UNIVERSITY OF CALIFORNIA SAN DIEGO

Motion Planning via Reinforcement Learning

A thesis submitted in partial satisfaction of the
requirements for the degree Master of Science

in

Electrical Engineering (Intelligent Systems, Robotics, and Control)

by

Awies Mohammad Mulla

Committee in charge:

Professor Nikolay A Atanasov, Chair
Professor Sylvia Lee Herbert
Professor Yuanyuan Shi

2024

The Thesis of Awies Mohammad Mulla is approved, and it is acceptable in quality and form for publication on microfilm and electronically.

University of California San Diego

2024

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT OF THE THESIS

Motion Planning via Reinforcement Learning

by

Awies Mohammad Mulla

Master of Science in Electrical Engineering (Intelligent Systems, Robotics, and Control)

University of California San Diego, 2024

Professor Nikolay A Atanasov, Chair

This thesis focuses on solving the problem of motion planning for autonomous mobile robots using reinforcement learning. In recent years, reinforcement learning has shown remarkable potential in training autonomous agents to navigate complex environments. In the context of autonomous ground vehicles, one of the fundamental challenges is achieving safe navigation amidst obstacles. Our work develops a machine learning model trained using the Proximal Policy Optimization (PPO) algorithm to guide an Ackermann drive car towards a goal while avoiding obstacles. The key innovation lies in the fusion of robot states and depth images using a neural network architecture to extract features for training an actor and a critic for the navigation problem. Effective training depends on optimality criteria for the planned path. Our

work considers the time to navigate to the goal, a collision penalty, and several regularization terms ensuring the smoothness of the control policy based on a Lipschitz continuity measure. We demonstrate that our approach can be transferred to a real F1-tenth Ackermann-drive robot with close to no tuning.

Chapter 1

Introduction

1.1 Motivation

The advent of Reinforcement Learning (RL) has revolutionized the field of autonomous robotics, offering a data-driven approach to train agents capable of making intelligent decisions in complex and dynamic environments. Specifically in the domain of autonomous navigation, deep learning plays a major role in solving perception tasks, for instance pedestrian trajectory prediction. On the contrary, the idea of incorporating data-driven approach for motion planning of autonomous systems is active area of research. Thanks to the significant improvement in the computation and actuators' capabilities (implementation of precise control commands), we can witness an increase in the level of autonomy without supervision among various systems ranging from warehouse automation to surgical assistance. Given the extensive deployment of deep learning algorithms in various aspects of daily life, it has become increasingly critical to quantify their reliability and safety. This is essential to ensure both the security of the surrounding environment and the safe operation of robots.

This work focuses on the specific application of training an Ackermann drive car to navigate through an obstacle-rich environments and reach a predefined goal. Navigating an Ackermann drive car in these environments poses a challenging problem, of implementing appropriate controls in cluttered environments, that necessitates the development of robust and adaptive control strategies. Traditional methods often rely on hand-crafted control policies that

may struggle to handle the diversity and complexity of real-world scenarios. In contrast, RL empowers agents to learn effective control policies through interaction with their environment, making it a promising avenue for addressing these challenges. The generated control policies are ensured to be well within the bounds of the actuators while maintaining stability in terms of change in policy (i.e., free of oscillating behaviours). Hence, reducing unnecessary strain on the actuators.

1.2 Contributions

The contribution of this thesis is an approach for point-goal navigation of an Ackermann-drive robot in an unseen environment using a neural network control policy to map raw sensor data and robot state information to actuator controls. The stochastic policy trained in the simulation successfully allows the agent to reach the goal in an obstacle-rich environment. Training the policy in simulation provides a safe, cost-effective setting to develop, test and refine RL algorithms before real world deployment. The trained policy is stable with respect to actuator bounds to deploy on a vehicle reducing the efforts in fine-tuning. Said stability was achieved by designing a regularization strategy of constraining the policy to be Lipschitz continuous. The algorithm was trained in two simulation environments and evaluated in same simulation engines along with evaluation of trained policy on real-world experiments.

1.3 Related Work

Incorporation of deep learning algorithms in various motion planning task has become an active area of research over the past few years. Although, obtaining the training data for such problem formulations was considered as one of the major challenges, the degree of generalization which can be achieved using these techniques outweighs the compute and time consumed by these approaches to model a solution. For instance, [1] addresses the problem of high speed navigation in unknown environment by designing an approach which uses Bayesian non-parametric learning

algorithm that encodes formal safety constraints as a prior over collision probabilities. This planner seamlessly reverts to safe behavior when it encounters a novel environment for which it has no relevant training data. Using traditional planners such as Model Predictive Control (MPC) in such unseen environments would guarantee the safety of the mobile robot, but they cannot achieve the safe high-speed navigation by themselves as done in [1]. In recent times there have been works which does not on rely traditional planners at any point in their algorithms. One of the novel works which falls under this category and from which our work has been inspired is [9]. In [9], a quadreped robot is trained to perform locomotion tasks using RL with a Transformer-based model that combines proprioceptive information and high-dimensional depth sensor inputs. Other major work which inspired our problem of point-goal navigation is [8]. In [8], agent is deployed with a pre-trained policy trained on diverse data along with autonomous practicing framework for continuous online improvements in large real-world environments. Using this framework, the robot autonomously navigate between sparse checkpoints and recovers from the collisions during practice and improve its driving behaviour to maximize speed. Agent in [8] can learn aggressive driving comparable to a human expert within 20 minutes of autonomous practice. So, inspiring from [8], we decided to model an approach which would let an agent undergo model-free training in simulation until it yields promising results, and later deploy the trained policy in a real world unseen environment.

Chapter 2

Background

2.1 Markov Decision Process

The problem being addressed here is formulated as a first-exit finite horizon Markov Decision Process (MDP). A Markov Decision Process is a mathematical framework used for modeling decision-making in processes where the next state of the agent is only dependent on the current state and action taken in that particular step. The state space (S) of an MDP is defined as set of all possible states in which an agent can exist. The action space (A) of an MDP is defined as the set of all possible actions that an agent can take at a particular state. Transition function, $P(s'|s,a)$, describes the probability of transitioning from state s to state s' ($s, s' \in S$) given that agent takes action s ($a \in A$). This encapsulates the dynamics of the environment. Reward function of an MDP, $R(s,a,s')$ provides the immediate reward received after transitioning from from state s to state s' on taking action a. It quatifies the action taken by the agent in a particular state. Discount factor ($\gamma \in [0,1]$) in an MDP represents the importance of future rewards compared to immediate rewards. Discount factor close to 1 means future rewards would have comparable emphasis to immediate rewards whereas, discount factor close 0 means agent places much higher emphasis on immediate reward compared to future rewards. So, for any MDP it is necessary to model the dynamics of the agent when interacting with the environment. We define system dynamics in the following section.

2.2 System Dynamics

The agent in our problem is defined as an Ackermann drive car. One of the primary reasons for adopting this model over the differential drive model, was to mimic realistic car-like movement as much as possible. It provides an upper-hand in high speed navigation by increasing manueverability of the vehicle. In this work, agent is provided with the controls, throttle (a) and steering angle (ϕ), at every timestep.

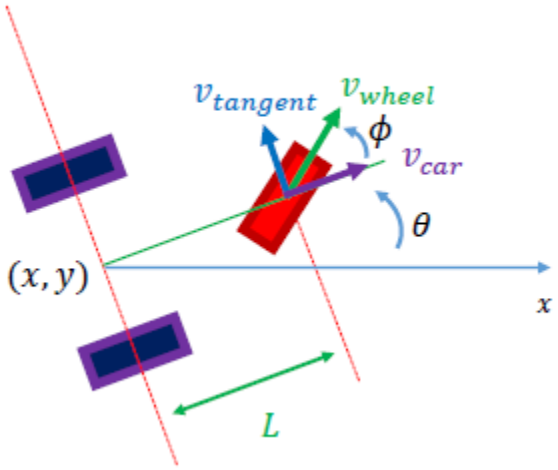


Figure 2.1. Kinematics of Ackermann Steer Vehicle

Fig. 2.1 represents a model of an Ackermann drive car, where, (x,y) represent the position of the car in world frame, ϕ represents steering angle, v_{car} represents heading velocity of the car, θ represent orientation of the car about z-axis (yaw). The equations of motion are given by,

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \cos\theta \\ \sin\theta \\ \frac{1}{L}\tan\phi \end{bmatrix} v^t_{car} \tag{2.1}$$

$$\dot{v_{car}} = a \tag{2.2}$$

where, L represents length of car. In our setup, we are providing absolute steering angle as one of the controls rather than rate of change steering angle. This was done to ensure that agent is able to implement tight turns given the limited information it is being fed about the environment.

2.3 First-Exit Finite Horizon Problem

The first-exit finite horizon problem is a form of stochastic optimal control problem commonly studied in the context of RL. The first-exit finite horizon problem is defined as:

$$\max_{\pi_{\tau:T-1}} V_\tau^\pi(s_\tau) := \mathbb{E}_{s_{\tau+1:T}} \left[\gamma^{T-\tau} R(s_T) + \sum_{t=\tau}^{T-1} \gamma^{t-\tau} R(s_t, \pi_t(s_t), s_{t+1}) \mid s_\tau \right]$$

subject to:

$$s_{t+1} \sim P(\cdot \mid s_t, \pi_t(s_t)), \quad t = \tau, \ldots, T-1$$
$$s_t \in S, \quad \pi_t(x_t) \in A$$

where, $V_\tau^\pi(s_\tau)$ denotes the value function (expected cumulative reward) at state s_τ, s_t denotes state at time t, $\pi_t(s)$ denotes policy function at time t, $P(s'|s,a)$ denotes transition function, $a \in A$ denotes action in action space, $\gamma \in [0,1]$ denotes discount factor, T is the planning horizon or timestep of terminal state (which here is a random variable depending on the set of terminal states ($s_T \in \mathscr{T} \subseteq S$)) depending on which is minimal, $R(s_t, \pi_t(s_t), s_{t+1})$ denotes immediate reward and $R(s_T)$ denotes reward in terminal state.

Said formulation is implemented in an episodic setting, where each episode represents a complete sequence of interactions between an agent and the environment. An episode begins at an initial state and concludes when a terminal state is reached or after a predefined number of steps. During an episode, the agent repeatedly takes actions based on the current state, receives rewards (representing feedback from the environment), and transitions to new states according to the environment's dynamics.

So, in our implementation planning horizon represents maximum number timesteps agent is provided to reach the goal. Set of terminal states, $\mathscr{T}$, consists of states where either agent collides with the obstacle or agent has reached the defined goal position.

2.4 Exploration - Exploitation in RL

In RL, the exploration-exploitation balance is a crucial principle that guides an agent's learning process. Exploration involves trying new actions to gather information about the environment, helping the agent discover potentially better strategies. Exploitation, on the other hand, involves using known actions that yield high rewards based on current knowledge. Striking the right balance between these two behaviors is essential: too much exploration can lead to excessive experimentation without progress, while too much exploitation can cause the agent to miss out on optimal strategies. Effective management of this balance, through methods like ε-greedy, ensures the agent can learn and perform optimally in dynamic environments. Here we model the policy function as probability distribution, so sampling from this distribution would serve as the purpose of exploration, while the action of maximum probability in the probability density function of the distribution would imply exploitation.

2.5 Distributions

As mentioned in Section 2.4, a stochastic policy specifies a probability distribution over the action space of the robot to support an exploration-exploitation trade-off while learning. Policy function represented as $\pi_\theta(a|s)$, where a is an action and s is a state, is a probability distribution, with mode of the distribution being the action representing the action with maximum value (or reward) at a particular state. To incorporate exploration of the agent while interacting with the environment agent samples an action from the distribution while training. Usually

$\pi_\theta(a|s)$ is a Gaussian probability density function such that,

$$a \sim \mathcal{N}(\mu_\theta(s), \sigma_\theta^2(s))$$

where, $\mu_\theta(s)$ represents the mean action to be taken at state s to maximise reward and $\sigma_\theta(s)$ represents variance in the action space depicting the randomness permitted at a particular state. The mean action, $\mu_\theta(s)$, is the output provided by the actor network (depending on the algorithm used, actor network can be modeled as look-up table or a neural network). Since, the mean in normal distribution is free of any kinds of bounds, it is not suitable for our formulation. The controls of the agent are bounded, so as a straight forward implementation, network output would be clipped to ensure that it is in accordance with bounds of the environment. As a result the trained policy only consisted of extreme controls in the action space (also led to oscillating behavior). Hence, considering this fact we landed on Beta distribution which is defined as,

$$\pi_\theta(a|s) \sim \text{Beta}(\alpha, \beta)$$

where α and β are the shape parameters of the Beta distribution. The probability density function (PDF) of the Beta distribution is given by:

$$\pi_\theta(a|s) = \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)} a^{\alpha-1}(1-a)^{\beta-1}$$

where a, would be a sampled action from the distribution for exploration, $\Gamma(.)$ represents gamma function, which generalizes the factorial function to continuous values and is defined as,

$$\Gamma(z) = \int_0^\infty t^{z-1} e^{-t}\, dt$$

For positive integers n, the gamma function satisfies $\Gamma(n) = (n-1)!$. The mean of the Beta distribution which is considered as action of maximum reward (action to be taken in deterministic

setting while evaluating) is given as,

$$\mu_B = \frac{\alpha}{\alpha + \beta}$$

In the case of normal distribution, mean action from corresponding action space is the output of the network, μ whereas for the beta distribution, output of the network would be the parameters of Beta distribution, α and β.

Chapter 3

Problem Formulation

We have addressed two major problems in this work, first being navigation of an RL agent to a point goal in an unseen environment. The second problem which we address is, regularization of the policy function used for navigation by constraining it to be Lipschitz continuous.

3.1 Point goal navigation

We are attempting to navigate an agent (Ackermann modeled car), to a point goal ($\mathbf{x}_G = [x_G, y_G]^T$). More precisely, a formulated neural network ($f(.)$) would take a series of state-space information (s_t) and depth images (I_t) as inputs and output the controls (u_t) to reach the desired goal. The state-space of the model at a particular time-step comprises of relative position of the goal w.r.t. agent, ΔX_t, magnitude of velocity of the agent ($\mathbf{v}_t$), and the steering angle (ϕ_t). Given the position and orientation of agent at ($\mathbf{x}_t = (x_t, y_t)$) and ($\omega_t$), respectively, the relative position of the goal in the current car frame is given by:

$$\Delta X_t = R(\mathbf{x}_G - \mathbf{x}_t)$$

where R is the rotation matrix given by:

$$R = \begin{bmatrix} \cos(\omega_t) & \sin(\omega_t) \\ -\sin(\omega_t) & \cos(\omega_t) \end{bmatrix}$$

We model the interaction between the agent and environment as an Markov Decision Process (S, A, P, R, H, γ), where S is the state-space, A is the action-space, $P(S'_t|S_t, u_t)$ is the transition probability, R is the reward function, H is the finite episode horizon, and γ is the discount factor. The agent learns a policy π_θ parameterized by θ to output action distributions conditioned on the current state. The goal is to learn the weights, θ, that maximize the discounted episode return: $\mathbb{E}_{\tau \sim p_\theta(\tau)}[\sum_{t=0}^{H} \gamma^t r_t]$, where r_t is the reward at every timestep t, $\tau \sim p_\theta(\tau)$ is the trajectory.

Since, we are considering current state-space information along with the history, to estimate the desired controls. The input of the neural network is given by:

$$S_t = [s_t, s_{t-1}, s_{t-2}, s_{t-3}, I_t, I_{t-1}, I_{t-2}, I_{t-3}]^T$$

where $s_t = [\Delta X_t, \mathbf{v}_t, \phi_t]^T$ and I_t is the depth image (64×64 pixels) at time-step t. The output of the neural network is given by:

$$u_t = f(S_t) = \begin{bmatrix} \delta \mathbf{v}_t \\ \phi_t \end{bmatrix}$$

where u_t is a vector including the throttle $\delta v \in [0, 10]$ and steering angle $\phi \in [-0.7, 0.7]$.

We were faced with another problem while deploying above mentioned formulation on a F1-tenth car. The policy obtained on training the agent in the simulation, seemed to complete the mentioned task in simulation. But on closely analysing the trained policy, we inferred that it was noisy. Hence, we devise the following formulation to deal with problem.

3.2 Regularization of RL policy

Our goal is to regularize the policy described in the Sec. 3.1, π_θ. The regularized policy should be such that it should not lose the underlying defining property which is essential to complete designed task; by being less noisy compared to unregularized policy. In other words, we would like the rate of change of π_θ with respect to consecutive states, to be bounded, so we don't observe any irregular changes in the policy for consecutive timesteps. This can be achieved by enforcing the Lipschitz continuity constraint Eq. 3.1 on the policy while training.

$$||\pi_\theta(x) - \pi_\theta(x')||_p \leq L||x - x'||_p \; \forall \; x, x' \in S, \tag{3.1}$$

where L is the Lipschitz constant. Determining the Lipschitz constant of a neural network is established as an NP-hard problem in [5]. We aim to minimize the ratio, $\frac{||\pi_\theta(x)-\pi_\theta(x')||_p}{||x-x'||_p}$ while training to overcome irregular jumps in the policy with respect to consecutive states.

Chapter 4

Reinforcement Learning Policy Design

To achieve our goal, we successfully designed a stable and robust policy. Said policy worked as an end-to-end framework from sensor data to the controls for the actuators. As mentioned in the Section 1.3, [9] tries to address a similar problem. [9] attempts to estimate the actuation to be provided to the quadrepad robot using an end-to-end neural network with its input being raw sensor reading. Inspiring from this work, we model the solution of our problem in a similar manner. We have divided the overall architecture into two significant parts: Feature Extractor and Actor-Critic network. Following sections would explain the mentioned approach in detail.

4.1 Feature Extractor

We are using a neural network ($h(.)$) to transform the raw sensor data into n-dimensional (n is varied depending on feature extractor) feature vector. We landed on a particular architecture after extensive experimentation. Details of each of the possible architecture is mentioned below.

4.1.1 State-Only Feature Extractor

In this case, we are only considering a history of states of the agent, while operating in an empty (obstacles devoid) arena. As we don't have any visual sensor to obtain data related to the environment, agent is not expected to perform obstacle avoidance of any kind. Since, the mapping from car's state to actuations based on relative position of the goal is not as complex

as compared to combined sensor readings. On the other hand, it is also not sparse enough to be pre-computed and tabulated. Hence, we modeled this feature extractor as an Multi-Layered Perceptron (Fig. 4.1) containing only two hidden layers with an activation function.

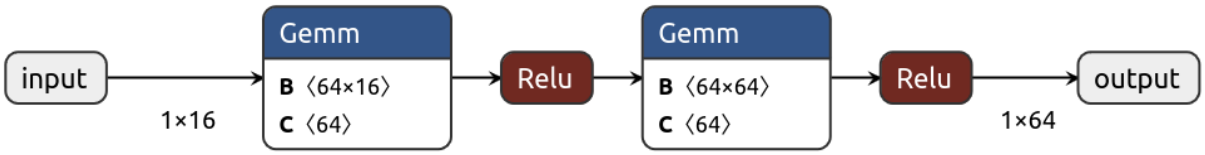


Figure 4.1. Architecture for State-Only Feature Extractor

4.1.2 State-Depth Concatenation Feature Extractor

We established in Section 4.1.1 that we can transform the states of the agent to a feature vector. We can adopt a similar approach to process the history of depth images. Now the architecture shown in Fig. 4.2 is inspired from a general encoder-decoder architecture in [4], which process depth images to obtain semantic map of an image. Since, we are not super-imposing the semantic map onto input image, we are only concerned about the extracted features from the image i.e., focused mainly on contracting path as mentioned in [4].

Here we process the depth images and states of the agent separately and concatenate the features obtained from both the networks to feed the actor-critic networks. The images are processed via a CNN (Fig. 4.2) and the state-space information is processed via a FCN (Fig. 4.1). Since, we are only concerned about the position and not about the visual characteristics of the obstacles and (for obstacle-avoidance), we decided to use depth images as the only visual input for the agent. Using RGB images would not only increase our computation requirements but also not assist the agent in solving the problem of obstacle avoidance.

4.1.3 Vision-Transformer Feature Extractor

Now, we try to process the depth images in a much complicated manner. We decided to use the Attention mechanism mentioned in [2], to correlate the patches of the image for obtaining the features of the history of depth images using a Transformer. The intuition behind using

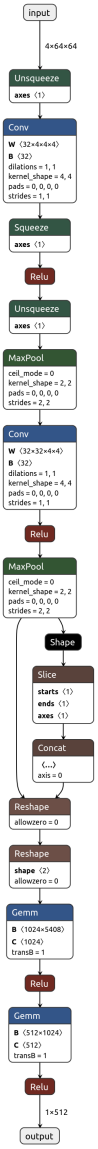


Figure 4.2. CNN to process images in State-Depth Feature Extractor

Transformer is that, multiple patches containing depth images would give a higher correlation helping the agent to identify the obstacle in a much better manner. Vision Transformer was implemented with 6 Multihead Attention layers which had 8 Attention Blocks each. The size of input for images is $4 \times 64 \times 64$. The input images are divided into patches such that we get blocks of the images of shape $4 \times 8 \times 8$. These 64 patches are flattened and used as the tokens for Attention mechanism to compute the correlation between the patches. Network was trained sequentially by varying the parameters of the environment to get to desirable results as compared

to the State-Depth Concatenation Feature Extractor.

- Train for 300k timesteps with no obstacles and a fixed goal.
- Train with random goal positions with no obstacles for 300k timesteps.
- Next train the previous model with fixed goal but 5 obstacles to introduce the agent about the notion of obstacles. This is carried out for 1M timesteps.
- Train with random goals and 5 obstacles for 400k timesteps. Lastly, agent is trained with 15 obstacles (number of obstacles could be varied) and random goal.

With this, it was found that the agent was able to systematically learn to navigate from a simple environment to the complex environment. Even after dedicating such an intensive computation power, the Vision-Transformer model performed upto the level of State-Depth Concatenation at best. Hence, we decided to finalise the state-depth concatenation feature extractor in our resultant policy which would be evaluated on F1-tenth car.

4.1.4 State-Depth Tranformer Feature Extractor

Now, we will process the state of the car along with the depth images in a single Transformer. Intuitively this makes much more sense compared to previous approach. As we can calculate the correlation of the state with the patches in the image, the agent can more accurately update the weights such that actuation provided by the network lets the agent to avoid the obstacles. The network architecture was fixed to 8 attention blocks with 6 multihead attention in each layer. The size of input for images is $4 \times 64 \times 64$. The input images are divided into patches such that we get blocks of the images of shape $4 \times 8 \times 8$. These 64 patches are flattened and used as the tokens along with state-space information vector ($[s_t, s_{t-1}, s_{t-2}, s_{t-3}]$) as one of the token, for Attention mechanism to compute the correlation between the patches and states of the agent. We were unable to observe convergence of the loss function in reasonable time for

this architecture. Hence, we decided to train the network under various the parameters of the environment. We observed following behaviour during training.

- The agent was trained for 330k timesteps with a static non-randomized environment. The agent was able to complete the required task. So, we decided increase the complexity.
- State-Depth Transformer was now trained with 1.2 Million time-steps (from randomly initialized network) with a static goal and static obstacles randomized at start of each episode.
- In this scenario, agent was able to navigate to the defined point-goal few times in the early stages of the training. We were able to observe drastic change in the policy after a certain number of epochs and agent was no more able to reach the required coordinates.
- We can interpret this behaviour from the normalized reward we obtain while training the current feature extractor. Normalized reward was observed to oscillate between 0 and 1 in an uncontrollable manner.
- We hypothesised, this could be possible due to limited training given the complexity and size of the architecture. We believe if this architecture is trained for larger amount timesteps, it would yield reasonable results.
- Since, the state-depth concatenation feature extractor was already performing exceptionally, it was not justifiable to use far more computational resources to achieve similar performance in state-depth-transformer feature extractor.

4.2 Policy Optimization Algorithm

The extracted features will serve as the input to the policy function. For low-dimensional state spaces we could generate a look-up table which would map the features to the required actions. But due to complexity of the system, we decided to model the actor as an Multi-Layered

Perceptron (MLP), represented as π_θ. To train our policy network, we adopted Proximal Policy Optimization (PPO) algorithm, [7] which is considered as state-of-the-art in reinforcement learning. PPO is an actor-critic based on-policy algorithm. As mentioned in Sec. 2.5, the actor network outputs the parameters for of the probability distribution of policy. Hence, the architecture would vary according to the probability distribution. After some experimentation, we landed on the architectures for actor and critic shown in the following figures.

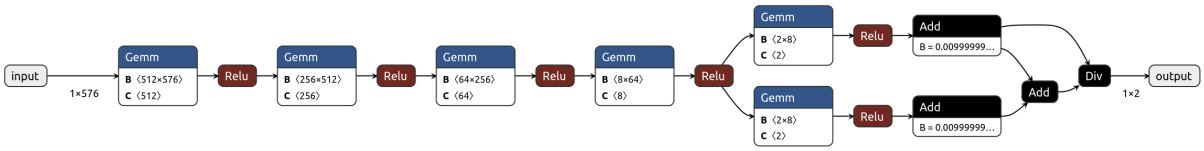


Figure 4.3. Architecture for Actor with Beta Distribution

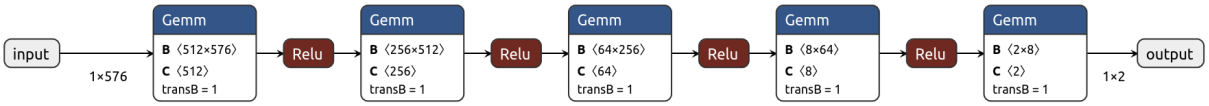


Figure 4.4. Architecture for Actor with Gaussian Distribution

The architectures for actor and the critic are similar in the manner such that each of those process the features, differing only on the output we obtain. Critic gives us the expected reward of a particular state (so, the architecture of critic would be independent of the policy distribtion), while the actor gives us the parameters of probability distribution modeling the controls, steering angle and throttle, as mentioned in Section 3.1.

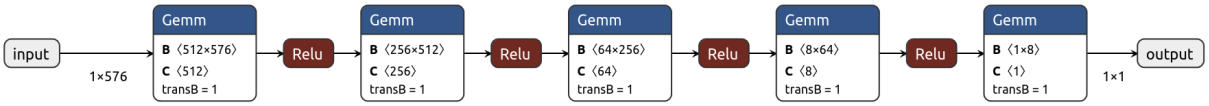


Figure 4.5. Architecture for Critic

4.2.1 Proximal Policy Optimization

From the MDP formulation, the finite episode horizon is $H = 1500$ timesteps, $\gamma = 0.99$, transition probability $P(S'_t|S_t, u_t)$ is the function of the policy distribution π_θ, and the reward

function at a particular timestep is represented as r_t. The pseudocode for the PPO is mentioned in Algorithm 1.

Algorithm 1. Proximal Policy Optimization (PPO)

for iteration = 1, 2, 3, ... **do**
  for actor = 1, 2, 3, ... N **do**
    Run policy $\pi_{\theta_{old}}$ in environment for H timesteps
    Compute advantage estimates $A_1, ..., A_H$
  end for
  Optimize surrogate L wrt θ, with K epochs and minibatch size $M \leq NH$
  $\theta_{old} \leftarrow \theta$
end for

In our implementation, $H = 1500, N = 4, K = 50$ and $M = 300$. Each buffer of the actor is of size $NH = 6000$. The loss L is optimized using a buffer for $K = 50$ epochs with minibatch size of $M = 300$. The advantage for a particular timestep is given by:

$$A_t = \delta_t + (\gamma\lambda)\delta_{t+1} + \cdots + (\gamma\lambda)^{H-t+1}\delta_{H-1} \tag{4.1}$$

where, $\delta_t = r_t + \gamma V_\theta(s_{t+1}) - V_\theta(s_t)$ is the temporal difference error, and λ is the GAE parameter. Now, the loss function is given by:

$$L_t^{CLIP+VF+S} = \hat{\mathbb{E}}_t[L_t^{CLIP} + c_1 L_t^{VF} + c_2 S[\pi_\theta(s_t)] + c_3 L_t^{LC}] \tag{4.2}$$

where, L_t^{VF} is a squared-error loss $(V_\theta(s_t) - V_t^{targ})^2$, V_t^{targ} is Temporal Difference (GAE Lambda) return (from [6]) at timestep t depending on the state in which agent is present. This is calculated using the user-defined reward function (4.2.2). S denotes an entropy bonus (this denotes the degree of randomness in the policy distribution during the training). Currently, this bonus is denoted as mean logit of the variance of the policy distribution. L_t^{LC} is regularization expression use to constraint the policy to be Lipchitz continuous. We will be discussing L_t^{LC} in Sec. 4.3. The hyperparameters c_1, c_2, c_3 are coefficients for squared-error loss, entropy bonus and regularization

expression respectively. L_t^{CLIP} is the clipped surrogate objective:

$$L_t^{CLIP} = \hat{\mathbb{E}}_t \left[\min \left(r_t(\theta) A_t, clip(r_t(\theta), 1-\varepsilon, 1+\varepsilon) A_t \right) \right] \tag{4.3}$$

where, $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$ is the probability ratio. The hyperparameter ε is the clipping parameter, which is an indication of how much a policy is permitted to deviate during consecutive iterations.

4.2.2 Reward Function

As we know from the equation 4.2, that the reward function is the essence of any RL algorithm. We wanted the reward function to be as sparse as possible for generalizing to agents of different dynamics. After experimentation we landed on a reward function which we implemented. Various reward functions which were experimented are mentioned below:

1. **Distance-based reward**: Reward is defined as:

$$r_t = \mathbb{1}_C(-1000) + \mathbb{1}_G(2000) + \alpha e^{(-\beta d)} - 1 \tag{4.4}$$

   In the above reward function, we are penalising the agent with respect to time (directly proportional), i.e., reward of '-1' at every timestep, so that agent does not rest in a particular position throughout the training. We are heavily penalising the agent when hitting the obstacle and rewarding equally heavily when it reaches the goal during training. We also provide the agent with an incremental exponential reward as it reaches closer to the goal (as a function of relative position of goal).

$$d = ||\Delta X_t||_2$$

2. **Distance and orientation based reward function**: Reward is defined as:

$$r_t = \mathbb{1}_C(-1000) + \mathbb{1}_G(2000) + \alpha e^{(-\beta d)} + \zeta e^{(-\eta \frac{y}{d})} - 1 \tag{4.5}$$

where, $\mathbb{1}_C$ is the indicator function for collision, $\mathbb{1}_G$ is the indicator function for reaching the goal, d is the relative distance of the agent from the goal, and y is the deviation from the goal in terms of facing direction.

$$d = ||\Delta X_t||_2$$
$$y = tan^{-1}(\frac{y_G - y_t}{x_G - x_t}) - \omega_t$$

4.3 Regularization of Policy Network

4.3.1 Lipschitz continuity regularization constraint

Up until now, we have designed a network which can successfully serve as an end-to-end framework between raw sensor readings and actuations. But on closely examining the trained policy we inferred that it is too noisy for the real-world deployment. In Fig. 4.6, we can see that, the steering angle obtained from the policy function is quite noisy. For more perspective, we can the change in the steering angle with respect to time in Fig. 4.7.

So we devised a regularization approach based on Lipschitz continuity of the actor with respect to the feature vector. As mentioned in Section 3.2, we would regularize the trained network using the constraint mentioned in Equation 3.1. We incorporate this constraint by devising L_t^{LC} in Eq. 4.2.

As mentioned in [5] estimating the Lipschitz constant in Eq. 3.1 accurately is an NP-hard problem. It has been established in [3], that under the L2 norm, the Lipschitz constant of a fully connected layer is given by the spectral norm of the weight matrices. We also know for a matrix (M) with size m-by-n, its 2-norm is bounded by its ∞-norm in the following relation:

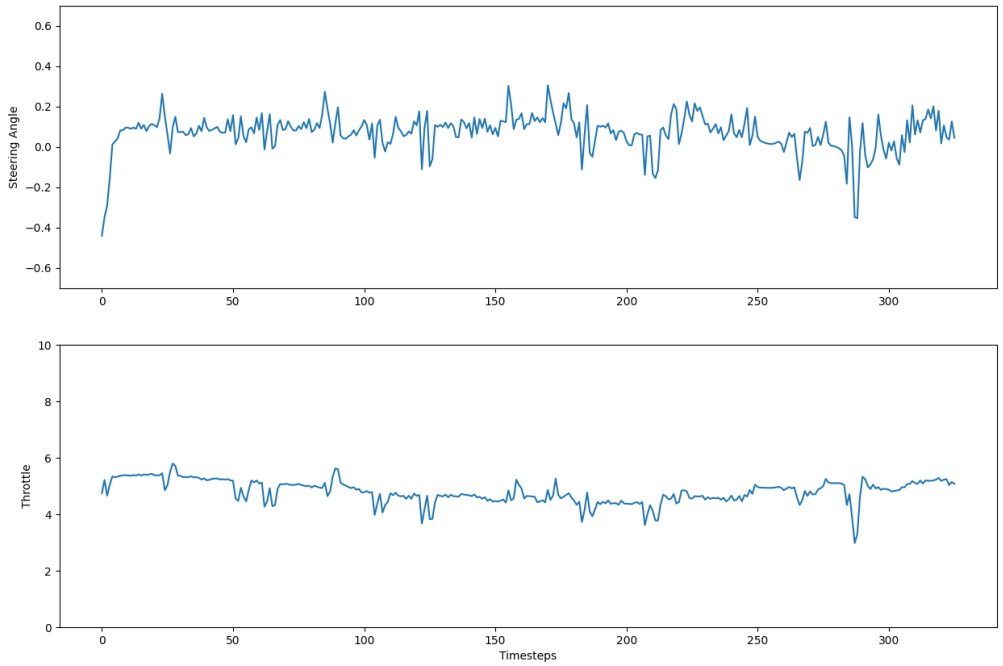


Figure 4.6. Policy of the agent without regularization

$$\frac{1}{\sqrt{n}}||M||_\infty \leq ||M||_2 \leq \sqrt{m}||M||_\infty$$

So, from above mentioned relation and [3] we can say that, for a MLP ($g(.)$),

$$\frac{||g(x_2) - g(x_1)||_\infty}{||x_2 - x_1||_\infty} \leq k\ \Pi||W_j||_\infty \tag{4.6}$$

where, x_2 and x_1 are inputs to the MLP, W_j represents weight matrix of j^{th} layer, $||.||_\infty$ represents p-norm, k is a constant depending upon $\sqrt{m}$.

4.3.2 Designing of regularization strategy

We will be designing a regularization expression based on inequality 4.6. Before deploying our regularization strategies on our agent we decided to experiment on a few toy problems.

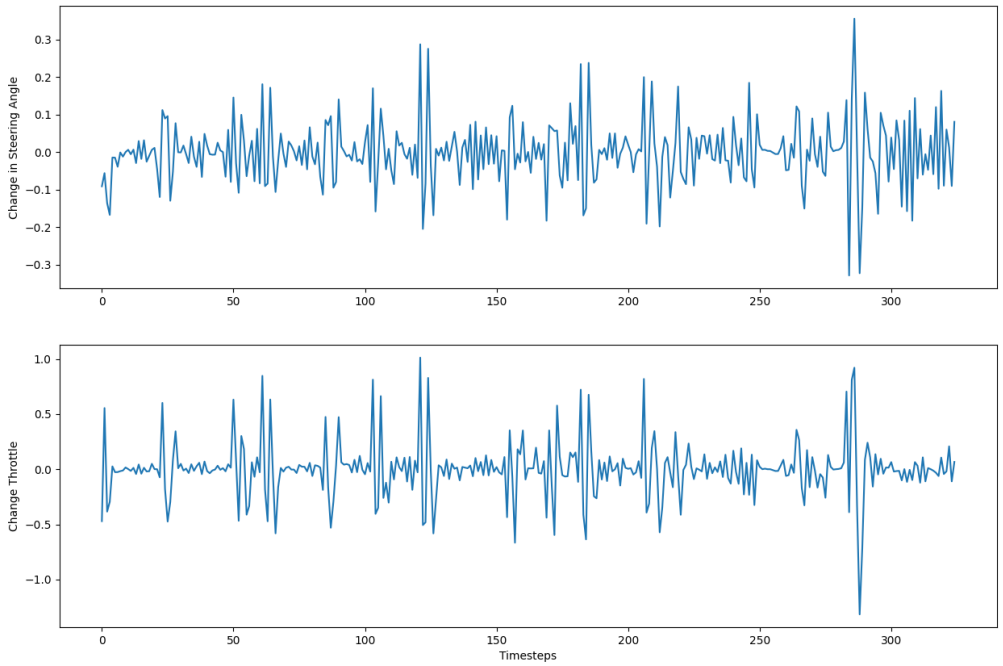


Figure 4.7. Change in policy of the agent without regularization

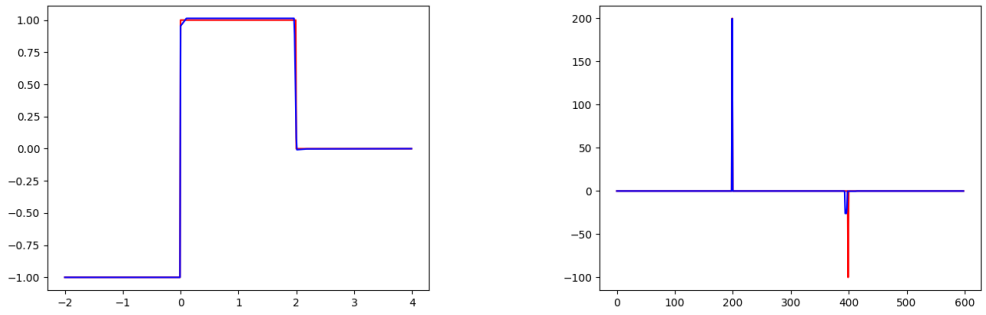
We consider 500 datapoints $([x_i, y_i])$ on the following functions,

$$y = g_1(x) = \begin{cases} -1 & x \in [-2, 0) \\ 1 & x \in [0, 2) \\ 0 & x \in [2, 4] \end{cases}$$

$$y = g_2(x) = \begin{cases} |x+2| & x \in [-3, -1) \\ |x| & x \in [-1, 1) \\ |x-2| & x \in [1, 3] \end{cases}$$

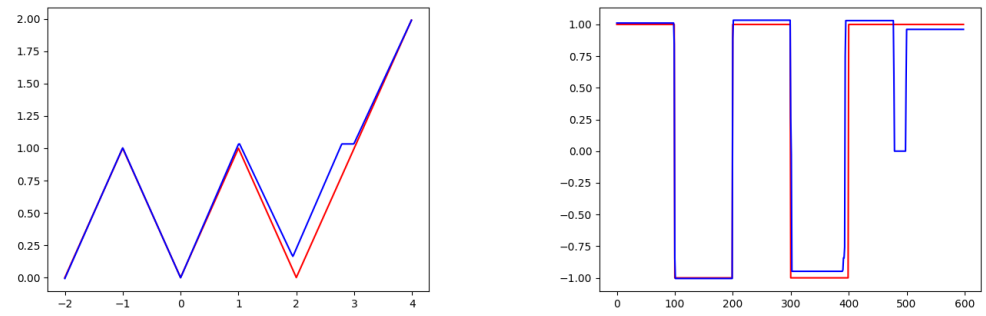
On plotting raw datapoints (represented by red curve in Fig. 4.8a and 4.9a), we can evidently see that these functions have high Lipschitz constants and we want to adopt an approach

which would work effectively on fully connected networks. We first train an MLP (denoted by $g_1(x)'$ and $g_2(x)'$ for $g_1(x)$ and $g_2(x)$ respectively) with 3 hidden layers containing 500 neurons each with ReLU activation function to overfit datapoints $g_1(x)$ and $g_2(x)$. Loss function used to train the MLP is the MSE loss (denoted by L). Curves after overfitting the devised MLP using the datapoints are shown in Fig. 4.8a and 4.9a represented by blue curve. Gradients of the overfitted MLPs are shown in Fig. 4.8b and 4.9b.



(a) Overfit using MLP **(b)** Gradient of the fit

Figure 4.8. Overfit and gradient of $g_1(x)$ respectively



(a) Overfit using MLP **(b)** Gradient of the fit

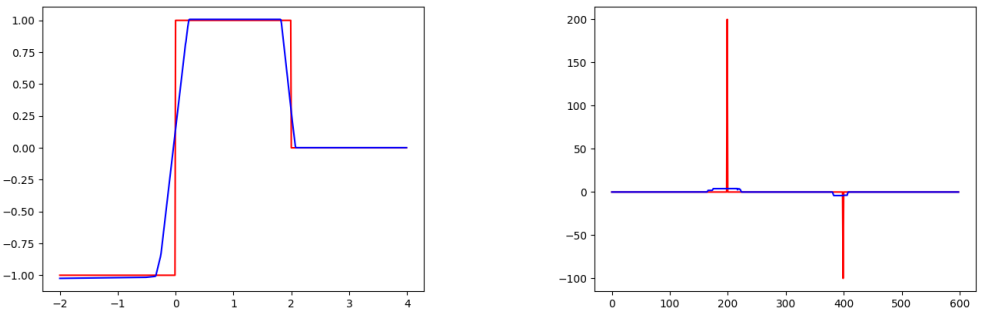
Figure 4.9. Overfit and gradient of $g_2(x)$ respectively

Since, for a 1-dimensional function Lipschitz constant will be effectively the gradient of the function. So, using inequality 4.6, we devise following regularization strategies,

$$r_1 = \max(||g(x_2) - g(x_1)||_\infty - \lambda||x_2 - x_1||_\infty,\ 0) \tag{4.7}$$

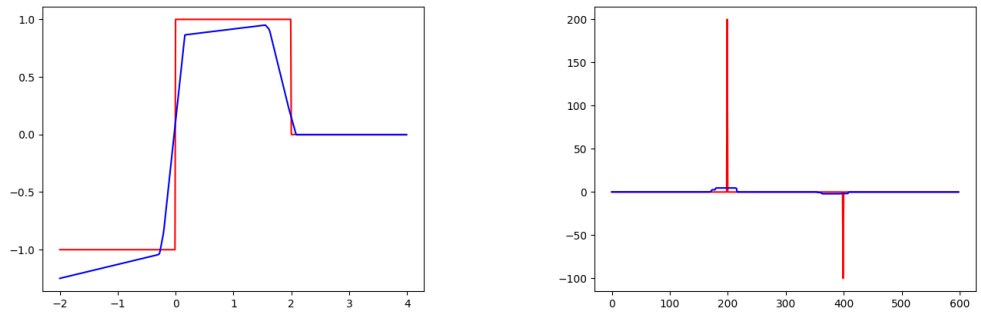
$$r_2 = \lambda(||g(x_2) - g(x_1)||_\infty - ||x_2 - x_1||_\infty \Pi ||W_j||_\infty) \tag{4.8}$$

where, λ represents regularization constant determining the impact of r_1 or r_2. Essentially, the loss function considering the r_1 or r_2 in the MLP, $g_1(x)'$ or $g_2(x)'$ is given by $L + r_1$ or $L + r_2$ respectively. So, the curves representing the fit of MLP on trained data are given by:



(a) Regularized fit **(b)** Regularized gradient of $g_1(x)'$

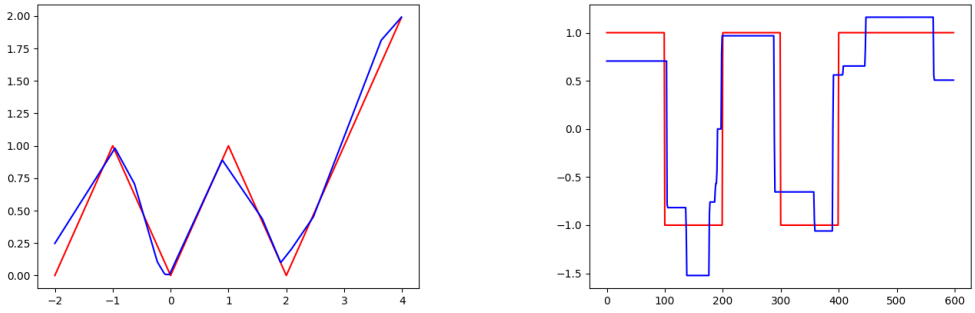
Figure 4.10. Regularized fit and gradient of $g_1(x)'$ using r_1



(a) Regularized fit **(b)** Regularized gradient of $g_1(x)'$

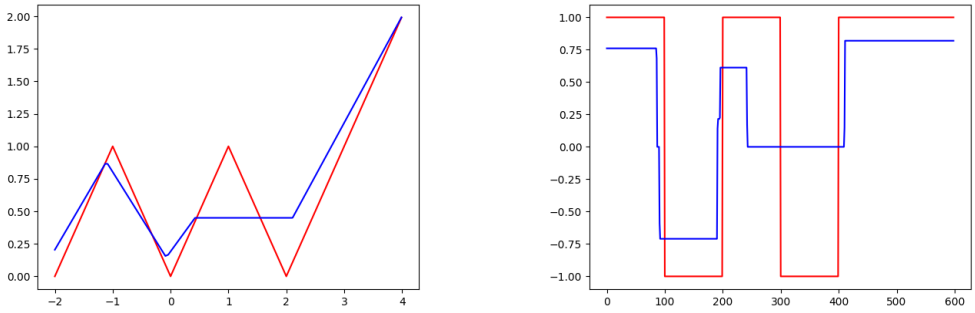
Figure 4.11. Regularized fit and gradient of $g_1(x)'$ using r_2

From, the curves we can see that r_2 reduces the Lipschitz constant more effectively, while slightly affecting the underlying nature of the perceptron. Given the nature of noise present in



(a) Regularized fit **(b)** Regularized gradient of $g_2(x)'$

Figure 4.12. Regularized fit and gradient of $g_2(x)'$ using r_1



(a) Regularized fit **(b)** Regularized gradient of $g_2(x)'$

Figure 4.13. Regularized fit and gradient of $g_2(x)'$ using r_2

our policy we decided to proceed with a regularization approach based on r_2

4.3.3 Regularization Of Actor Network

From Sec. 4.3.2, we can infer that Eq. 4.8 effectively constrained the Lipschitz constant of an MLP. So, we will be devising the regularization of out policy network, L_t^{LC} according to Eq. 4.8. Hence, we regularize the policy network π_θ, with respect to feature vectors, $h(S_{t+1})$ and $h(S_t)$ extracted from consecutive states, $S_{t+1}, S_t \in S$ according to Eq. 4.9.

$$L_t^{LC} = (||\pi_\theta(h(S_t+1)) - \pi_\theta(h(S_t))||_\infty - \Pi||W_j||_\infty \, ||h(S_{t+1}) - h(S_t)||_\infty) \tag{4.9}$$

where, W_j represents the weights of the actor network in j^{th} layer.

Chapter 5

Evaluation

In Chapter 4, we discussed various approaches to design the policy function for solving our problem. Now we will be evaluating the designed policy functions with respect to different feature extractors used on the basis of the criteria mentioned in the following sections. Evaluation of the policy was conducted in virtual as well as real-world setting. Due to the nature of our formulated problem (since it includes obstacle avoidance), for the safety purposes of the F1-tenth racecar (Fig. 5.17), we decided to train the model in simulation and deployed the model with optimal performance on the vehicle after ensuring we have a stable policy in simulation.

5.1 Virtual Experiments

We have considered two simulation engines - PyBullet and MuJoCo, for our experiments in a virtual setup. We chose PyBullet for its ease of defining system dynamics and low computational requirements. When an algorithm produced promising results in PyBullet, we moved onto a much more photorealistic simulation engine, which here is MuJoCo. This was done to ensure we have as much low discrepency in the visual data as possible compared to depth camera used in the F1-tenth car. One other major reason for using MuJoCo was due to its superior physics engine. In the following sections we walk through the experimental setup in the simulation, followed by comparison of different networks' performance on our evaluation criterias.

5.1.1 Environment Setup

Training

The environment setup in both simulation engines is identical (for training as well as for evaluation). Initial position of the agent being the origin in the world coordinates. The environment is modeled as a planar arena with obstacles and goal, spawning at random locations at the beginning of each episode. Goal ($[x_G, y_G]$) is spawned at random with bounds defined as $x_G \in [-15, 15]$ and $y_G \in [-15, 15]$. Since, the problem is formulated as first-exit finite horizon markov problem, the termination criteria of the episodes is either agent crashing into the obstacle or reaching the goal, with the finite horizon being 1500 timesteps. Fig. 5.1 and 5.2 represent MuJoCo and PyBullet environments respectively. In Fig. 5.1, red spheres represents the obstacles and yellow spheres represent the pre-defined point goal. In Fig. 5.2, red patch on the arena represents the point goal whereas blue spheres represents the obstacles to be avoided. Pybullet environment enables us to visualize RGB camera data, depth camera data and also gives a semantic segmentation map as shown in the left column in Fig. 5.2. But we are only concerned about the depth images since according to formulated problem in Sec. 3.

Evaluation

We will be evaluating the performance of the trained networks based on the following criterias:

1. Since, an episode will be terminated if agent hits an obstacle, we record the number of episodes ended in collision with the obstacle among all the evaluation episodes.

2. Number of episodes successfully completed by the agent i.e., it reaches the randomly defined point goal.

3. Average deviation from the shortest possible path to the goals, ignoring the obstacles.

Evaluation criteria (1) and (2) give us the success ratio (quantitive evaluation) of a

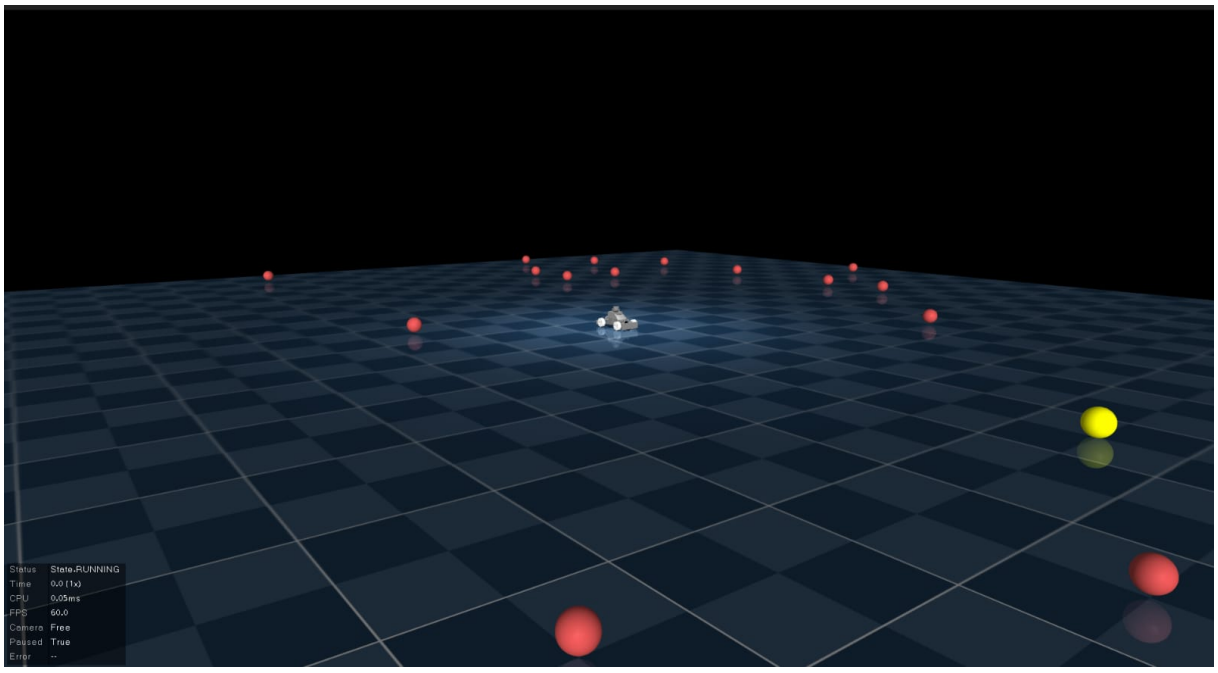


Figure 5.1. MuJoCo Environment

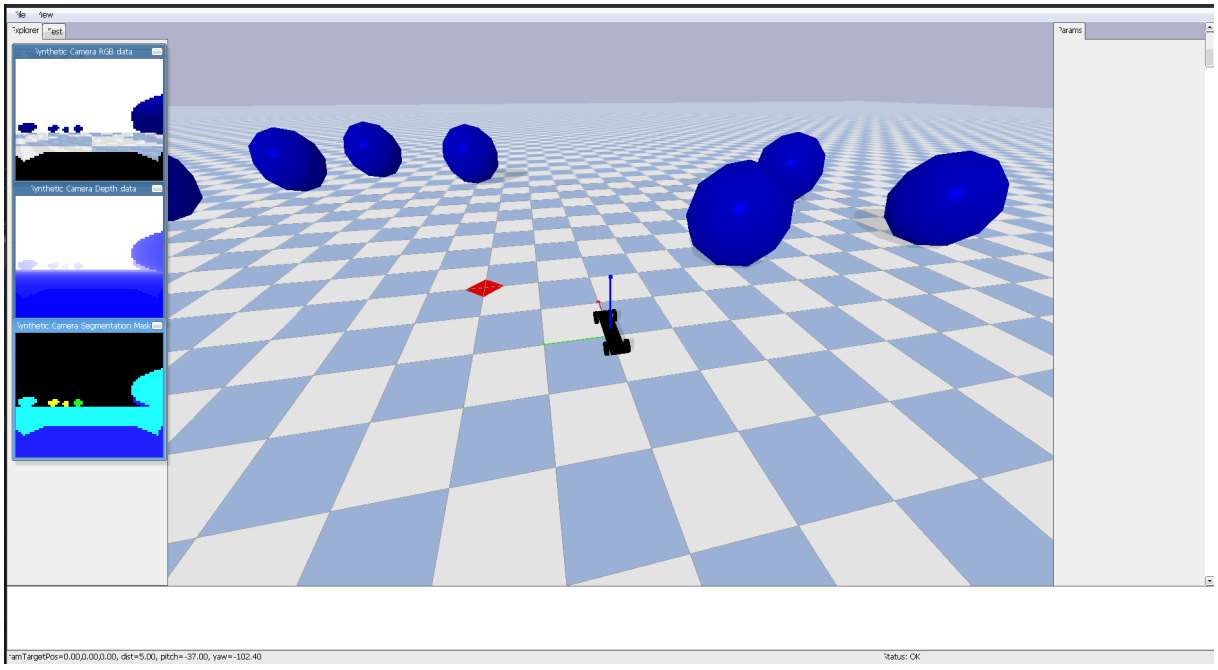


Figure 5.2. PyBullet Environment

particular model with respect to formulated task. Whereas, criteria (3) provides a qualitative evaluation of the successfully trained model i.e., how well a trained policy performs for real-

world deployment. Qualitative evaluation gives us certainity to deploy the policy on the vehicle safely and efficiently. We will be using 100 evaluation episodes for each of the trained policy.

5.1.2 Evaluation on the basis of Feature Extractors

As mentioned in above section, we have designed four feature extractors: State-Only, State-Depth-Concatenation, and lastly Vision-Transformer. All of the feature extractors except State-Only, are trained and evaluated in identical environment setup. State-Only feature extractor was trained and evaluated in an obstacle void environment, as obstacle avoidance is not expected from this model due to lack of any visual input.

State-Only Feature Extractor

Training: The environment used for training and evaluation in this scenario would be devoid of obstacles leaving the rest of the setup as is. The hyperparameters used for training the network comprising of State-Only feature extractor are mentioned below in the table. We have kept the architecture of actor and critic identical to rest to the one in rest of the instances for uniformity. The hyperparameters used for training the loss function Eq. 4.2 are mentioned in Table 5.1. Initially we trained the state-only feature extractor without constraining the actor to be Lipschitz continuous, and got a stable policy. Hence, $c_3 = 0$.

Table 5.1. PPO Configuration Parameters (State-Only Feature Extractor)

Parameter	Value
H	1500
No. of Envs.	4
Learning Rate	3×10^{-5}
No. of Epochs	50
Batch Size	300
ε	0.2
c_1	1.5
c_2	0.05
c_3	0
σ_{init}	1.01
T	800000

We trained the policy with state-only feature extractor using both Gaussian and Beta distribution. The normalized rewards obtained by the agent throughout the training are shown in Fig. 5.3 and 5.4. From the plot it is safe to say that the loss function has reached a minima.

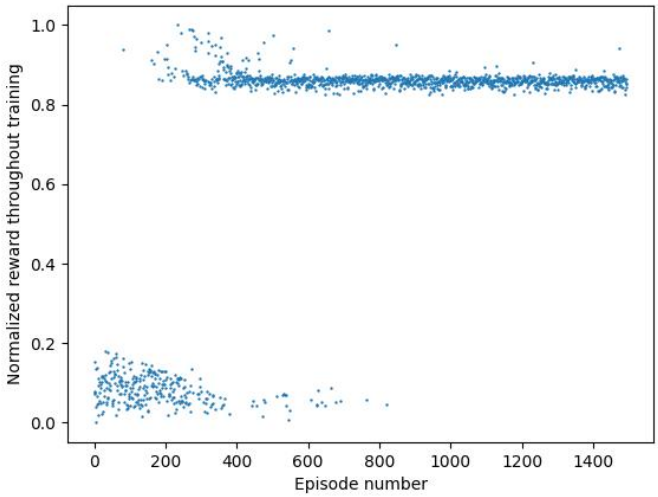


Figure 5.3. Normalized Reward for State-Only Feature Extractor (Gaussian)

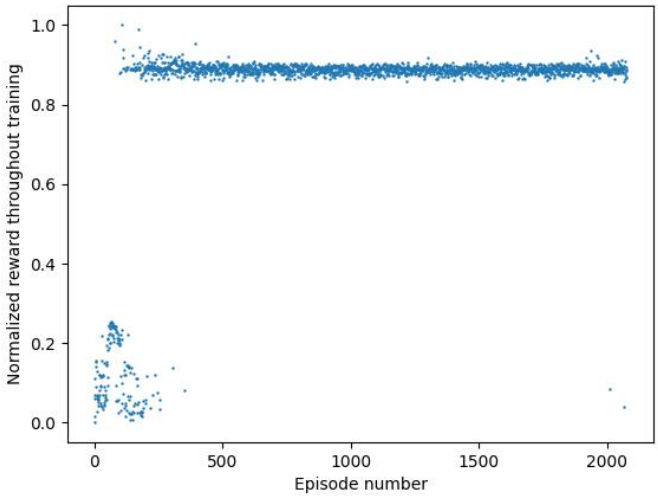


Figure 5.4. Normalized Reward for State-Only Feature Extractor (Beta)

Evaluation:

Table 5.2 represents the results obtained from 100 evaluation episodes. Fig. 5.6 and 5.5

represents the controls provided by actor while evaluation. Here the agent the tasked to reached point goal defined at $[x_G, y_G] = [14, 10]$.

Table 5.2. Evaluation of Trained Policy with State-Only Feature Extractor

100 Eval Episodes	Policy as Normal Distribution	Policy as Beta Distribution
Number of collisions	-	-
Average Deviation from shortest path	0.89	1.28
Completion of task	88	97

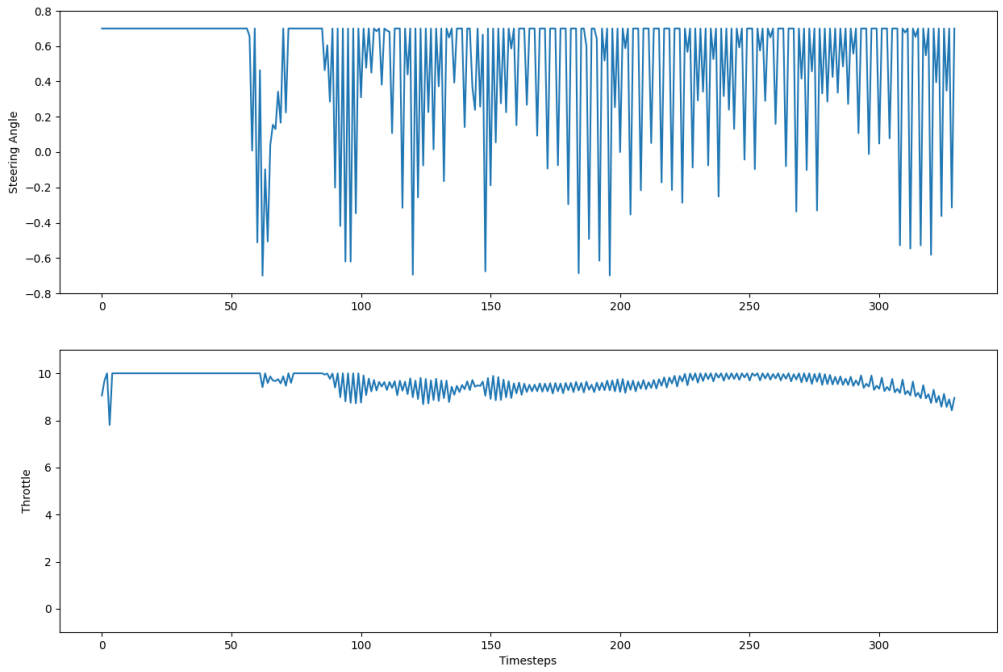


Figure 5.5. Controls from the policy using Gaussian distribution

Inference:

From Table 5.2 we can see that agent was able to complete the task with a high success rate. This was due to simplicity of the environment (obstacle devoid arena). Fig. 5.5 shows controls taken by the agent using policy designed as a normal distribution for reaching the goal defined at $[x_G, y_G] = [14, 10]$. As we can see the trained policy in this scenario is quite noisy (or oscillating) due to the bounds present in the action space which affected the training, making the

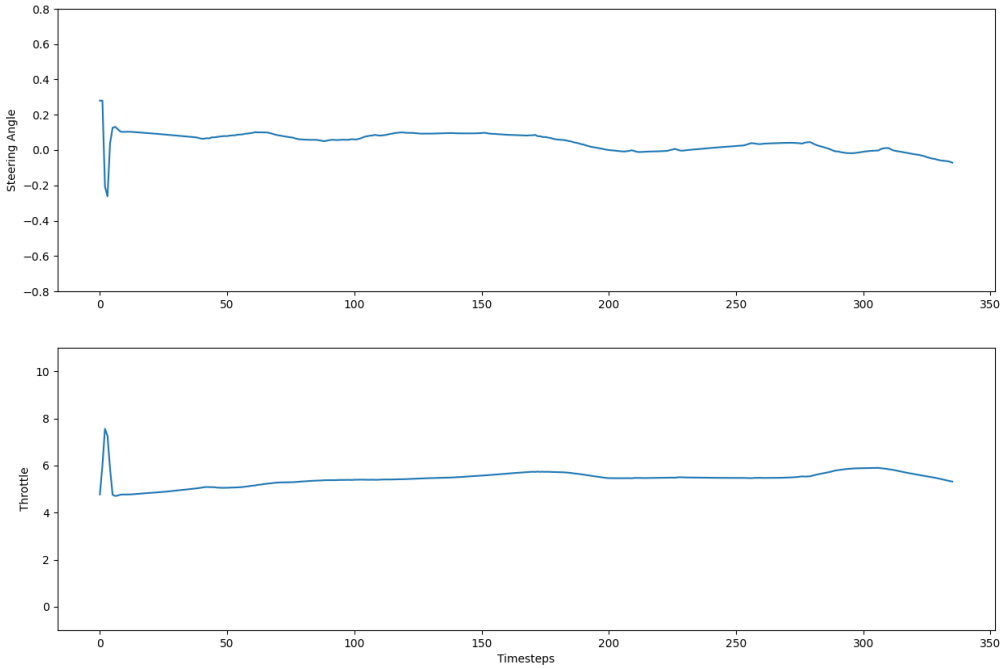


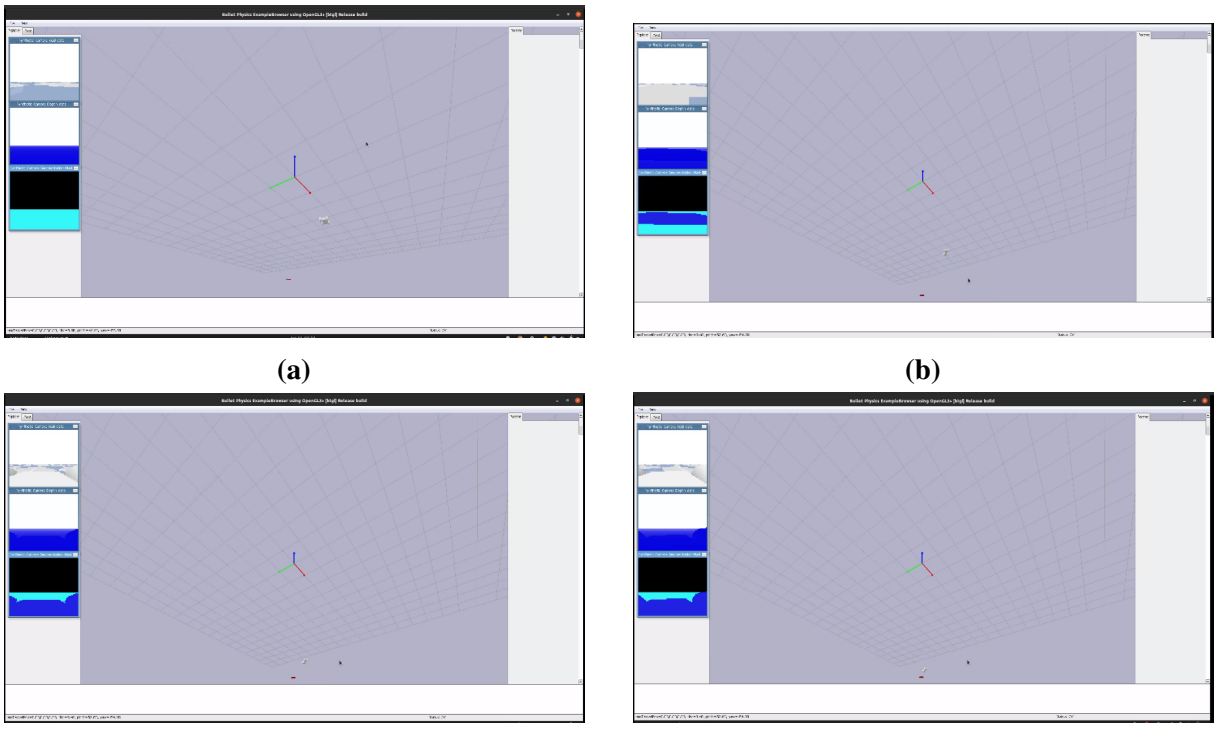
Figure 5.6. Controls from the policy using Beta distribution

resultant policy biased towards the extreme actions. Hence, we decided to model the distribution as Beta distribution which provides us with quite stable policy for identical evaluation episode as shown in Fig. 5.6. From Table 5.2, we can also observe that average deviation of the agent from shortest possible path is higher when using Beta distribution. This is due to the fact that agent with policy modeled takes a smooth path rather than oscillating about the shortest path which seemed to be the case in policy modeled as Normal distribution.

Fig. 5.7 (Path: 5.7a $\rightarrow$ 5.7b $\rightarrow$ 5.7c $\rightarrow$ 5.7d) shows the trajectory taken by the agent, implementing policy with state-only feature modeled as Beta distribution.

State-Depth-Concatenation Feature Extractor

Training: As mentioned earlier, the environment setup would be identical for State-Depth-Concatenation, and Vision-Transformer. The hyperparameters used for training the loss



(a) **(b)**

(c) **(d)**

Figure 5.7. Trajectory of agent navigating to $[14, 10]$

function Eq. 4.2 are mentioned in Table 5.3. In this part we will also analyse the effect of applying the regularization strategy formulated in Eq. 4.6.

Evaluation:

Table 5.4 represents the results obtained from 100 evaluation episodes. Fig. 5.8, 5.9, and 5.10 represents the controls provided by actor while evaluation. Fig. 5.12 and 5.11 represents change in actions with respect to time (agent is tasked to reached point goal defined at $[x_G, y_G] = [14, 10]$).

Inference:

From Table 5.4, we can see that success rate for the trained policy (modeled as Beta distribution) is quite low. This is because we have constrained the action space, and the environment used for training as well as evalution was cluttered. So, agent is not biased to implement controls which are only at the extremes as in the case of policy devised as Gaussian distribution. Higher number of collisions were observed in setup where policy was being regularized. This was one

Table 5.3. PPO Configuration Parameters (State-Depth Feature Extractor)

Parameter	**Value**
H	1500
No. of Envs.	4
Learning Rate	3×10^{-5}
No. of Epochs	50
Batch Size	300
ε	0.2
c_1	1.5
c_2	0.05
c_3	100
T	1200000

Table 5.4. Evaluation of Trained Policy with State-Depth Feature Extractor

100 Eval Episodes	Collisions	Avg. Deviation	Success
Policy with Gaussian distribution	11	6.41	51
Policy w/o regularization	13	2.96	33
Policy with regularization	20	1.51	25

of the major effect observed on constraining the policy. As the agent is not allowed to implement drastic changes in the controls, it is unable to evade the obstacle through narrow spaces. We can also observe that success rate of the Gaussian policy is higher compared to policy modeled as Beta distribution. This could be due to extreme controls taken by the agent, as it will have an improved manueverability to navigate in a cluttered environment. Higher deviation from shortest path in the case of policy devised as gaussian distribution is also a result of biased policy. It was observed in few evaluation that agent navigates to goal indirectly. Whereas, in other two cases, due to lack of implementation of extreme controls, once it deviates too much (and too far) from the direction of goal, it unable to correct itself ending the episode in failure.

Fig. 5.13 (Path: 5.13a $\rightarrow$ 5.13b $\rightarrow$ 5.13c $\rightarrow$ 5.13d) shows the trajectory taken by the agent, implementing policy with state-depth feature modeled as Beta distribution.

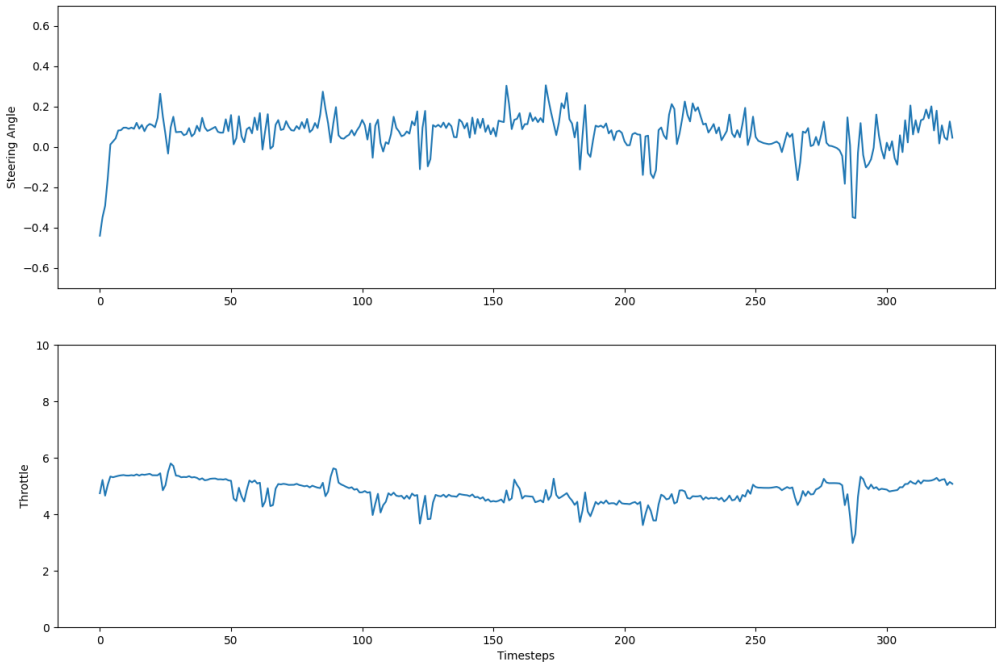


Figure 5.8. Controls from the policy without regularization

Vision-Transformer Feature Extractor

Training: The hyperparameters used for training the loss function Eq. 4.2 are mentioned in Table 5.5. We have modeled the policy as a Gaussian distribution as this was the only distribution being considered in the initial stages of this work. Since, we did not observe an improvement in the performance compared to State-Depth-Concatenation feature extractor we decided not to proceed with Vision Transformer feature extractor due to its high computational requirements.

As mentioned in Sec. 4.1.3, vision Transformer was trained in a sequential manner. Starting from a simple environment with no obstacle to a complex environment with random spawning of goal and obstacles. We were able to observe partial convergence of the reward after dedicating intensive computational resources. Fig. 5.14 shows the reward in the final iteration of sequential training i.e., with random spawning of goal and obstacles.

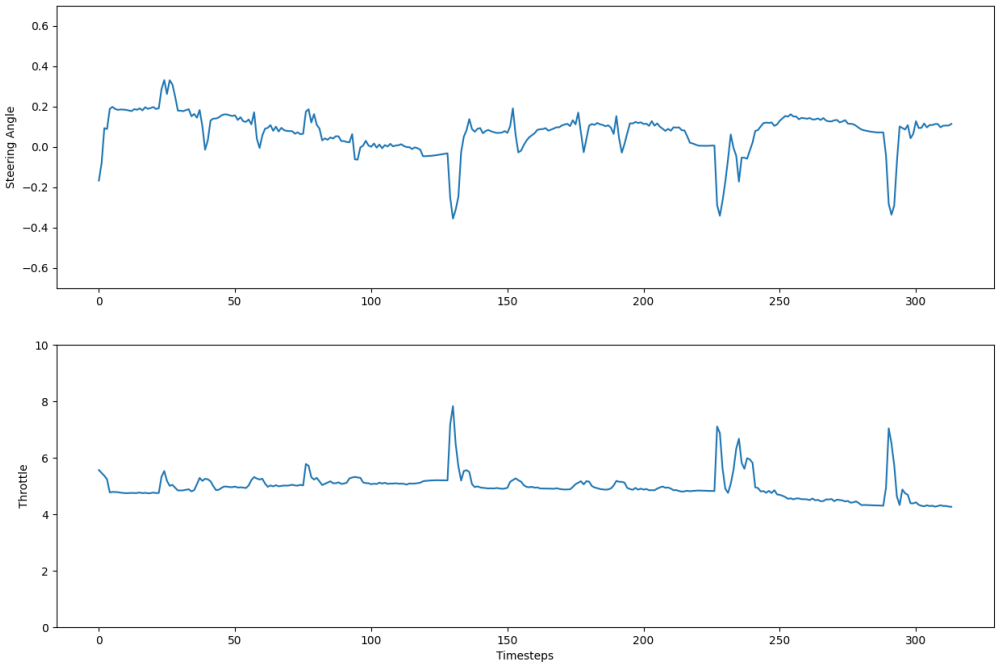


Figure 5.9. Controls from the policy using regularization

Evaluation:

Table 5.6 shows the evaluations on the policy trained using Vision-Transformer Feature Extractor. Ideally the number of collisions in the policy using Vision-Transformer feature extractor should be least compared to all of the trained policies as it correlates the patches of images. Increase in number of collisions could be due to the fact that loss function of model is not converged (evident from Fig. 5.14). Fig. 5.15 shows the controls taken by agent to navigate to goal coordinates $[14, 10]$.

Fig. 5.16 (Path: 5.16a $\rightarrow$ 5.16b $\rightarrow$ 5.16c $\rightarrow$ 5.16d) shows the trajectory taken by the agent, implementing policy with Vision Transformer feature extractor modeled as Gaussian distribution.

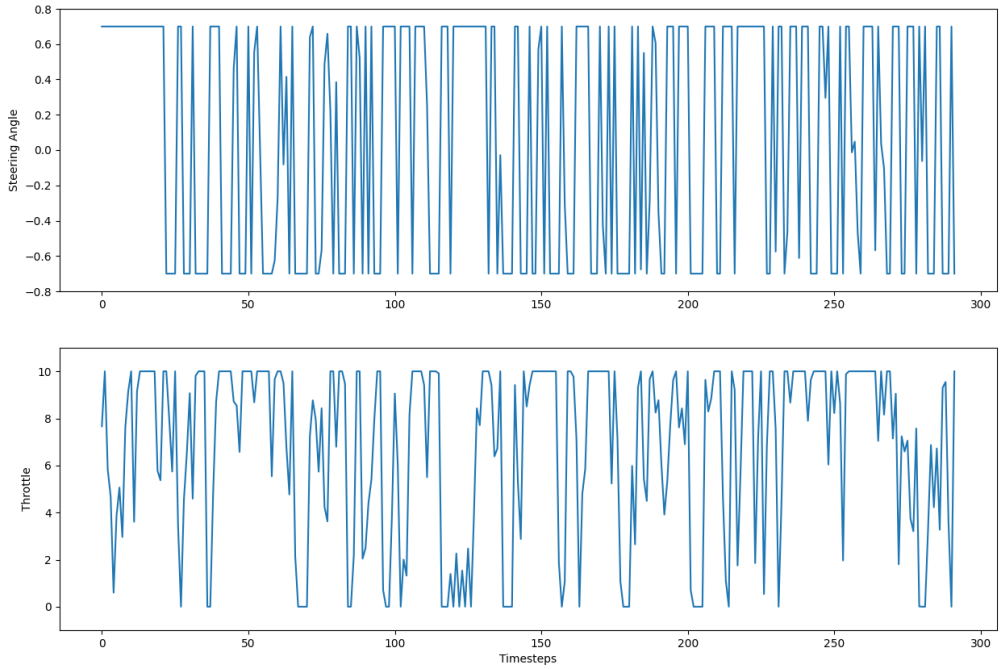


Figure 5.10. Controls from policy when modeled as gaussian distribution

5.2 Real-World Experiments

After obtaining satisfactory results in simulation, we decided to deploy the model with best performance onto a F1-tenth car, to conduct similar experiments in unseen real world environment. We deployed the network trained in the simulation on the car while scaling the output of the network according to bounds of the actuators of the car. The following section mentions the hardware and software specifications along with experimental setup.

5.2.1 Hardware

To evaluate the performance of the trained RL model on a real-world Ackermann drive car, we employed an F1-tenth scale vehicle as our physical platform (Fig. 5.17). This vehicle is equipped with hardware components essential for autonomous navigation, including an NVIDIA Jetson TX2 embedded computer with CUDA support, a VESC (as a motor controller), and an

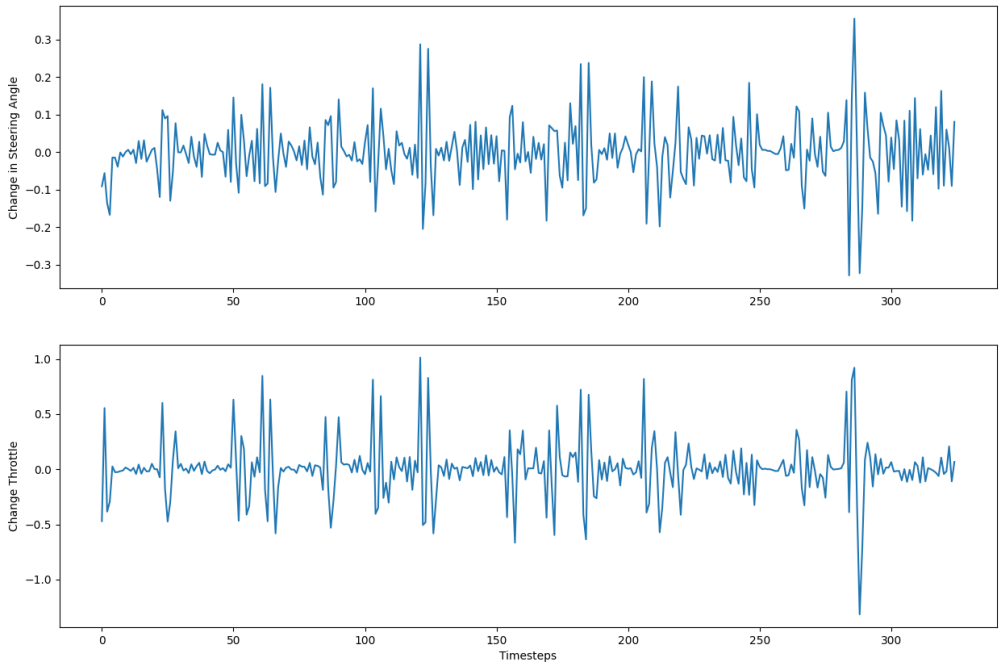


Figure 5.11. Change in controls w.r.t. time

Intel RealSense (D435) RGBD camera. Maximum permitted velocity of the car was 0.2 m/s for safety purposes.

5.2.2 Software

Our software stack is built on the foundation of ROS Melodic and runs on an Ubuntu 18.04 platform. ROS (Robot Operating System) provides a modular and flexible framework for managing robot hardware and software components, making it an ideal choice for our project. Model inference is responsible by TensorRT SDK from Nvidia to accelerate the inference of the neural network.

The ROS control node is a key component that integrates sensory data and the RL model for vehicle navigation. It subscribes to depth images and Ackermann car states, combining this information to feed into the trained PPO model. This model then computes the necessary steering

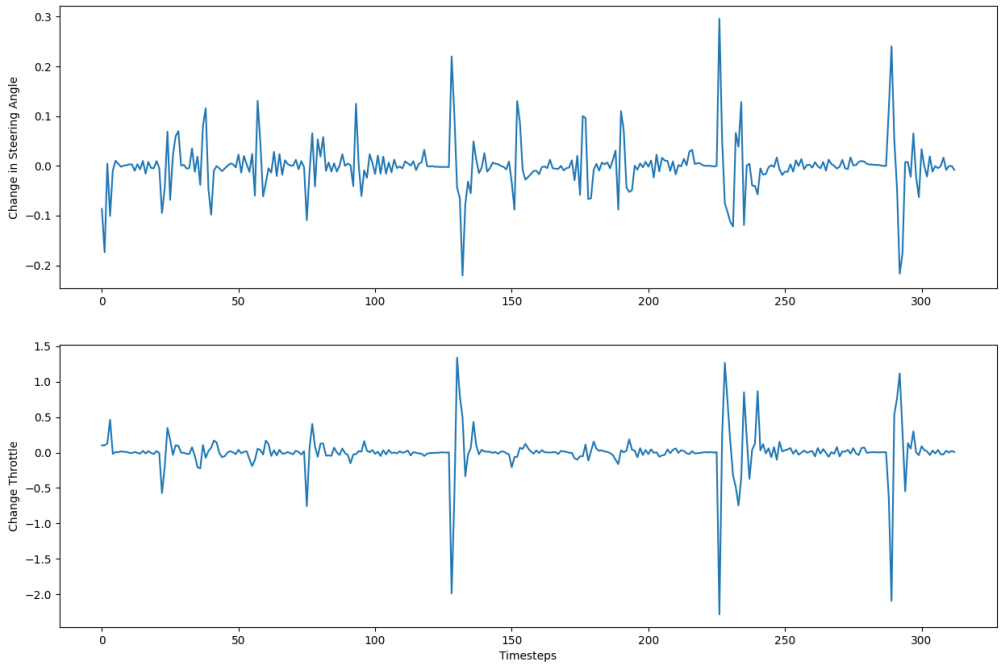


Figure 5.12. Change in controls w.r.t. time

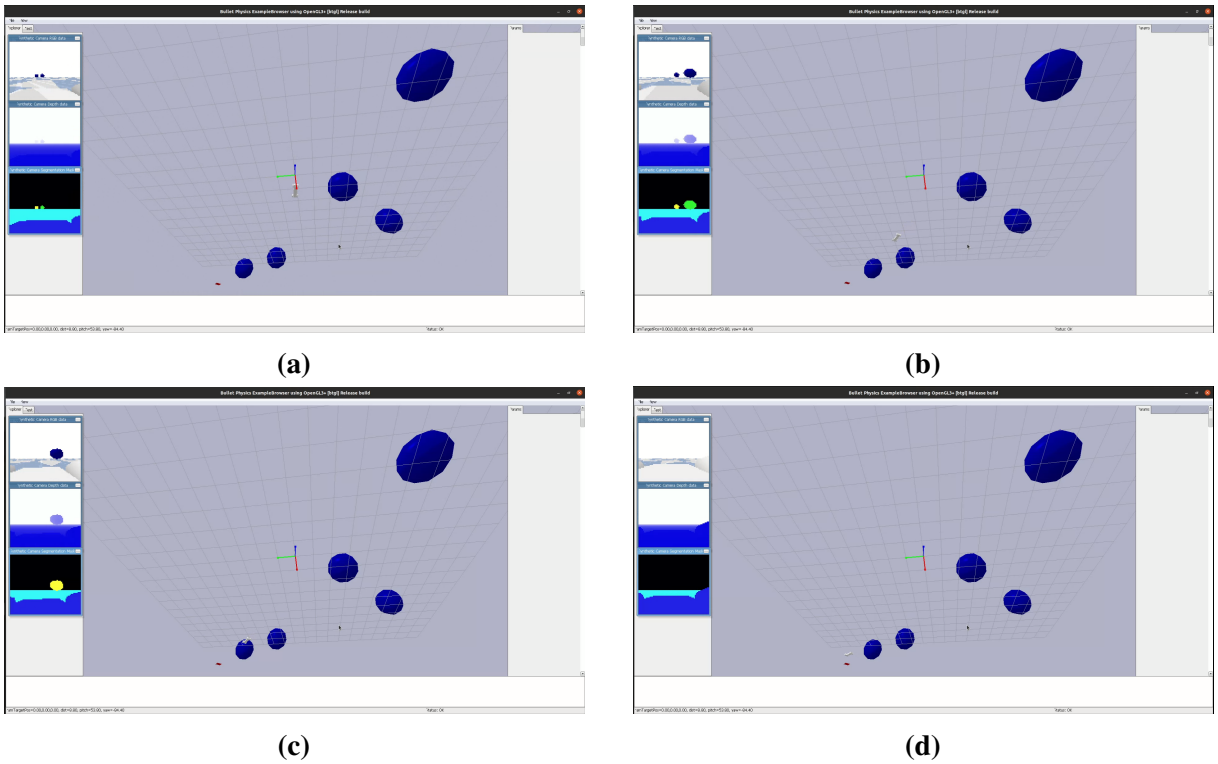
angle and heading velocity for navigation. The output is published to the VESC controller, translating the model's decisions into actual vehicle movement, ensuring precise and adaptive control in real-world driving scenarios.

Low-level control of the motors is handled by nodes publishing to the motor controller which here is the VESC. We are providing the input signal in terms of heading velocity and Ackermann steering as discussed above to the VESC ROS Nodes after processing the raw sensor data from three different nodes (Three parts of the Neural Network).

The control node subscribes to two essential types of data from the vehicle's sensors:

- Car States (VESC feedback): These include critical information about the vehicle's odometry and history. This data is crucial for the RL agent to understand the current state of the vehicle, including its position, velocity, and orientation.

- Depth Images: We utilize the Intel RealSense RGBD camera to capture depth images of



(a) **(b)**

(c) **(d)**

Figure 5.13. Trajectory of agent navigating to $[14, 10]$ in obstacle-rich environment

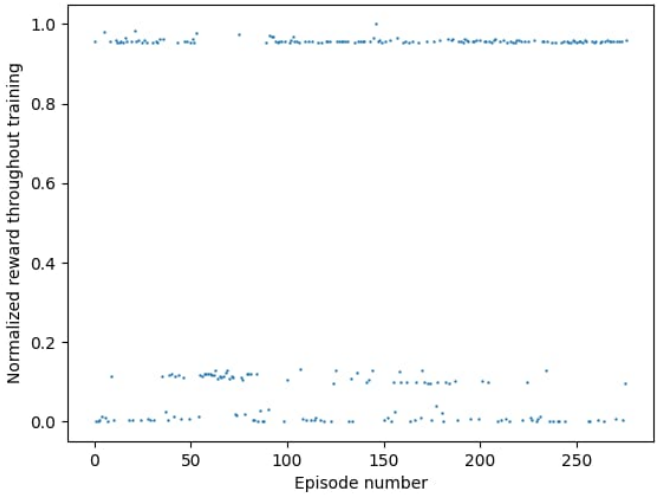


Figure 5.14. Normalized rewards for training Vision-Transformer Feature Extractor

the surroundings. These images provide crucial environmental information, helping the RL agent to perceive obstacles and make informed decisions.

Table 5.5. PPO Configuration Parameters (Vision-Transformer Feature Extractor)

Parameter	Value
H	1500
No. of Envs.	4
Learning Rate	3×10^{-5}
No. of Epochs	50
Batch Size	300
ε	0.2
c_1	1.5
c_2	0.05
σ_{init}	1.01
T	1000000

Table 5.6. Evaluation of Trained Policy with Vision-Transformer Feature Extractor

100 Eval Episodes	Collisions	Avg. Deviation	Success
Policy with Gaussian distribution	18	1.64	56

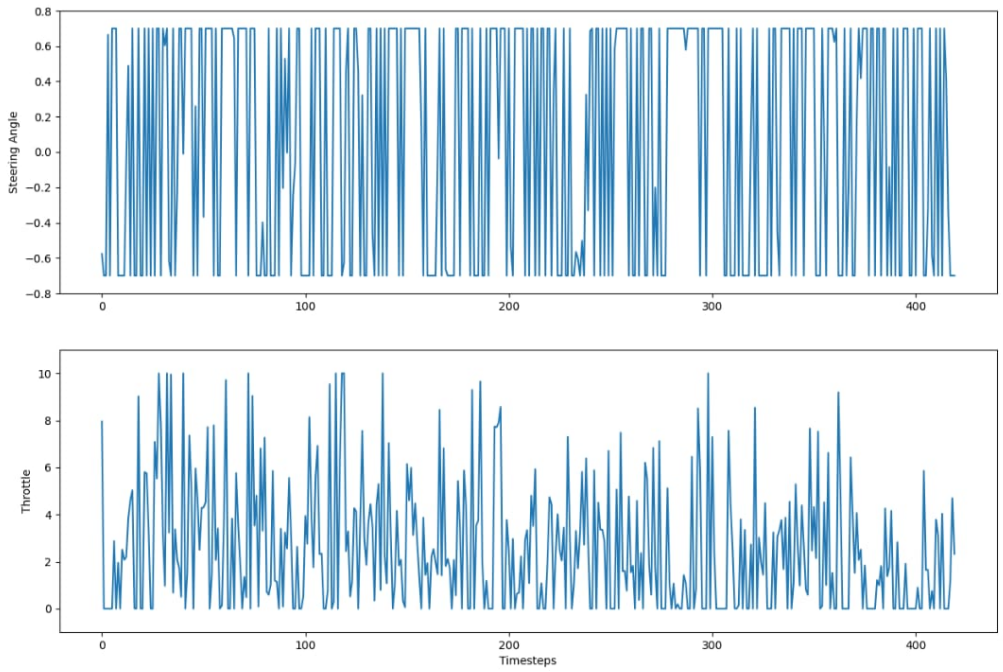
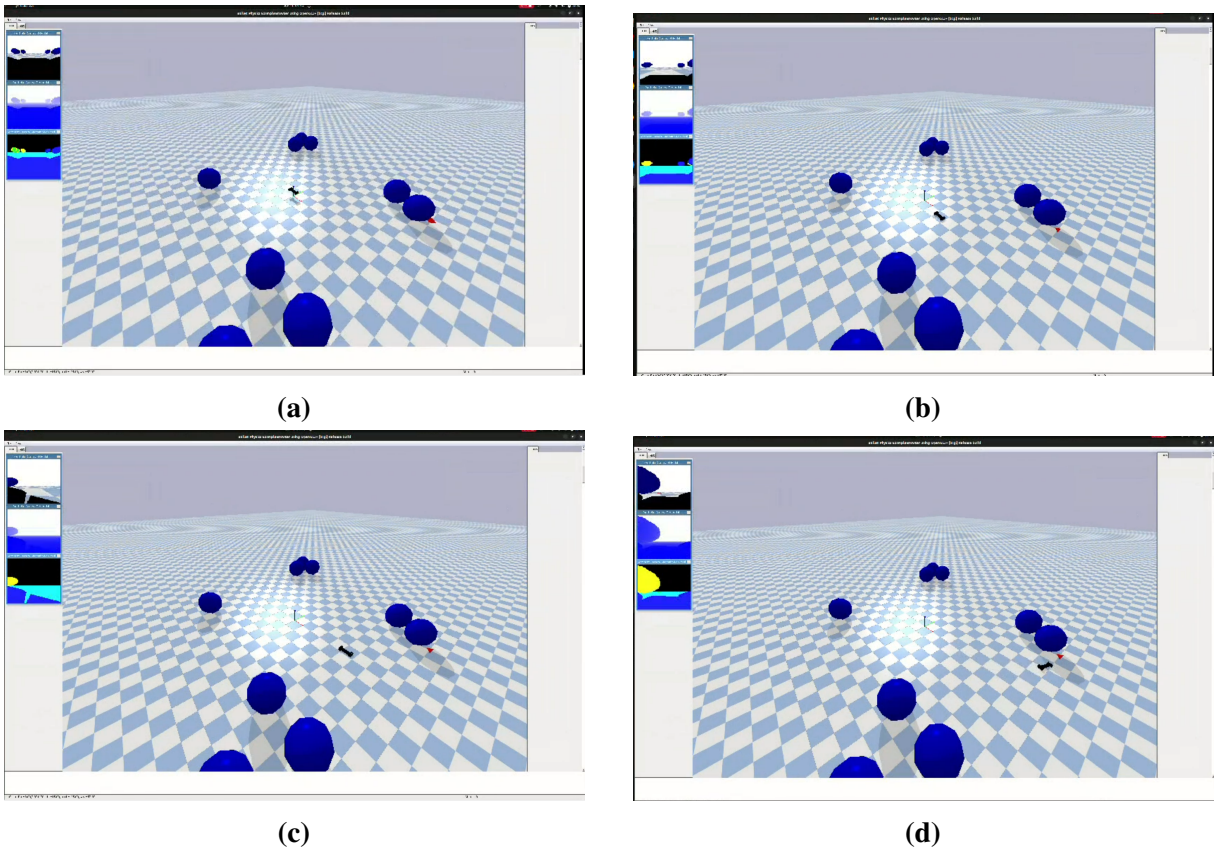


Figure 5.15. Controls from policy using Vision-Transformer Feature Extractor



(a) **(b)**

(c) **(d)**

Figure 5.16. Trajectory of agent navigating to $[14, 10]$ using Vision-Transformer Feature Extractor

5.2.3 Experiments

On analysing the results in Sec. 5.1, we decided to deploy the policy with State-Only and State-Depth concatenate feature extractor, modeled as Beta distribution. Since, rest of the trained policies are too noisy for real-world deployment although in some cases they yielded better results. Axes in the frame of car is aligned in identical manner as done in PyBullet simulation (heading direction representing x-axis and top direction representing z-axis). Experiments were conducted in the open space in the room 3301 of Franklin Antonio Hall, UC San Diego.

State-Only Feature Extractor

In the first evalution episode, coordinates of goal ($\mathbf{x}_G$) are defined as $\mathbf{x}_G = [2, 0]$, in meters w.r.t. the car. The trajectory followed by the car is shown in Fig. 5.18 (Path: 5.18a $\rightarrow$ 5.18b $\rightarrow$

Figure 5.17. F1-tenth car used in real world experiments

5.18c $\rightarrow$ 5.18d).

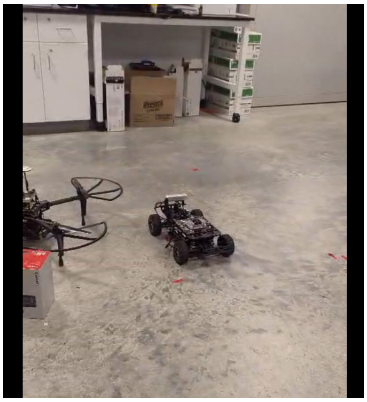
The car was successfully able to navigate to the defined goal. In the second evalution episode, coordinates of goal are defined as $\mathbf{x}_G = [1, 1]$, in meters w.r.t. the car. The trajectory followed by the car is shown in Fig. 5.19. (Path: 5.19a $\rightarrow$ 5.19b $\rightarrow$ 5.19c $\rightarrow$ 5.19d)

State-Depth Feature Extractor

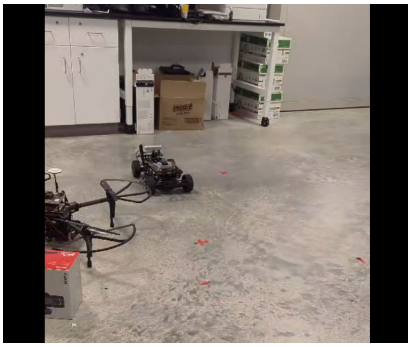
In the evaluation episode mentioned in Fig. 5.20, car was tasked to reach the goal defined as $\mathbf{x}_G = [2, 0]$. We can see that agent was able to maneuver away from the box (obstacle). But when an episode was implemented to navigate to the same goal without any obstacle. It implemented an undefined policy. We believe the possible reason for failure of this implementation could be noisy depth image data. Surrounding objects could be interfering with the agents depth images. As seen in Fig. 5.1 and 5.2, the obstacle space from the view of depth camera is quite sparse as compared to real world experiments. We believe this problem could be solved by training the feature extractor using real-world depth images along with depth images in

(a)

(b)

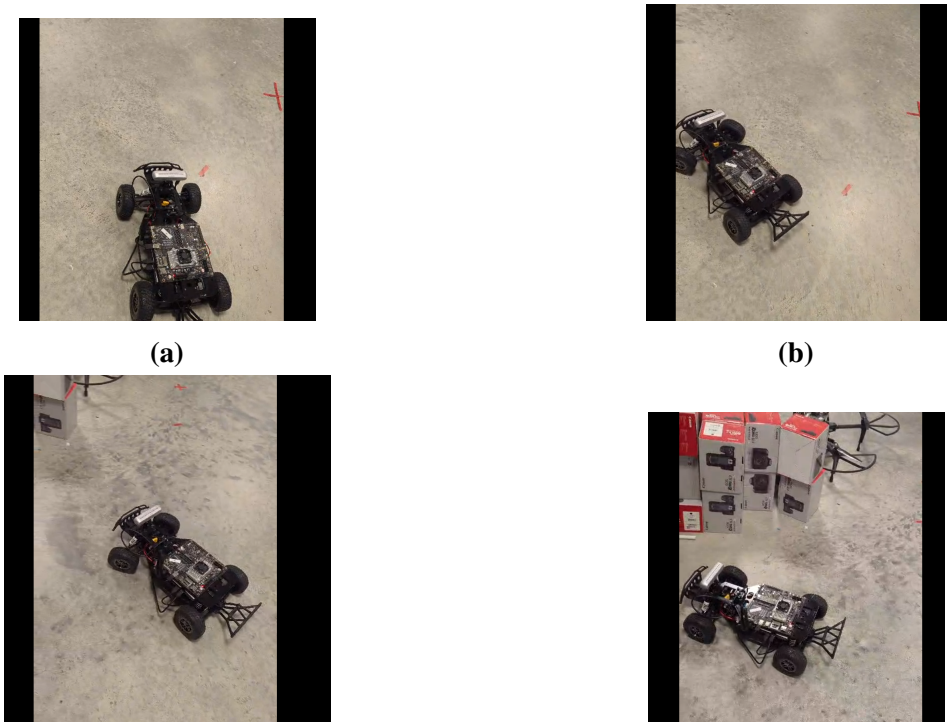


(c)



(d)

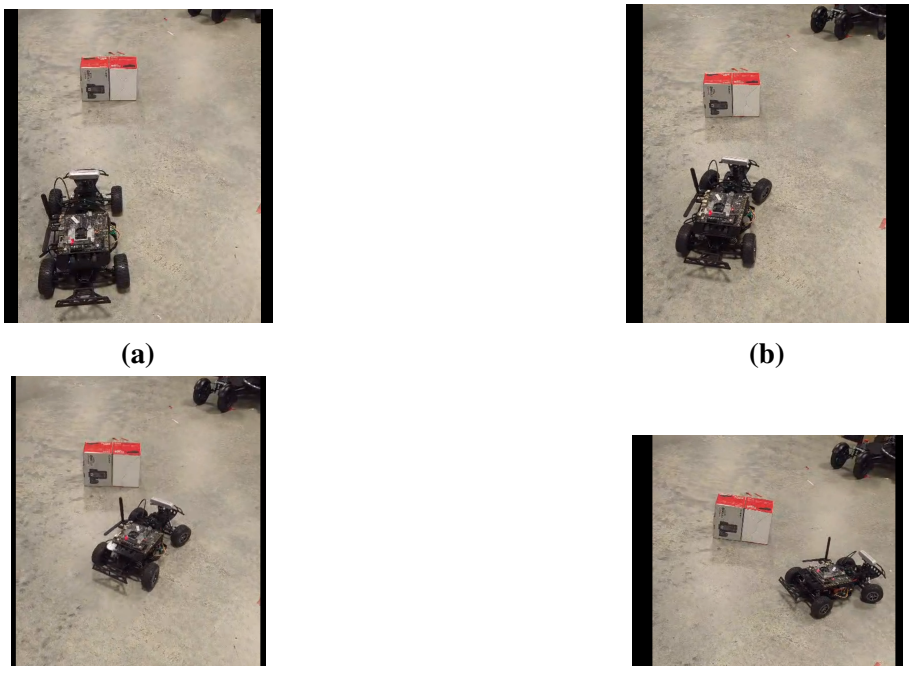
Figure 5.18. Trajectory of car navigating to $[2, 0]$ using State-Only Feature Extractor simulation. (Path: 5.20a $\rightarrow$ 5.20b $\rightarrow$ 5.20c $\rightarrow$ 5.20d).



(a) **(b)**

(c) **(d)**

Figure 5.19. Trajectory of car navigating to $[1, 1]$ using State-Only Feature Extractor



(a) **(b)**

(c) **(d)**

Figure 5.20. Trajectory of car navigating to $[2, 0]$ using State-Depth Concatenation Feature Extractor

Chapter 6

Concluding Remarks

6.1 Conclusion

In this thesis, we demonstrated an approach to design a neural network which is capable of performing motion planning task for an Ackermann modeled car where the action space of the agent is constrained. We also demonstrated that we can control the quality of the policy being trained by constraining the resulting network to be Lipschitz continuous. We were also partially able to achieve of deploying a policy trained simulation onto a real car without any fine-tuning. The deployed policy (modeled as Beta distribution) consisting of state-only feature extractor was able to complete the point-goal navigation task with a high success rate in obstacle devoid environment. From the results shown, we can conclude that our most optimal policy design (comprising of state-depth feature extractor with policy modeled as Beta distribution), is capable of navigating through relatively open or organized environment in a humanely manner.

6.2 Future Work

There are numerous potential extensions to the work done in this thesis. We can extrapolate this problem to navigate an agent in a 6DoF environment rather than a planar environment, so that agent would be capable of unmanned exploration in various environments. On increasing the complexity of the state-space, we can add arms to a differential drive car, for navigating and performing manipulation task in an indoor environment setting. We could also implement

a real-time training framework so that, model is constant updated (at fixed intervals) while navigating.

Bibliography

[1] Nicholas Roy Charles Richter, William Vega-Brown. Bayesian learning for safe high-speed navigation in unknown environments, 2015.

[2] Jean-Baptiste Cordonnier, Andreas Loukas, and Martin Jaggi. Multi-head attention: Collaborate instead of concatenate, 2021.

[3] Takeru Miyato, Toshiki Kataoka, Masanori Koyama, and Yuichi Yoshida. Spectral normalization for generative adversarial networks, 2018.

[4] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation, 2015.

[5] Kevin Scaman and Aladin Virmaux. Lipschitz regularity of deep neural networks: analysis and efficient estimation, 2019.

[6] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2016.

[7] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

[8] Kyle Stachowicz, Dhruv Shah, Arjun Bhorkar, Ilya Kostrikov, and Sergey Levine. Fastrlap: A system for learning high-speed driving via deep rl and autonomous practicing, 2023.

[9] Ruihan Yang, Minghao Zhang, Nicklas Hansen, Huazhe Xu, and Xiaolong Wang. Learning vision-guided quadrupedal locomotion end-to-end with cross-modal transformers, 2022.