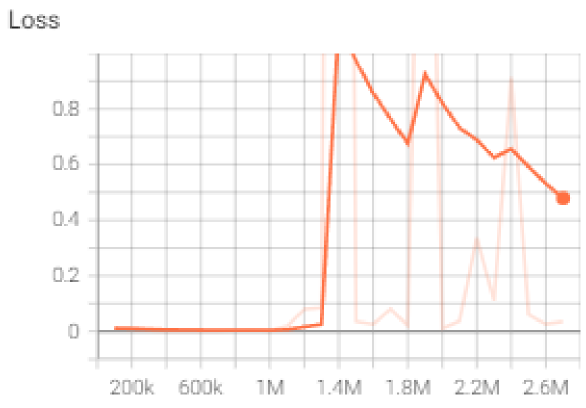


Fig.: Reward per episode

B. Double DQN

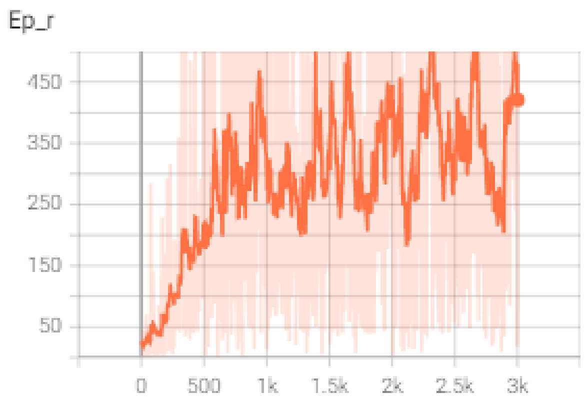


Fig.: Reward per episode

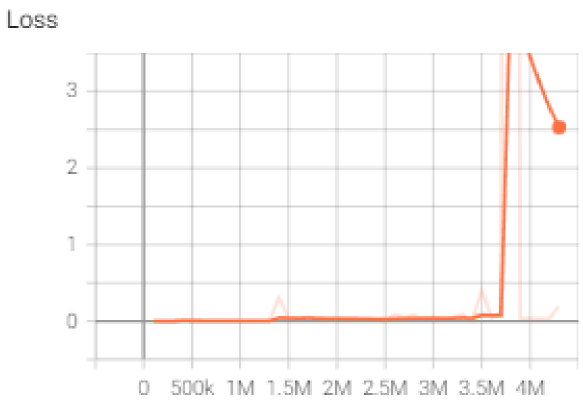


Fig.: Reward per episode

C. Dueling DQN

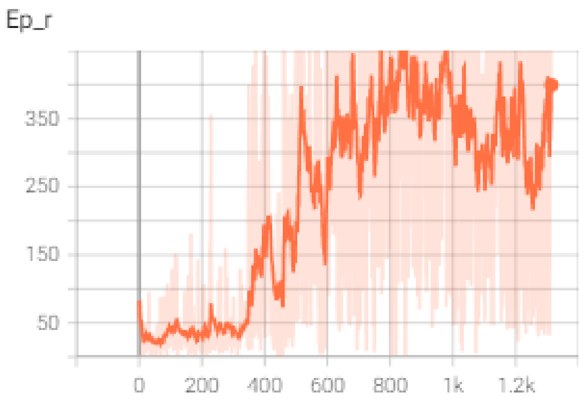


Fig.: Reward per episode

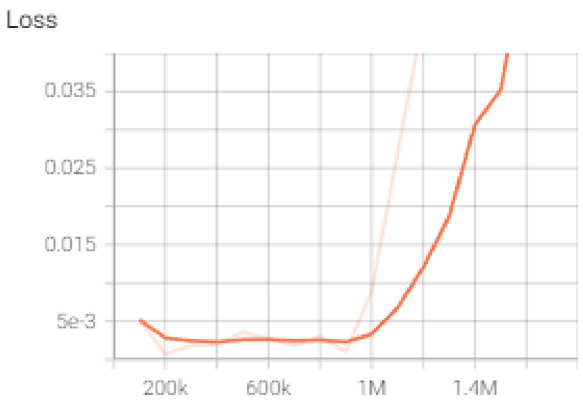


Fig.: Reward per episode

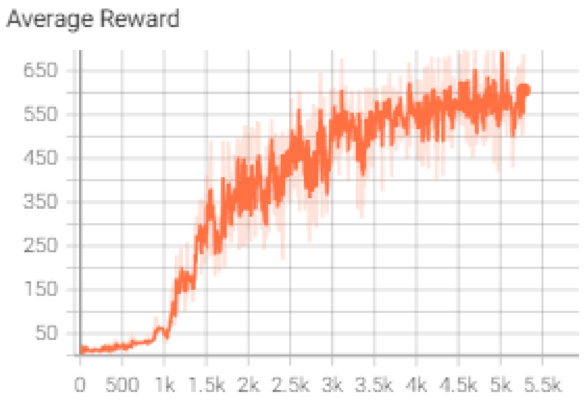


Fig.: Average Rewards

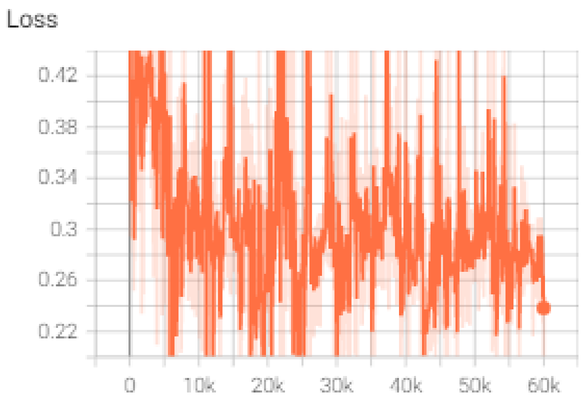


Fig.: Loss per update step

E. PPO in continuous space

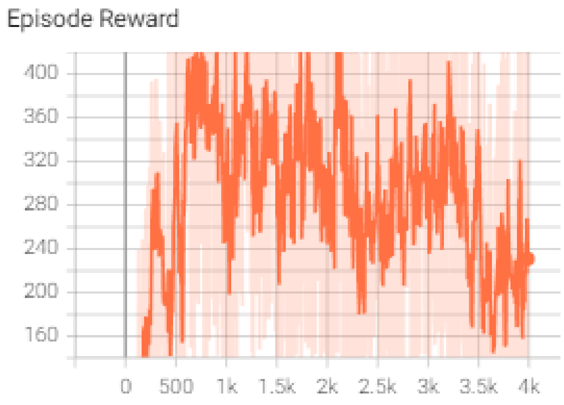


Fig.: Reward per episode

D. PPO in discrete space

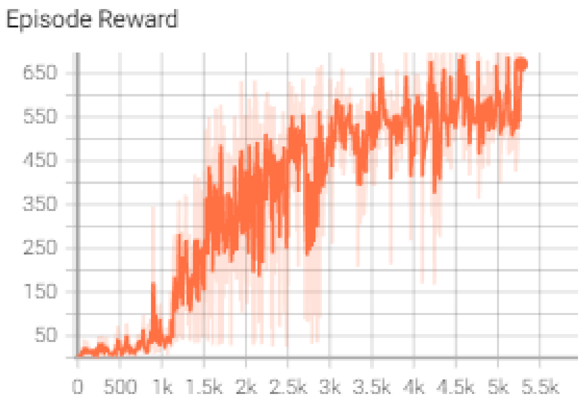


Fig.: Reward per episode

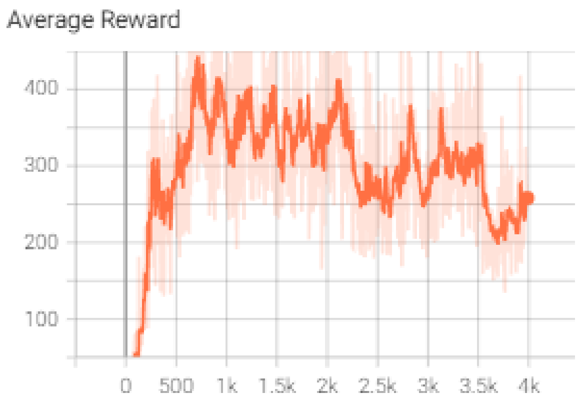


Fig.: Average Rewards

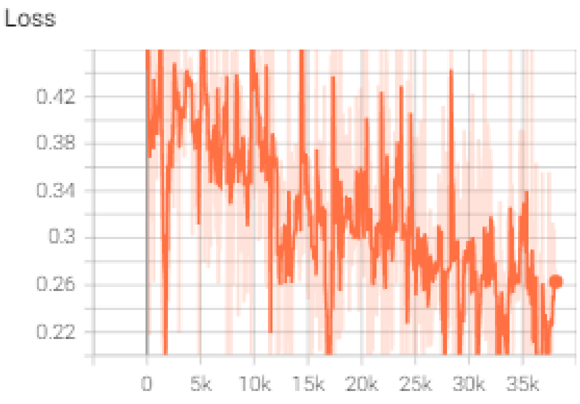


Fig.: Loss per update step

VI. DISCUSSION

The value based Reinforcement Learning algorithms performed decently good in making the agent learn how to drive with the given observation space.Traditional DQN was able to get a consistent reward of 400 after 2k episodes which was observed to be decent enough to drive the car. The Double DQN algorithm which improves the drawbacks of DQN was able to perform similar to DQN with little high episodic rewards. Dueling DQN was able to get the fastest improvements in rewards by getting consistent 350+ rewards in just 800 episodes. Even after able to drive the agent well, all the DQN based algorithms saw an explosion of loss at some point which is a drawback of DQN Family algorithms due to the over-estimation problem faced by Deep Q-Learning algorithms.

Policy Gradient Reinforcement Learning algorithms performed much better than value based methods. For policy gradient based methods we have use Proximal Policy Optimization for discrete and continuous space. PPO for discrete action space limits the descent in gradient which solves the problem of over-estimation that caused loss explosion in DQNs. Even though PPO saw a gradual improvement in reward, the increment was consistent without any excessive loss overshoots because of gradient clipping. PPO was able to achieve average rewards of 600 which was much better than DQNs. PPO for continuous space uses mean and variance for actions rather then generating a probability like in case of discrete space. As the agent learns the policy converges to a optimal mean with less variance. Since, the action is sampled from a distribution there are always chances of exploration.

VII. CONCLUSION

This project has demonstrated the effectiveness of navigating the autonomous vehicles using reinforcement learning methods. We have showed the effectiveness of deep Q-networks and policy gradient method in controlling the vehicles with higher dimension of sensory inputs in the MetaDrive simulator. The performance of the PPO algorithm in the driving is better than the DQN and it's variants with stable properties which is congruence to the understanding that policy gradient methods work better in higher dimensions and have better convergence properties. DQN is a value-iteration based method and the algorithm does not directly optimize for reward but instead focuses on learning a function approximator to predict Q-values that satisfy the recursive Bellman Equation. The optimal actions are then derived from the action which maximizes the Q-value however, PPO simplifies the computationally expensive constraint calculations and second-order approximations in the Trust Region Policy Optimization (TRPO) algorithm.

REFERENCES

[1] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play (2018)

[2] Stuart Russell,Peter Norvig, Artificial Intelligence: A Modern Approach

[3] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller, Playing with Atari with Deep Reinforcement Learning,2013

[4] S. Kuutti, R. Bowden, Y. Jin, P. Barber, S. Fallah, A survey of deep learning applications to autonomous vehicle control IEEE Trans. Intell. Transp. Syst. (2020)

[5] Huang, Z., Xu, X., He, H., Tan, J., Sun, Z., 2017. Parameterized batch reinforcement learning for longitudinal control of autonomous land vehicles. IEEE Transactions on Systems, Man, and Cybernetics:Systems 49(4), 730 – 741.

[6] Min, K., Kim, H., Huh, K., 2019. Deep distributional reinforcement learning based high level driving policy determination. IEEE Transactions on Intelligent Vehicles 4(3), 416 – 424.

[7] Hitesh Arora, Master's thesis, 2020. Off policy Reinforcement Learning for Autonomous Driving

[8] Oleguer Canal and Federico Taschin, Self-learned vehicle control using PPO (2020)

[9] Tanmay Agarwal, Master's thesis (2020), On-Policy Reinforcement Learning for Learning to Drive in Urban Settings