DEEP REINFORCEMENT LEARNING ON EDGE: A FEASIBILITY STUDY FOR AUTONOMOUS DRIVING

Thesis to be submitted in fulfillment of the
requirements for the degree
of

M.Tech in Embedded Controls and Software

by

Thummalabavi Sankshay Reddy
23AT61R04

Under the guidance of
Dr.Ayantika Chatterjee

ADVANCED TECHNOLOGY DEVELOPMENT CENTRE
INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR
MAY, 2025

Advanced Technology Development
Centre
Indian Institute of Technology,
Kharagpur
India - 721302

CERTIFICATE

This is to certify that we have examined the thesis entitled **DEEP REINFORCEMENT LEARNING ON EDGE: A FEASIBILITY STUDY FOR AUTONOMOUS DRIVING**, submitted by **Thummalabavi Sankshay Reddy**(Roll Number: *23AT61R04*), a postgraduate student of **Advanced Technology Development Centre** in fulfillment for the award of degree of M.Tech in Embedded Controls and Software. We hereby accord our approval of it as a study carried out and presented in a manner required for its acceptance in fulfillment for the Post Graduate Degree for which it has been submitted. The thesis has fulfilled all the requirements as per the regulations of the Institute and has reached the standard needed for submission.

Supervisor
Dr.Ayantika Chatterjee
Advanced Technology Development
Centre
Indian Institute of Technology, Kharagpur

External Examiner

Place: Kharagpur
Date: May 3, 2025

DECLARATION

I certify that,

1. The work contained in this report is original and has been done by me under the guidance of my supervisors.
2. The work has not been submitted to any other Institute for any degree or diploma.
3. I have followed the guidelines provided by the Institute in preparing the report.
4. I have confirmed the norms and guidelines given in the Ethical Code of Conduct of the Institute.
5. Whenever I have used materials (data, theoretical analysis, figures, and text) from other sources, I have given due credit to them by citing them in the text of the report and giving their details in the references. Further, I have taken permission from the copyright owners of the sources, whenever necessary.

Name: Thummalabavi Sankshay Reddy
Roll no: 23AT61R04
Department:Advanced Technology Development Centre
Place: IIT Kharagpur, West Bengal, India
Date: May 3, 2025

ACKNOWLEDGEMENTS

I express my heartfelt gratitude to my supervisors, Dr. Ayantika Chatterjee and my senior Sumansmita Rout, for their invaluable guidance, support, and encouragement throughout the course of my M.Tech project. Their insights and expertise have been instrumental in shaping this work and have inspired me to push the boundaries of my understanding. I extend my sincere thanks to the faculty and staff of the Advanced Technology Development Centre at IIT Kharagpur for providing a conducive learning environment and the necessary facilities for my research.

Thummalabavi Sankshay Reddy
IIT Kharagpur
Date:May 3, 2025

ABSTRACT

Autonomous Driving System (ADS) integrates AI, sensors, and communication networks to enhance safety, traffic efficiency, and sustainability. Adopting machine learning (ML) techniques provides better handling of the complexities and unpredictability of real-world scenarios. ML methods such as Deep Reinforcement Learning (DRL) leverage deep learning (DL) for robust feature extraction and reinforcement learning's sequential decision-making capabilities in dynamic environments. However, DRL often demands significant computational power, typically supported by cloud servers or GPUs. Communication between edge-based ADS and cloud-based intelligence may lead to challenges such as high latency, reduced reliability, and elevated costs. This thesis focuses on applications of Autonomous Driving Systems (ADS) such as lane deviation detection, setting velocity ranges, collision avoidance, and optimal path finding for navigation. To address this, we explore deploying the DRL model integrated with an autoencoder onto a low-powered edge device. This work especially focuses on the Proximal Policy Optimization (PPO) of the DRL model and improves it with a trainable exploration rate and Generalized Advantage Estimations to handle high-dimensional space. This approach is evaluated using a Processor-in-the-Loop (PIL) framework integrated with the realistic 3D CARLA simulation to achieve a coherent real-time interaction loop between the virtual environment and the edge. The edge prediction for this process achieved an inference time of 36.85 ms with a Root Mean Square Error (RMSE) of 0.0086. To the best of our knowledge, this is the first effort to show the practical feasibility and effectiveness of DRL model deployment on edge devices for real-time autonomous driving applications.

Keywords: ADS, DRL, PPO, Edge computing, CARLA, PIL

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Autonomous Driving Systems (ADS) integrate advanced technologies like sensors, artificial intelligence (AI),rule-based algorithms and communication networks, etc. This will enable vehicles to navigate and operate autonomously or with minimal human intervention. It will enhance safety by reducing human error and improving traffic efficiency [25]. Moreover, it contributes to maintaining sustainability by optimizing fuel consumption and enabling smarter urban mobility solutions[13]. Traditional ADS follows a rule-based, modular approach, where specific applications like ACC, lane maintaining, and parking assist were designed as standalone components [31]. These systems relied on pre-programmed rules and algorithms to handle specific scenarios, like maintaining a safe distance or staying within lane boundaries. For these reasons, it faced limitations due to the complex dynamics and unpredictable nature of real-world environments, such as diverse road conditions, weather variations, unexpected obstacles, etc. Rule-based systems struggled to adapt to these complexities, leading to a stagnation in the progression toward full autonomy[31]. To overcome these challenges, modern ADS are designed with AI and machine learning (ML) techniques [18].

The integration of ML methods into the automobile domain spans a wide spectrum, ranging from simpler control tasks, such as braking, to highly complex operations like fully autonomous driving [4]. Techniques applied in this field vary significantly, encompassing supervised learning approaches like Deep Neural Networks (DNN) as well as more complex methodologies such as Reinforcement Learning (RL) [14]. However, the ML applications in ADS face many challenges, such as DNN, which is more effective for classification and feature extraction tasks, but heavily depends on comprehensive datasets, which are not always readily available in the context of

autonomous driving. Additionally, these methods often fail to adequately handle applications involving sequential decision-making and dynamically evolving environments [17]. In contrast, Reinforcement Learning (RL) based techniques, like Deep Q Neural Networks and On Policy Gradients, have demonstrated superior adaptability in handling complex driving scenarios [19]. Model-based reinforcement learning is explored to mitigate the high data dependence issue by incorporating simulation-based approaches for autonomous driving training. Additionally, safety-driven learning frameworks, such as safe RL and constrained optimization techniques, are developed to ensure autonomous systems make decisions while minimizing the risk of collisions. Currently, many autonomous driving systems are increasingly adopting Deep Reinforcement Learning because of their interactive nature and inherent reward-punishment mechanisms [19]. DRL, in particular, demonstrates effectiveness in managing dynamic and uncertain environments by making policies to learn and adapt interactively [14].

However, deploying DRL models directly on edge devices presents significant challenges, primarily due to the heavy computational and memory demands associated with processing raw, high-dimensional sensory inputs such as images. These constraints limit the feasibility of real-time inference and decision-making on resource-constrained hardware platforms. To address this issue, a Variational Autoencoder (VAE) [15, 26] can be integrated into the DRL pipeline to compress high dimensional observational input into a low dimensional latent representation (commonly referred to as the z-dimension). This dimensionality reduction significantly decreases the input complexity for the policy (Actor) network, enabling lightweight and efficient inference suitable for edge deployment. The VAE architecture not only reduces computational overhead but also preserves the most salient features required for effective policy learning. Furthermore, its probabilistic encoding mechanism enhances robustness to noise and variability in input sensory data, which is the key for maintaining performance in dynamic real-world driving environments. This integrated VAE-DRL approach provides a promising solution for achieving scalable, adaptive, and efficient autonomous navigation on edge computing platforms.

1.1 Need for Simulation Software and Edge Deployment

Simulation software like CARLA is designed for training and testing in realistic urban environments. Training and testing costs, as well as safety considerations, are being effectively managed using high-end simulation software like CARLA and TORCS [8, 43]. Nonetheless, the critical necessity for split-second decision-making and robust security introduces additional complexities, especially concerning latency and security issues when deploying these sophisticated models on cloud servers or high-end GPUs [22].

Considering these points, this is the first work that proposes a framework deploying the trained DRL models specifically tailored for autonomous driving onto resource-constrained edge devices embedded within vehicles. Moreover, our framework suggests integrating Processor-in-the-Loop (PIL) methods with high-fidelity 3D simulation software rather than relying exclusively on physical hardware testing. This strategy leverages advanced 3D simulation environments, which can accurately mimic real-world scenarios, thereby demonstrating the feasibility and effectiveness of deploying DRL models on edge platforms for autonomous driving.

1.2 The key contributions

- This is the first study that explores edge-based DRL implementation for autonomous tasks such as collision avoidance, lane-keeping, speed regulation, and optimal path planning using an edge platform.
- The DRL model, integrated with a VAE, is trained on the CARLA 3D simulation using a GPU. A dataset is then generated from the trained model, followed by model quantization to balance computational latency and accuracy for efficient edge deployment. Finally, the quantized model is evaluated directly on edge hardware to assess computational timing and loss performance without approximations.
- The Processor-in-the-Loop (PIL) technique consolidates the GPU and edge modules into a cohesive real-time interaction loop. This facilitates the high-

performance GPU-based simulation to dynamically interact with the resource-limited edge device, which is tasked with computing real-time policy decisions.

- Evaluating and comparing the performance of the optimized DRL model on GPU-based systems and resource-constrained edge devices by analyzing parameters such as computational latency, decision-making speed, cumulative reward, and overall task efficiency (e.g., collision avoidance, lane-keeping accuracy, and velocity control).

1.3 Organization of the Thesis

The thesis is divided into five chapters, each contributing to the training and deployment of a DRL-based autonomous driving system with edge inference capabilities.

- **Chapter 1: Introduction** This chapter explains the domain of autonomous driving systems (ADS), outlining the motivation, research challenges, and the primary objectives of the work. It also highlights the contributions and scope of the thesis.

- **Chapter 2: Background** This chapter explains foundational knowledge required for understanding the work. It describes traditional rule-based ADS, the role of machine learning in ADS, and introduces Deep RL concepts such as policy gradient, actor-critic, and DRL architectures.

- **Chapter 3: Literature Survey** A comprehensive review of recent research in DRL for autonomous navigation and edge computing. It identifies gaps in current solutions related to scalability, safety, and inference efficiency, providing context for the proposed framework.

- **Chapter 4: Proposed Framework** This chapter details the system architecture integrating CARLA simulation, Variational Autoencoder (VAE), and PPO-based DRL agent. It explains each component from data acquisition to edge deployment, with a focus on training, feature compression, and inference flow.

- **Chapter 5: Conclusion** Observations and insights are presented in this chapter regarding the experimental design, performance evaluation. It discusses how the proposed framework handles exploration, model performance, and adaptive nature to dynamic environments and summarizes the outcomes and contributions of this thesis.

- **Chapter 6: Future work** This thesis concludes with possible future extensions, focusing on improving generalization, safety mechanisms, and real-world deployment of edge-based autonomous driving systems.

Chapter 2

Background

2.1 Overview of Autonomous Driving Systems (ADS)

Autonomous Driving Systems (ADS) are designed to allow vehicles to sense their surroundings, autonomously plan actions, and move safely without relying on human input. These systems integrate a combination of sensors like LiDAR and cameras, computational units, and control software to achieve autonomy [12]. ADS operate in complex, dynamic environments, requiring robust perception, prediction, and planning modules for real-time decision-making and control.

2.1.1 Traditional Rule-Based Systems

Early developments in ADS relied on rule-based systems, where expert-defined heuristics and deterministic models governed behavior. These systems typically followed a modular pipeline: perception (e.g., object detection), localization, mapping, planning, and control [25]. Each module operated independently, often using handcrafted rules and classical control techniques like PID controllers.

Prominent examples include:

- **Stanley (2005)**: Developed by Stanford for the DARPA Grand Challenge, Stanley utilized a rule-based architecture combined with sensor fusion and map-based navigation [39].
- **CMU Navlab**: A series of autonomous vehicles built by Carnegie Mellon University that used a combination of computer vision and reactive planning heuristics [38].

- **Bertha (2013)**: A Mercedes-Benz research project demonstrating highway autonomy through predefined rules, high-definition maps, and deterministic behavior models [46].

While effective in structured environments, rule-based systems struggle with generalization and scalability. Their performance degrades in unstructured or novel scenarios due to the rigidity of hand-engineered logic.

2.1.2 Machine Learning in Autonomous Driving

To overcome the limitations of rule-based systems, the ADS community has increasingly turned to Machine Learning (ML), which enables data-driven decision-making. ML-based systems leverage supervised learning for perception tasks (e.g., object detection [27]), and reinforcement learning or imitation learning for control and planning [3, 6]. ML methods offer improved adaptability to new environments, robustness under uncertainty, and the ability to learn complex, nonlinear decision policies directly from data. This paradigm shift supports end-to-end approaches and hybrid systems where learning augments traditional modules. As the field progresses, Deep Reinforcement Learning (DRL) is also gaining traction for sequential decision-making under partial observability and dynamic conditions [19, 21].

2.1.2.1 Deep Neural Networks (DNNs)

DNNs are widely employed in autonomous driving for tasks such as lane detection, trajectory prediction, and behavioral cloning. DNNs can model non linear relationships in high dimensional data, making DNN suitable for learning end-to-end driving policies or function approximations within a modular ADS [3]. For example, NVIDIA's PilotNet uses a DNN to map camera visual images directly to control commands.

2.1.2.2 Convolutional Neural Networks (CNNs)

CNNs are a specialized type of DNN, particularly effective for image-based perception tasks. CNNs are used in object detection, semantic segmentation, and scene understanding—critical components of ADS perception modules [5, 11]. State of the art models like YOLO and Mask R-CNN have been adapted for real time collision avoidance and road scene parsing in self-driving applications.

- **YOLO model**: Enables real-time object detection from onboard cameras [27].
- **DeepDriving**: Predicts affordance indicators from images to support decision-making [5].
- **Mask R-CNN**: Facilitates instance-level segmentation for detecting and classifying objects in complex scenes [11].

They are widely used in autonomous driving for object detection tasks such as identifying vehicles, pedestrians, and traffic signs from camera inputs, and when integrated with Reinforcement Learning, they enhance decision-making by extracting high level spatial features from visual data, enabling the agent to perceive and respond effectively to dynamic driving environments.

2.2 Reinforcement Learning

2.2.1 Overview

Reinforcement Learning (RL) is a machine learning approach in which an agent interacts with its environment, experimenting with actions and adjusting its behavior based on the rewards or penalties it receives. The agent learns by trying different actions and getting feedback as rewards or penalties, with the goal of maximizing its total rewards over time. In contrast to supervised learning, Reinforcement Learning learns through the outcomes of its actions rather than depending on labeled datasets. This makes it especially useful for solving complex sequential decision making problems like autonomous systems.

2.2.2 Reinforcement Learning Concepts

Exploration vs. Exploitation[9] An agent must balance *exploration*—trying new or suboptimal actions to gather information about the environment—and *exploitation*—leveraging the best-known actions to maximize reward. Early in training, exploration is emphasized, gradually shifting toward exploitation as the agent learns an effective policy [40]. Techniques like ϵ-greedy or entropy regularization are commonly used to manage this trade-off.

Fully Observable vs. Partially Observable Environments In *fully observable* environments, such as chess, the agent has access to the complete state at all

times. In contrast, *partially observable* environments, like autonomous driving or robotic navigation, reveal only a subset of the true state, requiring the agent to maintain memory or beliefs about the hidden state, often using methods like Recurrent Neural Networks (RNNs) or Partially Observable Markov Decision Processes (POMDPs) [40].

Discrete vs. Continuous State and Action Spaces State and action spaces can be either *discrete* or *continuous*. Discrete spaces, as found in board games like chess, consist of a finite set of states and actions. Continuous spaces, common in robotics and real-world control problems, involve real-valued variables, requiring specialized algorithms like Deep Deterministic Policy Gradient (DDPG) or Proximal Policy Optimization (PPO) [40].

Deterministic vs. Stochastic Policies A *deterministic policy* maps each state to a specific action and is suitable for deterministic environments. A *stochastic policy* outputs a probability distribution over actions, enabling the agent to handle uncertainty and ambiguity more effectively. Stochastic policies are especially valuable in environments with hidden variables or when learning from demonstrations [40].

On-Policy vs. Off-Policy Learning *On-policy* methods, such as Proximal Policy Optimization (PPO), learn using the same policy for both data collection and policy improvement. *Off-policy* methods, like Q-learning and Deep Deterministic Policy Gradient (DDPG), allow learning from past experiences generated by different policies. Off-policy learning improves sample efficiency and enables experience replay [40].

Monte Carlo vs. Temporal Difference Methods *Monte Carlo* methods estimate returns by averaging rewards over complete episodes, offering unbiased estimates but requiring episode termination. *Temporal Difference (TD)* methods, such as TD(0), TD(λ), or SARSA, update estimates using bootstrapping—relying partly on existing value estimates. TD methods can learn online and are better suited for continuing tasks [40].

2.2.3 Types of Reinforcement Learning

Reinforcement Learning (RL) algorithms can be broadly classified based on how they learn and represent the environment or policy. Two major distinctions are: *model-free vs. model-based* methods and *value-based, policy-based, and actor-critic* approaches.

- **Model-Free vs. Model-Based:**
  - *Model-Free* methods learn directly from experience without building an explicit model of the environment. Examples include Q-Learning and Policy Gradient methods.
  - *Model-Based* methods attempt to learn a model of the environment's dynamics (transition probabilities and rewards) and use this model to plan or simulate future states, enabling more sample-efficient learning.
- **Value-Based, Policy-Based, and Actor-Critic Methods:**
  - *Value-Based* methods focus on learning value functions, such as $V(s)$ or $Q(s, a)$, and derive the policy indirectly by selecting actions that maximize value (Q-Learning, DQN).
  - *Policy-Based* methods learn a parameterized policy directly, without needing value functions. These are especially useful in continuous action spaces (REINFORCE).
  - *Actor-Critic* methods combine both approaches: the *actor* updates the policy directly, while the *critic* evaluates the policy using a value function. This structure helps stabilize and accelerate learning (A3C, PPO).

2.3 Agent-Environment Interaction

In Reinforcement Learning, the learning process is framed as an interaction between an *agent* and an *environment* over discrete time steps. At each time step t, the agent observes the current state s_t of the environment and selects an action a_t based on its policy $\pi(a|s)$. The environment then transitions to a new state s_{t+1} and returns a reward r_t to the agent. This sequence of interactions forms a trajectory:

$$s_0, a_0, r_0, s_1, a_1, r_1, \ldots$$

The goal of the agent is to learn a policy that maximizes the expected cumulative reward, also known as the return. This interaction loop is typically modeled using a Markov Decision Process (MDP)[36], which provides a formal framework for describing the probabilistic transitions and rewards in the environment. The quality of an agent's actions depends on how effectively it can use the observed states and rewards to improve its decision-making over time.

2.3.1 Markov Decision Process (MDP)

A Markov Decision Process (MDP)[36] is a mathematical framework used to formalize the agent-environment interaction in Reinforcement Learning. It provides the basis for most RL algorithms and encapsulates the decision-making process under uncertainty. An state-action pair is defined by the tuple $(\mathcal{S}, \mathcal{A}, P, R, \gamma)$. The MDP framework assumes the *Markov property*, meaning the future state depends only on the current state and action, not on the full history of prior states. MDPs serve as the foundational model for most Reinforcement Learning algorithms, connecting key RL concepts:

- **Value Functions:** The state-value function $V(s)$ and action-value function $Q(s, a)$ are defined with respect to MDP transitions and rewards.
- **Policy Learning:** Policies are learned to optimize expected return based on the MDP structure, using either model-free or model-based approaches.
- **Bellman Equations:** The recursive nature of value estimation in RL stems from the Bellman equation, which is derived from the MDP framework.
- **Exploration and Planning:** In model-based RL, planning relies on learned or known MDP dynamics; in model-free RL, exploration strategies assume MDP properties to update value or policy functions.

By providing a formal representation of decision-making under uncertainty, MDPs enable RL agents to reason about long-term consequences, plan actions, and optimize performance across various environments. Reinforcement learning concerns a subset of MDPs where the agent is unaware of the transition probabilities $p(s' \mid s, a)$ and reward function $r(s, a)$, requiring exploration to learn the relationship between state-action pairs and rewards.

2.3.2 Reward Function

Reward function is essential for reinforcement learning (RL). At each time step tt, the reward is commonly represented as $r_t = R(s_{t+1}, s_t, a_t)$[36]. Alternatively, a reward can also be designed for a set of trajectories τ by summing rewards over a finite time horizon T:

$$R(\tau) = \sum_{t=0}^{T} r_t \tag{2.1}$$

A discount factor γ^t will give importance to immediate rewards over distant ones, aiding the agent in learning a good policy.

2.4 Classical Reinforcement Learning

Traditional reinforcement learning (RL) methods are primarily value-based approaches [37], where the objective is to estimate the desirability of being in a particular state or taking a specific action in that state. These techniques are rooted in the Markov Decision Process (MDP) framework and focus on calculating value functions that inform the agent's decisions over time.

2.4.1 Q-Functions

The *action-value function*, or *Q-function*, denoted by $Q^\pi(s, a)$, estimates the expected return from taking action a in state s under a given policy π. The Q-function is central to many RL algorithms, especially those that aim to learn optimal behavior, such as Q-learning. The optimal action-value function $Q^*(s, a)$ satisfies the Bellman optimality equation:

$$Q^*(s, a) = \mathbb{E}\left[R(s, a) + \gamma \max_{a'} Q^*(s', a')\right] \tag{2.2}$$

Here, s' is the next state where current action a is applied for present state s. This recursive relationship forms the basis of Q-learning updates.

2.4.2 Value Iteration and Policy Iteration

Classical RL methods such as *Value Iteration* and *Policy Iteration* directly use these value estimates to improve decision-making. In *Q-Learning*, an off-policy algorithm, the agent iteratively updates the Q-values using the following rule [42]:

$$Q(s, a) \leftarrow Q(s, a) + \alpha\left[R(s, a) + \gamma \max_{a'} Q(s', a') - Q(s, a)\right] \tag{2.3}$$

Here, α is the learning rate, and γ is the discount factor. This process gradually refines the estimates of future rewards and converges toward the optimal Q-function under certain conditions.

These foundational methods laid the groundwork for more advanced approaches such as *Deep Q-Networks (DQN)* [21], which combine Q-learning with deep neural networks to handle high-dimensional state spaces.

2.5 Deep Reinforcement Learning: Neural Network-Based Decision Making

2.5.1 Policy Gradient Methods

Unlike value-based methods, policy gradient approaches directly optimize the policy $\pi(a|s;\theta)$ by performing gradient ascent on the expected return. This method is suitable for continuous action spaces and stochastic policies but suffers from high variance, which motivates the use of baselines and variance-reduction techniques. Policy gradient methods are reinforcement learning techniques that use gradient descent to optimize policies concerning expected rewards [10]. In terms of how the policy changes, the methods use an estimate of the gradient of the expected rewards to determine if the policy changes would be advantageous or not.

2.5.2 Actor-Critic Methods

The Actor-Critic*(A2C)* [20, 35] architecture is a widely used framework in RL [26] that combines policy-based and value-based methods to optimize decision-making in complex environments. The actor is responsible for selecting actions based on the current policy. It maps states to actions using a parameterized policy $\pi_\theta(a|s)$, which can be either deterministic or stochastic. The actor continuously updates its policy parameters (θ) to maximize expected rewards. The critic evaluates the actions taken by the actor by estimating the value function. It learns to approximate either the *state-value function* $V(s)$ or the *advantage function* $A(s,a)$, which helps in reducing the variance of policy updates. The critic provides feedback to the actor to guide policy improvement.

$$A(s,a) = Q(s,a) - V(s) \tag{2.4}$$

The *Actor-Critic* method addresses key challenges in RL by balancing exploration and exploitation while leveraging value estimates for stable learning. This architecture forms the foundation for advanced algorithms such as *Advantage Actor-Critic (A2C)*, *Proximal Policy Optimization (PPO)*[28], and *Deep Deterministic Policy Gradient (DDPG)*[35], which are widely used in continuous control tasks, including autonomous driving in CARLA.

2.5.3 Key Algorithms in DRL

DRL is a ML model that combines Reinforcement Learning (RL) with Deep Learning techniques to enable agents to make decisions from high-dimensional inputs such as raw pixel data, time series, or sensor readings. While traditional RL methods rely on box representations or handcrafted features, DRL leverages deep neural networks to learn representations, policies, and value functions directly from data [21].

In DRL, neural networks are typically employed in one or more of the following roles:

- **Policy Network**: Maps observations to actions, used in policy-based methods.
- **Value Network**: Estimates value functions like $V(s)$ or $Q(s, a)$, as in value-based methods.

This integration enables agents to operate in environments with complex, high-dimensional state and action spaces, such as robotic manipulation, autonomous driving, and video games, where classical methods are computationally infeasible or require domain-specific engineering.

2.5.3.1 Deep Q-Network (DQN)

The Deep Q-Network (DQN) algorithm [21] builds on traditional Q-learning by utilizing a convolutional neural network (CNN) to approximate the Q-value function Q(s,a)Q(s,a). It introduces two significant innovations: experience replay and the use of a target network, both of which contribute to more stable learning by mitigating the issues of correlated data and reducing the risk of oscillations in training. DQN gained widespread recognition for its success in Atari 2600 games, where it achieved human-level performance by learning directly from raw pixel inputs. This approach is particularly well-suited for environments with discrete action spaces, enabling the

learning of effective policies from high-dimensional inputs without the need for manual feature extraction.

2.5.3.2 Deep Deterministic Policy Gradient (DDPG)

The Deep Deterministic Policy Gradient (DDPG) algorithm [19] is tailored for environments with continuous action spaces. It integrates deterministic policy gradients with function approximation, employing two neural networks: one for the policy (actor) and the other for the Q-function (critic). To enhance stability, DDPG utilizes target networks and experience replay. This method has been successfully applied to tasks such as robotic arm control, autonomous vehicle steering, and drone navigation. DDPG is particularly effective for problems with high-dimensional, continuous action spaces, offering efficient policy updates with reduced variance compared to stochastic approaches.

2.5.3.3 Proximal Policy Optimization (PPO)

PPO [28] is a On-policy method that improves stability by using a clipped surrogate objective to constrain the policy update. PPO avoids overly large updates that can destabilize training, while remaining simpler to implement than Trust Region Policy Optimization (TRPO). PPO has been applied in domains such as humanoid locomotion, robotic walking, game AI (OpenAI Five), and simulation-to-real policy transfer. It combines strong performance with robustness and ease of use. It supports continuous and discrete actions and performs well in both fully and partially observable environments.

2.5.3.4 Asynchronous Advantage Actor-Critic (A3C)

The Asynchronous Advantage Actor-Critic (A3C) algorithm [20] leverages multiple agents running in parallel across diverse environments to enhance training stability and promote better exploration. Each agent operates with its own local model and updates a shared global model in an asynchronous manner. This approach integrates the actor-critic architecture with an advantage function to help minimize the variance in the learning process. A3C has proven effective in tasks such as navigation, strategy games, and robotic path planning, especially in scenarios where parallel data processing and rapid convergence are crucial. Its high data efficiency, quick learning

through parallelization, and improved exploration make it well-suited for large-scale and real-time systems.

2.6 Proximal Policy Optimization

Proximal Policy Optimization algorithm [28, 26] is considered in this study due to its effectiveness in constraining policy updates within a safe range, thereby enhancing training stability and convergence. It builds upon the foundational ideas introduced by Trust Region Policy Optimization [29], which limits policy changes to a trust region to prevent large, destabilizing updates during training. PPO achieves a similar effect but replaces TRPO's complex optimization constraints with a clipped objective function, making it significantly more computationally efficient and easier to implement in practical scenarios. Proximal Policy Optimization (PPO) seeks to combine the advantages of Trust Region Policy Optimization (TRPO) [29] by utilizing first-order optimization methods. A key component of PPO is the use of a clipping mechanism, which constrains the effect of gradient updates to prevent drastic changes in the action probabilities. This clipping ensures that the updated policy remains within a reasonable range compared to the previous policy, thereby stabilizing the learning process and avoiding large, unpredictable shifts in the agent's behavior.

Algorithm 1 Traditional PPO

for iterations = 1, 2, ... **do**
  for actor = 1, 2, ..., N **do**
    Run policy $\pi_{\theta_{\text{old}}}$ in environment for T time-steps
    Collect a set of trajectories with policy $\pi_i = \pi(\theta_i)$ in training environment
    Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$ according to value
  end for
  Optimize surrogate L respect to θ,with K epochs and mini-batch size $M \leq NT$
  $\theta_{\text{old}} \leftarrow \theta$
end for

2.6.1 Clipped Surrogate Objective

The core PPO objective is defined as:

$$L^{\text{CLIP}}(\theta) = \mathbb{E}_t \left[\min \left(r_t(\theta)\hat{A}_t,\ \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t \right) \right] \tag{2.5}$$

with:

- $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$: the probability ratio between the new and old policies.
- ϵ: a hyperparameter (typically 0.1 or 0.2) that constrains the update step.

This formulation ensures conservative policy updates by penalizing excessive deviations from the previous policy.

2.6.2 Advantages of PPO

Proximal Policy Optimization (PPO) offers several advantages that make it particularly suitable for sequential decision-making tasks in Autonomous Driving Systems (ADS):

- **Stability in Learning:** PPO uses a clipped surrogate objective that restricts large policy updates, providing stable and reliable learning, essential for safety-critical applications like autonomous driving.
- **Sample Efficiency:** By enabling multiple epochs of mini batch updates from the same trajectory data, PPO improves sample efficiency, which is beneficial in simulation-heavy ADS training environments such as CARLA.
- **Support for Continuous Action Spaces:** PPO works well with continuous actions (e.g., steering angle, throttle), which are inherent in driving control tasks, by modeling stochastic policies with continuous distributions.
- **Robustness in Noisy or Partially Observable Environments:** PPO maintains performance under noisy sensory inputs and partial observability, common conditions in real-world driving scenarios like poor lighting or occlusions.
- **On-Policy Learning:** PPO uses data collected by the current policy, allowing tighter feedback loops and policy alignment, which is important when adapting to rapidly changing traffic dynamics.

2.7 Variational Autoencoder (VAE)

The Variational Autoencoder (VAE)[2][32] acts as a feature extraction unit, enhancing the observational state space that the agents are trained on, making it more structured and eventually simpler to predict. A Variational Autoencoder (VAE) [15, 26] has three main components: the encoder, the latent space z_{dim}, and the decoder. The encoder includes a CNN for feature extraction of the input data, such as lane markers, sidewalks, roads, pedestrians, etc. Afterward, the features are compressed into a lower dimensional latent state by mapping them to a probability distribution characterized by a mean (μ) and standard deviation (σ). This ensures that similar inputs produce similar latent vectors, enabling smooth interpolation. The latent space (z-space) serves as the probabilistic bottleneck where sampling occurs, introducing a degree of randomness that enhances the model's generalization ability. The decoder rebuilds the observation data from the sampled latent variables, effectively learning meaningful representations. In our work, VAE is utilized solely for latent space representation, which is then provided to a reinforcement learning agent, significantly reducing the computational burden while preserving task-relevant information. The latent space is sampled by interpreting one of the parallel heads as the mean of a Gaussian distribution, and the other head as the standard deviation. These outputs are used to sample the latent vector z from the Gaussian distribution, which is subsequently passed to the decoder. The decoder starts with a fully connected layer to reshape z to the dimensions of the final convolution output from the encoder. This allows the image to be reconstructed to its original size via transposed convolutions.

2.8 CARLA Environment

Car Learning to Act [8] is an open source 3D high end simulator designed for autonomous application research , developed by the Computer Vision Center (CVC). It provides a high realistic environment for training and validating reinforcement learning (RL) models in realistic urban scenarios [7]. It offers a flexible API for controlling vehicles, pedestrians, and infrastructure elements, making it suitable for reinforcement learning applications like behavior prediction, control decision making, and end to end autonomous driving. One of the key reasons CARLA is essential for autonomous driving (AD) training is its ability to provide a controlled and cost-effective

alternative to real-world testing. CARLA provides different layouts, like rural (town 1) and urban (town 2), with their own road patterns and obstacle complexities. In this work, our goal is to evaluate whether reinforcement learning can be effectively applied to a driving scenario that mirrors real-world conditions. The agent must learn to interpret when to focus on certain objects and when to ignore them based on input state.

2.9 Reward Shaping

Shaping of reward function is a critical technique in RL used to guide agent behavior by modifying the reward signal to accelerate learning and improve convergence. In complex environments, especially those with sparse or delayed rewards, agents may struggle to explore effectively or identify optimal policies. Reward shaping introduces additional signals or intermediate rewards that can lead the agent toward desirable behaviors more efficiently [24].

Properly shaped rewards can:

- **Accelerate Learning**: By providing more frequent feedback, agents can learn useful behaviors faster.
- **Encourage Exploration**: Shaped rewards can steer the agent toward unexplored states or useful strategies.
- **Reduce Sparse Reward Problems**: Intermediate rewards help the agent connect sequences of actions with long-term goals.
- **Incorporate Domain Knowledge**: Engineers can embed prior knowledge about desirable behavior, improving sample efficiency.

However, poorly designed reward shaping may result in suboptimal policies or unintended behaviors due to reward hacking. Careful design is essential to ensure alignment between the shaped reward and the true task objective [1].

Chapter 3

Literature Survey

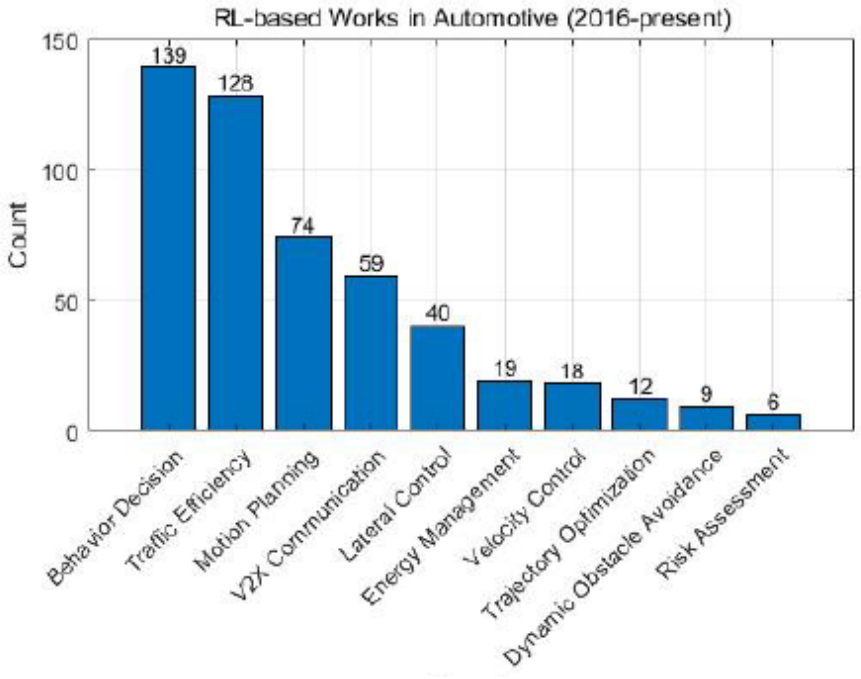


Figure 3.1: Recent works of Deep Reinforcement Learning in Autonomous Driving Domain[44]

There are numerous methods and techniques employed in the domain of autonomous driving, ranging from traditional rule-based systems to complex ML algorithms. Despite the progress made, each of these methods has limitations that must be addressed to develop truly autonomous systems capable of safely and efficiently navigating real-world environments.

3.1 Deep RL in Autonomous Domain

Table 3.1: Comparison of Algorithms for Various Scenarios

Algorithm	**Platform**	**Objective/Pros**	**Ref.**
TD3	CARLA	Instantaneous Solution from Pretrained Network	[31]
Actor-Critic	Custom Simulator	Motion Planning	[32]
SAC	MATLAB	Efficiency on Highways	[33]
Fuzzy DRL	Deep Racer platform	Fuzzy Deep Reinforcement Learning	[34]
TRPO and PPO	F1 Tenth	Policy Learning for Speed Racing	[35]
SAC	CARLA	Control Methods for Handling Limits	[36]
DDPG	Custom Simulator	Motion Planning	[37]
DDPG	Zala Zone	Trajectory Learning	[38]
Double DQN	ROS and Gazebo	Superior Generalization	[40]
DDPG	Unity	Navigation in Off-road Environments	[41]

Models like Convolutional Neural Networks (CNNs) are effectively utilized by NVIDIA Corporation to map raw pixels from camera inputs directly to steering commands, demonstrating a novel end to end approach in autonomous driving [4]. These requirements pose significant barriers, particularly for edge computing environments where resource constraints are a critical challenge. Hence, CNNs often need to be supplemented with more adaptive frameworks like Deep Reinforcement Learning (DRL) [28, 35, 33] to handle control in complex and dynamic driving environments effectively. RL methods, such as DQN [33], have showed remarkable efficiency in tasks like Atari games and robotic control. However, they face significant challenges, particularly when dealing with continuous control spaces or high-dimension input data. In such cases, the "curse of dimensionality" [16, 36] makes it increasingly difficult to estimate Q-values accurately as the state-action space grows exponentially. Furthermore, DQN [33] can become unstable or diverge due to the complexities introduced by Bellman's equation [36] when applied to such environments. These challenges highlight the need for alternative methods like policy gradient approaches, which are better equipped to handle the intricacies of continuous and high-dimension settings.

Recent works [19, 28, 14] in the autonomous driving domain using DRL, such as DDPG and PPO, have achieved significant results in areas like lane deviation minimization, collision avoidance, and optimal path planning. PPO is better suited for

edge computing environments compared to DDPG[35] due to its relative simplicity and low computational overhead. While DDPG utilizes a more complex architecture involving four different neural networks [35] and a replay buffer, PPO employs a more streamlined approach that eliminates the need for a replay buffer and focuses on directly optimizing the policy. This simplicity reduces memory usage and computational overhead, making PPO particularly advantageous for resource-constrained edge devices [28, 33]. However, these advancements rely heavily on expensive setups, such as GPUs, making them less feasible for deployment in resource-constrained environments [34, 22]. Notably, minimal resource setups like embedded or edge computing devices are often overlooked in these studies. These setups present unique challenges, including low computational power, limited memory, and stringent power consumption requirements [45]. Deploying DRL models on the edge requires careful selection of a model with minimal complexity to meet the constraints of limited computational resources. For this purpose, Proximal Policy Optimization (PPO) with an actor-critic architecture is a suitable choice due to its balance of efficiency and performance [28].

To further reduce computational demands, the DRL model is supplemented with a Variational Autoencoder (VAE). This integration enables the use of a low-dimensional latent space, significantly minimizing the computational load while maintaining robust decision-making capabilities [15]. Moreover, research has explored quantization techniques, like post-training quantization, to compress deep learning models for edge AI applications. Pruning methodologies have also been employed to optimize neural network architectures, reducing the number of computations required while preserving essential task performance [23]. Additionally, attention-based mechanisms and lightweight convolutional networks have been introduced to enhance the efficiency of vision-based DRL models for real-time deployment [41]. These techniques pave the way for scalable, energy-efficient AI solutions for real-world autonomous driving applications.

Despite the advancements in traditional rule-based systems and supervised learning models like CNNs, these methods often fall short in dynamically evolving and uncertain environments due to their limited adaptability and reliance on extensive labeled datasets. Traditional approaches struggle with real-time generalization to novel scenarios, while supervised learning techniques face challenges in decision-making under uncertainty. DRL offers a promising alternative by enabling agents to learn

optimal and valid actions by interacting with the environment, without explicit supervision. Among DRL methods, PPO stands out due to its low computational overhead, robustness, and stability during training. Its actor-critic architecture, clipped objective function, and avoidance of replay buffers make it particularly suitable for edge deployment in autonomous driving, where resources are limited but responsive, and real-time decision-making is critical. PPO's balance between performance and simplicity makes it an ideal candidate for scalable and adaptive autonomous systems.

Chapter 4

Proposed Framework

This work explores end-to-end prediction, using a PPO-based RL agent (actor and critic networks) integrated with a VAE, which compresses high dimensional semantic images into a low dimensional latent space (z-dim). This enables efficient policy learning while reducing computational overhead. Afterwards, the VAE and the actor model are quantized and deployed onto a low-cost edge device to enable real-time inference under resource constraints. To overcome the challenges of physical hardware testing and to enable closed-loop evaluation, we implemented a Processor-in-the-Loop (PIL) framework that bridges the CARLA simulation environment with edge-based computation.

4.0.1 System Model

The system consists of a Processor-in-the-Loop (PIL) based Deep Reinforcement Learning (DRL) pipeline integrated with the CARLA simulation environment and designed for real-time prediction using edge devices. It comprises three major modules: the Simulation Environment, DRL Training Module (on GPU), and the Edge-based Prediction Node.

4.0.1.1 Simulation phase

This module simulates an autonomous vehicle operating in a virtual urban environment using CARLA. It captures two primary sensory inputs, Image Data via a semantically segmented camera, and Navigation Data (e.g., waypoints, speed, etc.) from virtual sensors. These inputs represent the state of the environment and are forwarded to both training and prediction nodes.

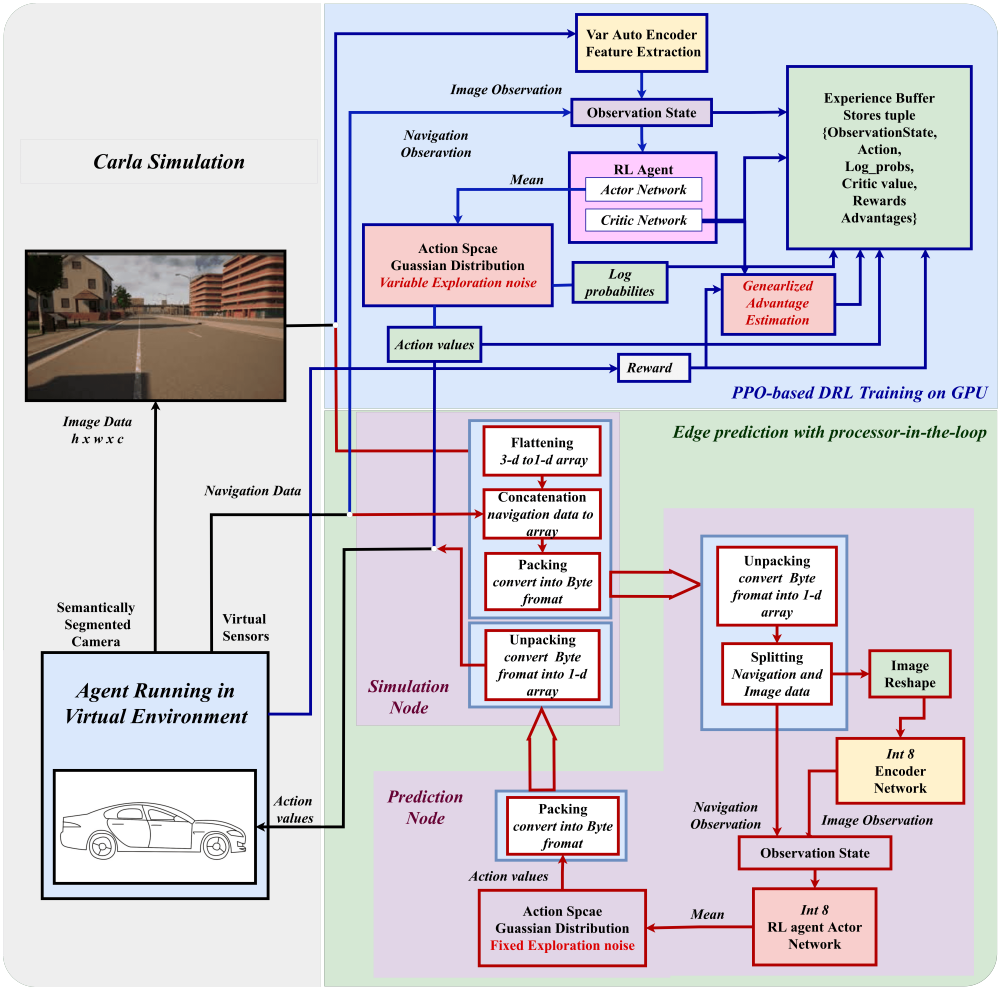


Figure 4.1: End-to-End DRL-Based Prediction with PPO in a Processor-in-the-loop Framework

4.0.1.2 Training phase

This module handles online training of the DRL model using the modified PPO algorithm. Image Observation is passed through a Variational Autoencoder (VAE) for dimensionality reduction, generating a compact latent vector. The VAE output and navigation observation are concatenated to form a complete observation state. This state feeds into the RL Agent, consisting of two fully connected neural networks which are, Actor(to output actions) and a Critic(to estimate state value). The action space follows a Gaussian distribution with variable exploration noise during training. The network outputs include log probabilities, action values, and the estimated mean, which are used to calculate rewards and advantages. Experience tu-

ples—containing observation states, actions, log probabilities, rewards, critic values, and advantages—are stored in an experience buffer.

4.0.1.3 Prediction phase

Prediction using PIL needs both a simulation and an edge node to work in a closed loop. In the simulation node, Image and navigation data are first flattened and converted into byte format using serialization techniques. The data is then unpacked and sent to the prediction node for inference. The packed byte stream is unpacked, split into navigation and image data, and the image is reshaped. The processed data is passed through a quantized Encoder Network to match edge device constraints. The resulting latent image features and navigation data form the observation state, which is input to a lightweight RL Actor Network. This actor network generates action values using a Gaussian distribution with fixed exploration noise, suitable for deterministic behavior during inference. The actions are sent back to the CARLA simulation for execution, closing the PIL feedback loop.

4.0.2 Proposed PPO-based DRL Training on GPU

In the Fig. 4.2, the training framework incorporates an agent built by pipelining the VAE and the DRL model that leverages Proximal Policy Optimization (PPO) for decision-making in the CARLA simulated environment.

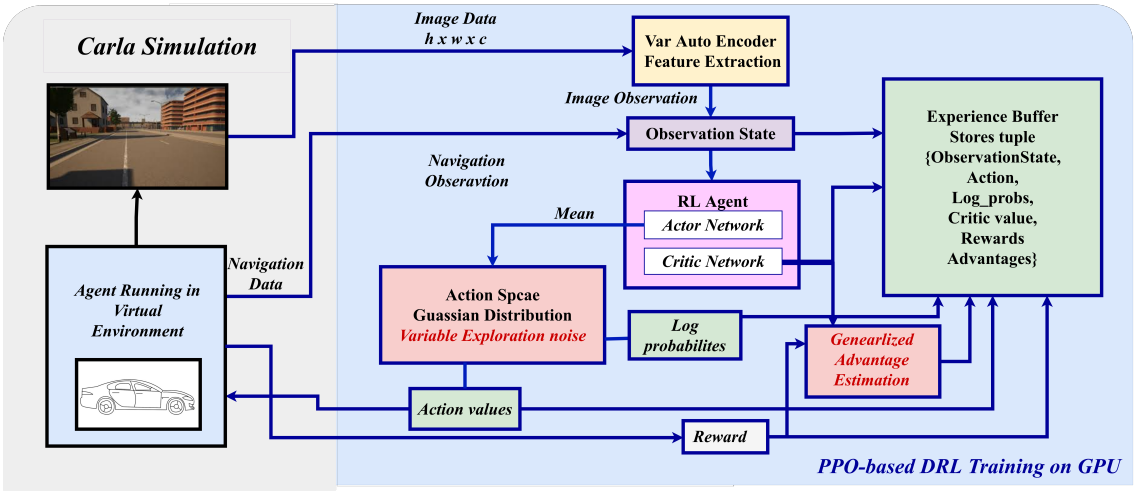


Figure 4.2: PPO based DRL Training

4.0.2.1 Training Model (VAE and DRL)

In the simulation, VAE is employed for feature extraction of lane markers, sidewalks, roads, pedestrians, etc., and it provides the dimensionality reduction of the feature map through the latent space, as mentioned in section 2.7. The input (160×80 RGB image), specifically a semantically segmented (SS) frame) is fed to VAE, as illustrated in Fig. 4.3. The dataset used for training the VAE was sourced from a publicly available repository[26], where data was collected by manually driving a vehicle in the CARLA simulator to record observations and actions across diverse urban scenarios and fed to the VAE to process the input image through a series of Convolution layers to generate a 95-dimensional latent vector, which captures the spatial and semantic structure of the scene. With this latent space, additional navigation features like vehicle speed, throttle, steering angle, distance from the lane center, and orientation angle are concatenated, resulting in a 100-dimensional observation state. This observation vector combines both visual and dynamic driving state information, which is passed to the agent as an input. The RL agent (DRL model) uses the actor and critic architecture, as mentioned in section 2.5.2, to make sequential decisions. In this model, the actor is responsible for calculating mean values based on the present observation, as depicted in Fig. 4.3. Using these mean values an action (throttle and steer) is sampled from gaussian distribution, while the critic decides the chosen actions by estimating their value, guiding the learning process. We train our DRL model is trained using the PPO algorithm to keep policy updates within a stable range, which is explained below.

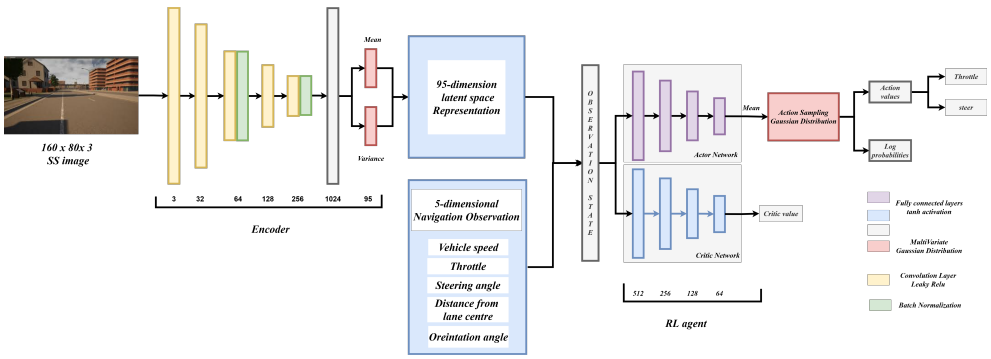


Figure 4.3: Training Model (VAE integrated with RL agent)

4.0.2.2 Proposed PPO for autonomous Driving

In this thesis, we modify the traditional PPO as defined in the section 2.6 framework, specifically tailored for autonomous driving by incorporating a variable training rate, Generalized Advantage Estimation (GAE)[30], and an entropy bonus to balance exploration and exploitation. Modifications as follows:

Initialize policy π_θ , value function parameters π_ϕ,exploration noise σ_{noise}, and buffer B
for each iteration $k = 1, \ldots, K$ **do**
    for each episode $n = 1, \ldots, N$ **do**
        if n mod $p = 0$ **then**
            Decrement σ_{noise} by factor of x
        end if
        Collect trajectories using old policy $\pi_{\theta_{old}}$
        Store transitions (s_t, a_t, r_t, d_t) in buffer B
    end for
    Compute advantage estimates $\hat{A}_t$ using GAE and value targets V_{target}
    for each update step $i = 1, \ldots, n_{\text{updates}}$ **do**
        Compute sampling ratio $r_t(\theta)$
        Compute total policy loss $L_{\text{policy}} = -L_{\text{clip}} - \beta L_{\text{entropy}}$ and
        value function loss L_{value}
        Compute gradients and update parameters:
        $\theta \leftarrow \theta - \eta \nabla_\theta L_{\text{policy}}$
        $\phi \leftarrow \phi - \eta \nabla_\phi L_{\text{value}}$
    end for
    Update old policy weights: $\pi_{\theta_{\text{old}}} \leftarrow \pi_\theta$
    Update old value function weights: $\pi_{\phi_{\text{old}}} \leftarrow \pi_\phi$
    Clear buffer B
end for

- **Variable Exploration Noise (σ_{noise})**: Introduced an explicit exploration noise parameter (σ_{noise}) that decays over time, promoting higher exploration early in training and more exploitation later. Exploration is introduced by sampling actions from a Gaussian distribution parameterized by the predicted mean $\mu_\theta(s_t)$ and standard deviation σ_{i}. The exploration noise, which is defined by a hyperparameter σ_{noise}, acts as a standard deviation, allowing the policy to sample varied actions around the mean rather than producing deterministic outputs. The action distribution in our algorithm implementation is defined as:

$$\Sigma_\theta = \text{diag}(\sigma_\text{i}) \tag{4.1}$$

Where $\mu_\theta(s_t)$ is the mean of the action probability distributions, and Σ_θ is a diagonal covariance matrix constructed directly from the exploration standard deviation σ_θ, such that:

$$a_t \sim \pi_\theta(a_t \mid s_t) = \mathcal{N}(\mu_\theta(s_t), \Sigma_\theta) \tag{4.2}$$

To enhance exploration efficiency, we adopt a variable exploration noise strategy by modifying the standard deviation σ_noise over time. Unlike fixed noise levels, a time-varying standard deviation allows the agent to dynamically adjust its exploration behavior as training progresses. σ_i is assigned based on the current value of σ_noise, which is varied across training phases. Initially, a higher value of σ_noise encourages broad exploration by introducing more randomness into the actions. As training progresses and the policy improves, σ_noise is gradually reduced, resulting in more deterministic and refined actions that reflect the learned behavior of the agent.

- **Generalized Advantage Estimation (GAE)**: GAE[30] was originally proposed to reduce the high variance in advantage estimates while maintaining low bias. This thesis adopts the GAE advantage function, which guides the direction and magnitude of policy updates by quantifying the relative value of actions. In this framework, to improve the accuracy of advantage calculations during policy training, where feedback signals can be sparse or delayed and decisions must be made in rapidly changing environments, GAE [30] helps decrease the variance of gradient estimates while preserving bias within acceptable limits. By leveraging a weighted sum of temporal differences across multiple time steps, GAE captures the long-term impact of actions more effectively than simple advantage estimates.

  The advantage function using GAE is computed as:

$$\hat{A}_t = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l} \tag{4.3}$$

Where temporal difference δ_t is the residual given by:

$$\delta_t = r_t + \gamma V_\phi(s_{t+1}) - V_\phi(s_t) \tag{4.4}$$

The hyperparameter $\lambda \in [0, 1]$ controls the bias-variance tradeoff, where higher λ values reduce bias but increase variance. By integrating GAE, our PPO implementation ensures more stable training, reducing high variance in gradient estimates while retaining sufficient exploration for improved policy learning, and $\gamma \in [0.95, 0.99]$ is the discount factor, balancing shortterm and longterm rewards.

- **Total Loss Function**: It is defined as the aggregation of clipped surrogate loss, value loss, and an entropy regularization term.

Value Function Loss To ensure accurate estimation of state values, PPO includes a squared-error loss term for the value function:

$$L^V(\theta) = \mathbb{E}_t \left[\left(V_\theta(s_t) - V_t^{\text{target}} \right)^2 \right] \tag{4.5}$$

where $V_\theta(s_t)$ is the predicted value and V_t^{target} is typically the empirical return $\hat{R}_t$.

Entropy Bonus To promote sufficient exploration during training, an entropy regularization term is added:

$$L^S(\theta) = \mathbb{E}_t \left[\beta \cdot \mathcal{H}(\pi_\theta(\cdot|s_t)) \right] \tag{4.6}$$

where $\mathcal{H}$ denotes the entropy of the probability distribution and β is a scaling coefficient.

Total PPO Loss

$$L(\theta) = L^{\text{CLIP}}(\theta) - c_1 L^V(\theta) + c_2 L^S(\theta) \tag{4.7}$$

where coefficients c_1 and c_2 are weighting the value loss and entropy, respectively. This loss function enables our policy to achieve good performance across a wide range of reinforcement learning tasks while maintaining simplicity and ease of implementation [28].

4.0.2.3 Trajectory Collection and Policy Update

It is essential to collect trajectories consisting of the agent's states, actions, and the corresponding rewards obtained through interaction with the environment. These collected trajectories are then utilized to optimize the policy using the clipped objective, enabling stable updates while leveraging the estimated advantages from the agent's experiences. For this purpose, the CARLA virtual environment, as discussed in section 2.8, is used to run the RL agent. As a virtual training environment, 'town 2' of CARLA is chosen due to its urban construct and road patterns, such as elbow turns and T-junctions, which are commonly found in real-world urban towns, as they introduce complex navigation challenges. The agent executes its policy $\pi_{\theta_{\text{old}}}$ in the environment for T time-steps to collect tuples of the form $(s_t, a_t, r_t, d_t, V(s_t; \theta_v))$, which are stored in experience buffer on episode basis. These trajectories include environmental feedback based on the agent's actions, incorporating factors such as safety, lane-keeping, and driving efficiency. The agent receives a reward signal derived from these factors, with the reward function guiding the learning process by reinforcing desirable actions and discouraging undesirable ones. Each episode follows a predefined route in the Town 2 environment, and successful completion is marked by the agent reaching its destination. Episodes are terminated upon meeting predefined termination conditions such as collisions, deviation from the lane, or exceeding the maximum allowed episode length. Agent is run for N such episodes, gathering the data to form a observation-action dataset for training and used to compute the Policy and the value loss (as described in Section 4.0.2.2) to update both the actor and critic networks by calculating gradients. The new policy is updated using the clipped surrogate objective, ensuring that it does not deviate significantly from the old policy and thereby keeping stable training.

4.0.3 Post-training Quantization

To enable the efficient inference of training models on the edge devices, we incorporate post-training quantization techniques for both the Variational Autoencoder (VAE) and the actor-network. These compression methods are adopted to make the models computationally efficient for the edge. The uncompressed TensorFlow (TF) model is quantized to TensorFlow Lite (TFLite) models such as Floating-point16 (FP16) and Integer8 (INT8). These compression methods significantly reduce memory footprint

and inference latency with negligible degradation of performance, enabling efficient and stable policy execution with adequate accuracy on low-power edge hardware.

4.0.4 Edge prediction with Processor-in-the-loop

The Processor-in-the-loop (PIL) framework, illustrated in Fig. 4.4, integrates both the server (simulation node) and the edge nodes in a coherent real-time interaction loop. An Ethernet-based communication is established to interconnect these two nodes. Each node performs remote procedure calls (RPCs) to enable seamless bidirectional data exchange in real-time. The simulation node transmits serialized observation data to the edge, while the edge node processes the input and returns the computed action values. This closed-loop interaction ensures synchronized execution between the environment and the model, allowing for the assessment of real-time inference performance on the edge device. By continuously exchanging serialized data packets over Ethernet, PIL enables rigorous testing of the real-time feasibility, performance, latency, and computational efficiency of deep reinforcement learning (DRL) models deployed onto edge devices. The detailed explanation of each node is discussed below:

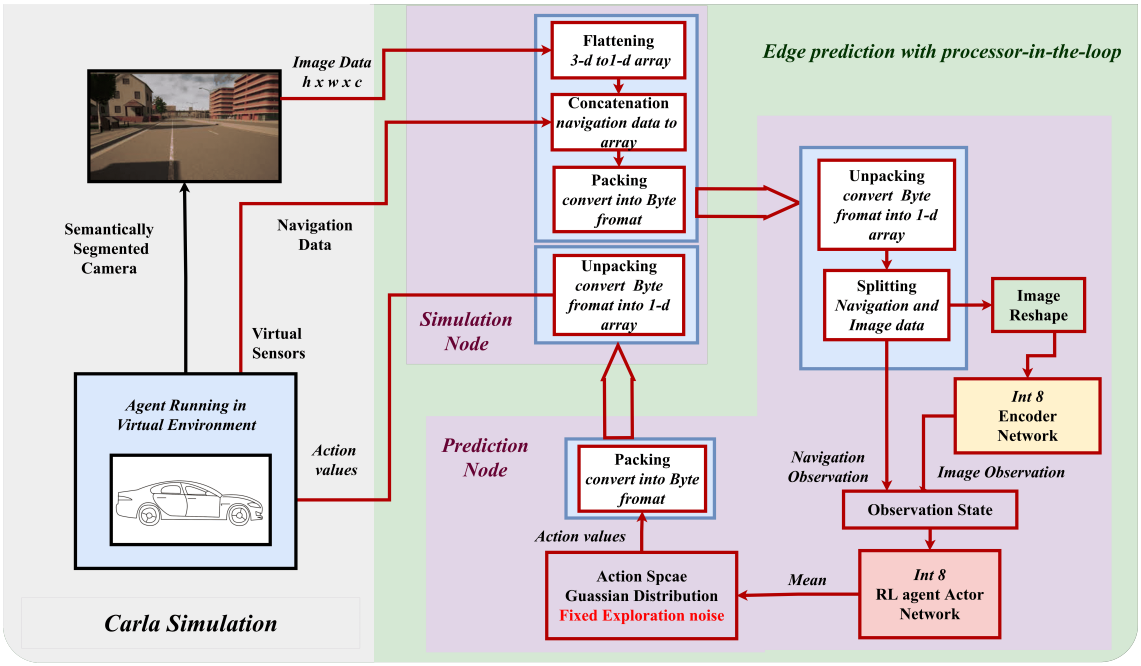


Figure 4.4: Prediction using Edge device (PIL)

4.0.4.1 Simulation Node

Simulation node like CARLA requires high-end systems with GPUs due to their need for real-time rendering of complex 3D environments and parallel processing of sensor data for autonomous driving tasks. The simulation node hosts the CARLA environment, generating real-time semantically segmented (SS) images along with navigation-related data such as vehicle position, velocity, orientation, and lane adherence. These data collectively represent the vehicle's observation of its environment at each timestep. The simulation node performs data pre-processing and serialization of data to convert these observations into structured binary packets optimized for fast transmission over Ethernet. In return, the simulation node receives the binary packets of action values computed by the edge node. These action values are unpacked and applied to the simulated vehicle within CARLA, effectively closing the loop for real-time control and enabling accurate evaluation of edge-based decision-making performance.

4.0.4.2 Prediction Node

This section outlines the deployment process of the VAE and the actor-network on an edge device for real-time inference. The VAE and the actor-network receive serialized observation data from the simulation node. However, unlike the training framework, the image data is serialized to a linear array and packed into a byte format, which is suitable for transmission. Upon receiving this packed data, the edge node then unpacks it and reshapes it into the format suitable for the VAE, which is $160 \times 80 \times 3$. The VAE then encodes the image into a compact latent vector, which is the image observation, effectively capturing the essential visual features of the driving environment in a lower-dimensional representation. This latent vector is then concatenated with the navigation observation vector, forming a complete observation state. The actor-network is pre-trained and processes the observation state to generate mean action values. These values parameterize a Gaussian distribution defined using a fixed exploration noise, from which an action is sampled. The final action output is then serialized and transmitted back to the simulation node.

Chapter 5

Experiment Setup and Results

The experiments are conducted on a server equipped with an NVIDIA Quadro GPU. The software environment includes Python 3.7, CARLA 0.9.8, and TensorFlow 2.11. Moreover, for edge prediction, Raspberry Pi 4 Model B is used, which has the configuration of a quad-core 64-bit ARM Cortex-A72 processor and 8GB RAM.

Table 5.1: Training Parameters for VAE

Hyperparameter	**Value**
Learning rate	1×10^{-4}
Batch size	64
Loss Function	MSE
Optimizer	Adam
Activation	ReLU
Latent space dimension z_{dim}	95
Epochs	10

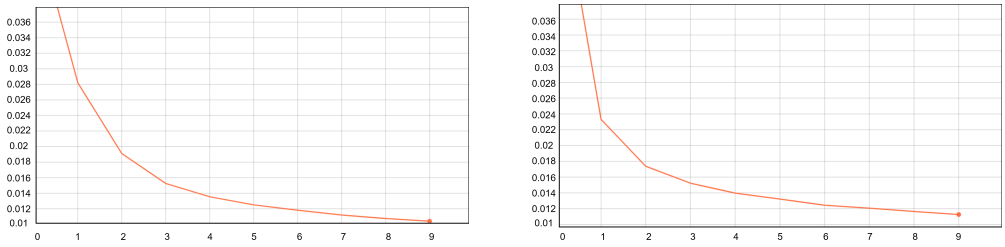
Table 5.2: Training Parameters for RL

Hyperparameter	**Value**
Initial Exploration Noise σ_{noise}	0.4
Advantage Estimation	GAE
Learning Rate	1×10^{-4}
Batch Size	1
Policy Clip ϵ	0.2
Discount Factor γ	0.99
Lambda λ	0.95

Hyperparameter settings: Table 5.1 summarizes the hyperparameters used for training the Variational Autoencoder (VAE). The Mean squared error (MSE) is chosen as the loss function as it effectively reduces the error between the input and output images. The learning rate of 1×10^{-4} is selected to provide a balanced trade-off between convergence speed and training stability, preventing overshooting while allowing the model to progress during updates enhanced with the Adam optimizer for its adaptive nature making it well-suited for deep neural networks. Moreover, a batch size of 64 with 10 epochs is applied to leverage mini-batch training, which improves generalization.

Similarly, in table 5.2, hyperparameters used for training the PPO-based RL agent are summarized. The training is conducted with a learning rate of 1×10^{-4} and an

initial exploration noise of 0.4, chosen to encourage exploration in the beginning stages of training. This exploration noise is gradually reduced over time to shift the agent's behavior toward exploitation as learning progresses. A batch size of 1 is used to accommodate the variable number of training samples generated per episode in the interactive, episodic training process. A policy clip [28] of 0.2 is also used, as it has proven effective in stabilizing training in other reinforcement learning-based approaches for continuous control. Generalized Advantage Estimation (GAE) [30] is employed with discount factor $\gamma = 0.99$ to effectively capture long term rewards and smoothing parameter $\lambda = 0.95$ for enabling the agent to compute more stable and informative advantage values by balancing bias and variance in advantage estimation, which is particularly good in autonomous driving scenarios where long term rewards are impacted by present action.



(a) Training Loss (b) Validation Loss

Figure 5.1: Training and validation loss of the VAE

Results:

Before experimenting with the RL agent, the VAE is trained using 12,000 SS images collected from 'town 02' in 3D CARLA simulator. Training is conducted using the hyperparameters listed in Table 5.1. During inference, only the encoder component of the VAE is utilized, as reconstruction through the decoder is not necessary for policy decision-making. The training performance of the VAE, as shown in Figure 5.1, represents two graphs as training loss and validation loss. According to the graphs, training loss and validation loss are steadily decreasing and remain within a desirable range, indicating that the model learns compact latent representations properly while minimizing the reconstruction error.

After the VAE, the RL agent is trained using the PPO algorithm, while keeping the weights of the encoder fixed to ensure consistent feature encoding. To train the RL model, we determine a 500-meter route in Town 2 of the CARLA simulator, as

shown in Fig. 5.2, which includes elbow turns and straight road segments. At each intersection, the agent is directed to follow the straight path to maintain the route consistency. To execute this task, an interactive episodic approach is employed, guided by a dense reward function. In this function, rewards are defined by termination conditions to guide the learning process, particularly policy diverging from diverging by ensuring desired driving behaviors. In this setup, the episode termination criteria are defined below:

- Occurrence of infractions, such as collisions
- Deviation from the driving lane exceeds 3 m
- Exceeding the set velocity range 15 km/h to 35 km/h
- Reaching the maximum allowed episode length of 7500 timesteps

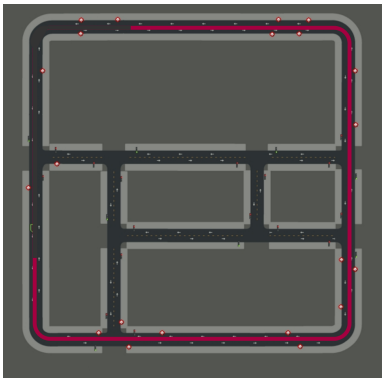
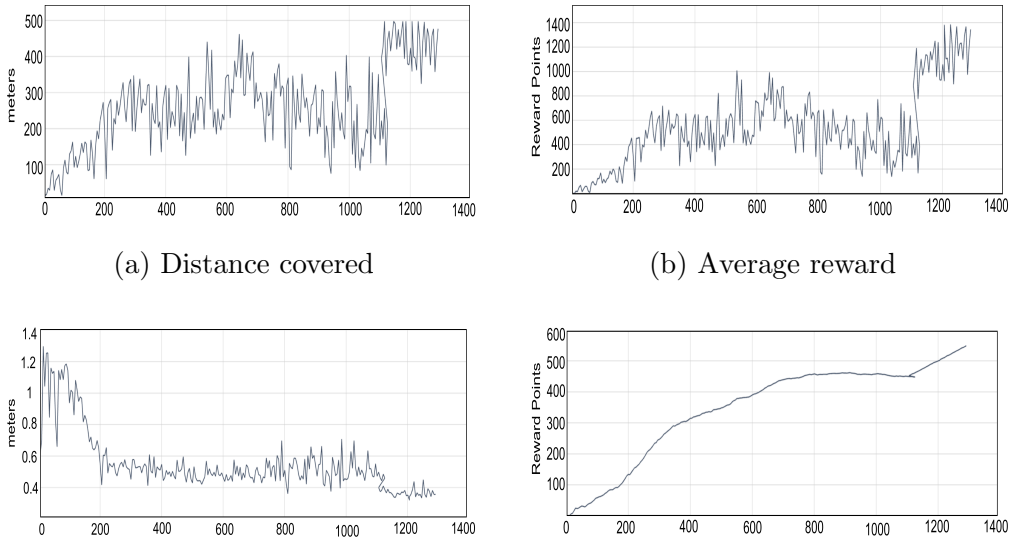


Figure 5.2: Predetermined route of Town 2 in CARLA simulator

Using these termination conditions, we train the agent for a total of 1200 episodes, with exploration noise gradually reduced by 0.05 for every 300 episodes from an initial value of 0.4. This scheduling enables a smooth transition from exploration to exploitation. To accelerate learning, a checkpoint is saved every 100 meters. If the agent reaches a terminal state, such as a collision or deviation from the lane, it is reset to the most recent checkpoint. This strategy enables the agent to rapidly return to challenging segments of the route, facilitating faster convergence and focused learning in complex driving scenarios. As training progresses, the average reward per episode and cumulative reward increase, while the lane deviation gradually decreases,



(a) Distance covered

(b) Average reward

(c) Lane Deviation from center

(d) Cumulative Reward

Figure 5.3: Training Results of RL Agent on GPU per episode

Table 5.3: Model Memory Requirements Under Different Quantization Levels

Model	***TF Model (float32)***	**Floating-point16 (*FP16*)**	**Integer8 (*INT8*)**
Actor	988 KB	456 KB	233 KB
Encoder	52.6 MB	26.8 MB	13.45 MB

as shown in Figure 5.3. Additionally, the PPO agent demonstrates the ability to reach the destination of 500 meters, confirming successful learning and policy convergence.

After the training phase, we move towards edge prediction by deploying the model on the edge. As mentioned before, we work on the Raspberry Pi 4 Model B, which has low memory and low processing power without a GPU or TPU. As a result, it is difficult for real-time inference while maintaining a high level of accuracy. Therefore, we try to achieve a balance between accuracy and latency. To fulfill this goal, we adopted the post-training quantization provided by TensorFlow on both the DRL (only Actor network) and the Variational Autoencoder (VAE) models. Given the limited memory resources of edge devices, minimizing model size is a key objective. Table 5.3 presents the memory requirements under different numerical precisions. Both models showed a reduction in size after quantization, with the Integer8 (*INT8*) version occupying the least memory. These results confirm the effectiveness of quantization in producing memory-efficient models suitable for real-time inference on edge

hardware. To evaluate the performance of the quantized models, a dataset of 10 episodes is generated using the 32-bit GPU-trained model. Each episode contains semantic segmentation (SS) images, navigation data, the predicted mean from the model, and the corresponding computational latency. This dataset is then used to test the *Floating-point16 (FP16)* and *Integer8 (INT8)* quantized models on the edge. For each episode, the inference time and the RMSE between the outputs of quantized models and the original 32-bit GPU model are recorded for 10 episodes. This comparison helps us understand how well the quantized models perform in terms of both speed and accuracy.

Table 5.4: Root Mean Square Error

	GPU	**Edge**	
Ep.	***TF-32***	***FP16***	***INT8***
1	0.02379	0.02379	0.02412
2	0.01469	0.01469	0.01500
3	0.00795	0.00795	0.00840
4	0.00252	0.00252	0.00363
5	0.00128	0.00128	0.00271
6	0.01585	0.01585	0.01613
7	0.00281	0.00281	0.00373
8	0.00000	0.00000	0.00302
9	0.00484	0.00484	0.00574
10	0.00272	0.00272	0.00352
Avg.	0.00764	0.00764	0.00860

Table 5.5: Inference Time (in msec)

	GPU	**Edge**	
Ep.	***TF-32***	***FP16***	***INT8***
1	26.28	76.14	36.97
2	26.80	76.37	37.03
3	27.53	74.79	36.94
4	28.06	75.54	36.98
5	33.71	74.88	36.93
6	31.31	75.15	36.87
7	30.41	75.04	36.96
8	60.32	75.05	36.97
9	31.67	75.11	37.01
10	34.00	76.47	35.87
Avg.	33.01	75.45	36.85

The above tables demonstrate the comparison between the quantized models and the original TF-32 model, without any compression, in terms of inference time and accuracy. Table 5.4 represents the RMSE results of models before and after quantization. The Floating-point16 (*FP16*) model preserves the same accuracy as the original TF-32 model, maybe because the reduced precision affects the weights rather than the model structure. However,*INT8* shows a slight drop in accuracy, which indicates the quantization error due to lower precision. Furthermore, the table 5.5 depicts the latency in the inference phase, showing that the TF-32 model takes approx. 33 ms to process one sample in GPU while *FP16* takes only 75 ms on a low-powered edge device without any GPU unit. In contrast, *INT8* takes approx. 36 ms to process one sample, which shows comparatively very little inference time with adequate accuracy.

Based on these results, the *INT8* quantized model is selected for the Processor-in-the-Loop (PIL) evaluation, as it provides an optimal trade-off between computational efficiency and prediction accuracy.

Table 5.6: GPU Prediction Results over 10 Episodes

Ep.	**Total Time (s)**	**Reward**	**Dist. (m)**	**Latency per action (ms)**	**Speed (km/h)**
1	100.48	1722.09	500	40.20	17.93
2	80.95	1364.26	395	40.36	17.57
3	102.07	1742.96	500	41.23	17.64
4	80.04	1289.90	393	42.28	17.68
5	101.50	1668.55	500	42.24	17.75
6	83.69	1276.41	400	43.87	17.21
7	100.48	1547.93	500	44.24	17.93
8	101.21	1561.83	500	44.98	17.78
9	100.17	1491.43	500	46.19	17.96
10	100.54	1380.66	500	49.18	17.89
Avg.	95.91	1502.50	478.80	43.88	17.73

In the edge prediction using PIL, the *INT8*-encoder model as well as the *INT8*-actor model are deployed onto the edge device and evaluated for their performance by interfacing CARLA in a closed-loop. Action values are computed using the output of the actor model and standard deviation (σ_{i}) parameterized by exploration noise. This noise (σ_{noise}) is fixed at 0.2 during evaluation, corresponding to the final value reached at the end of the training phase. Evaluation is carried out over 10 episodes with performance metrics such as distance covered, average speed, total time taken to reach the set distance, reward per episode, and inference latency, which includes communication and computational time. These results are compared against the GPU baseline to assess the performance of *INT8* models. The overall loop latency for edge prediction averaged 118.45 milliseconds, with inference time being 36.85 milliseconds per sample, as shown in Table 5.7. The agent achieved an average distance of 407.4 meters and an average reward of 1236.97 during these runs. In comparison with the GPU results illustrated in Table 5.6, the agent deployed on the edge covered approximately 85.1% of the average distance and achieved around 82.3% of the average reward of GPU metrics. Notably, the overall loop latency on the edge is dominated by communication delay (81.6 ms), which is significantly higher than

Table 5.7: Edge prediction *int8* Results with PIL over 10 episodes

Ep.	**Total Time (s)**	**Reward**	**Dist. (m)**	**Latency per action(ms)**	**Speed (km/h)**
1	95.52	1183.49	327	112.47	12.32
2	143.70	1455.82	500	113.88	12.50
3	95.14	952.43	319	107.91	12.07
4	150.56	1362.90	500	121.62	11.95
5	62.29	799.04	212	116.15	12.25
6	120.79	1276.91	472	129.97	14.04
7	83.57	940.33	287	123.95	12.36
8	127.90	1678.00	500	125.84	14.15
9	132.89	1568.00	500	108.74	13.61
10	118.08	1145.80	457	128.99	13.93
Avg.	113.41	1236.97	407.4	118.45	12.92

the model's computational inference time (36.85 ms). The above analysis shows that this work is executed on the edge, which includes the PIL setup. This signifies that models like DRL can be used for sequential decision-making in real-time prediction.

5.1 Comparison of Related Work on DRL Applications for Autonomous Driving

Table 5.8 presents a comparative summary of key research works focused on DRL for driving tasks. The comparison covers experimental setups, algorithms used, primary control targets, whether edge implementation was considered, availability of timing analysis from sample acquisition to action execution, and notable comments. Notably, most prior works do not perform timing analysis or focus on edge feasibility, which is a key aspect addressed in this thesis. In contrast, this work specifically addresses deployment on edge devices, provides detailed latency analysis from sensing to action, and uses a processor-in-the-loop (PIL) setup to validate real-time performance.

Paper	**Exp. Setup**	**Algo.**	**Targets**	**Edge Imp.**	**Time analysis**	**Comments**
[4]	NVIDIA DevBox	CNN	Lane Following	-	-	Used CNN for autonomous steering control; Neural network used is complex
[33]	CARLA simulator	DQN	Collision Avoidance and path efficiency	-	-	Comparison between DQN and PPO in various driving Scenarios, Performance metrics includes Reward
[35]	CARLA Simulator	DDPG	Lane Following, Collision Avoidance,Auto braking	-	-	Empirical analysis on continuous control,Performance metrics includes episodic Reward
[26]	CARLA and NVIDIA GPU	PPO	Lane Following, Collision Avoidance, set Velocity Range, Path efficiency	-	-	PPO based DRL training on High end GPU, Performance metrics includes episodic Reward, lane deviation, distance traveled
This Work	CARLA, NVIDIA Quadro, Rpi, PIL	PPO	Lane Following, Collision Avoidance, set Velocity Range, Path efficiency	Rpi	yes	Model trained using PPO based DRL algorithm and Prediction with deployed model on EDGE in PIL loop, Performance metrics includes episodic Reward, lane deviation, distance traveled, RSME, Inference time, Overall latency(comp. + Comm. time) across various TensorFlow and lite models

Table 5.8: Comparison of DRL Applications on Edge or Simulated Platforms for Autonomous Driving

Chapter 6

Conclusions

The proposed framework connects the ADS simulation environment with a low-power edge device for real-time evaluation using a Processor-in-the-Loop (PIL) setup. The VAE-DRL model, trained with the PPO algorithm on a GPU, is quantized further for deploying on the edge device to enable continuous sequential decision-making in a real-time feedback loop. The evaluation of this approach uses training performance metrics such as episode-wise reward, distance traveled, and completion time. However, this is to be noted that choosing the appropriate hyperparameters for training the RL agent turns out to be moderately complex. Particularly due to a variable number of samples per episode, which varies as training progresses, it also made it difficult to select a proper batch size. Moreover, the initial exploration noise is town-specific and depends on different road layouts and navigation complexities. During training in certain instances, the DRL model produced *NaN* values for the mean output, which is found to be caused by large values in the latent space generated by the VAE. To address this, value clipping is applied to the latent space representations.

In addition to the above issues, this thesis raises a point regarding the compatibility of the quantized PyTorch model with ARM-based architecture. According to [26], initially, we followed the GitHub repository that provided a PyTorch-based implementation of the DRL model, which is used for training. Nevertheless, during inference on the edge device, the *INT8* quantized PyTorch model is not able to run on the Raspberry Pi due to its incompatibility with ARM-based architecture. Due to these deployment challenges, we adopted TensorFlow. It is important to note that, before performing real-time inference using the PIL, a dataset is generated using the GPU-trained model. This dataset is then used to run inference on the edge with quantized models.

Furthermore, moving onto the inference phase, prediction performance is assessed through RMSE loss and inference latency by comparing edge results with the GPU-based model, which served as the baseline model. Our variable exploration rate training strategy enabled a smooth transition from exploration to exploitation. As a result, the model is converged and exhibits driving behavior that is aligned with the defined objectives. The results from edge prediction using the PIL framework demonstrate that deploying DRL models on low-power edge devices for autonomous driving is feasible, as the computational latency of the *INT8* quantized model is comparable to that of the *TF32* model running on a GPU. Despite the communication delay introduced by the PIL loop, our DRL agent is still able to perform effectively, maintaining reliable decision-making and driving behavior during real-time inference on the edge device.

This thesis explores and validates the feasibility of the practical deployment of DRL-based autonomous driving models on low-power edge devices using PIL evaluation. Merging Reinforcement Learning with the autonomous driving domain by implementing it in the actual scenario and tackling important issues is the first work that provides the foundation for end-to-end prediction of deep reinforcement learning on the edge.

Chapter 7

Future work

This work presents an approach to Autonomous Driving Systems (ADS) using Deep Reinforcement Learning (DRL) on edge devices, a relatively new and promising direction with vast potential for future extensions and real-world deployment.

- Future work will focus on extending this framework to multi-agent systems and real-world vehicle integration to further assess the reliability of Deep Reinforcement Learning on the edge for autonomous driving.
- Future research can explore performing training directly on edge devices rather than relying on server-based infrastructure, which would allow for on-device learning, reduce latency, and enhance real-time adaptability.
- Instead of using a fixed or manually-tuned variable exploration noise, integrating a trainable exploration noise parameter could allow the agent to autonomously adjust exploration during training, potentially leading to more efficient policy convergence.
- Incorporating elements of deterministic policy learning from algorithms like DDPG into the PPO framework may help improve stability and action consistency, especially in safety-critical driving tasks, while maintaining the robustness and simplicity of PPO.
- The action space can be expanded to include additional crucial control signals, such as braking control, enabling more comprehensive vehicle actuation and better handling of complex driving scenarios.

- Deploying more sophisticated neural network architectures, including attention mechanisms or transformer-based models, could improve the agent's ability to capture temporal dependencies and extract more meaningful features from high-dimensional inputs.
- Moreover, since RL depends on server-based training, it introduces potential security concerns. Therefore, we aim to provide a secure framework on the edge for RL implementation.
- We can also explore RL integration in a secure vehicle-to-vehicle ($V2V$) scenario in the $6G$ network.

Bibliography

[1] Dario Amodei et al. Concrete problems in ai safety. *arXiv preprint arXiv:1606.06565*, 2016.

[2] Eugenia Anello. Variational autoencoder with pytorch. *medium.com/dataseries/variational-autoencoder-with-pytorch-2d359cbf027b*, 2021. [Online]. Available: `https://medium.com/dataseries/variational-autoencoder-with-pytorch-2d359cbf027b`.

[3] Mariusz Bojarski et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.

[4] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016. Version 1, 25 Apr 2016.

[5] Chenyi Chen et al. Deepdriving: Learning affordance for direct perception in autonomous driving. In *ICCV*, pages 2722–2730, 2015.

[6] Felipe Codevilla et al. End-to-end driving via conditional imitation learning. In *ICRA*, 2018.

[7] Felipe Codevilla, Matthias Müller, Alexey Dosovitskiy, Antonio López, and Vladlen Koltun. Learning by cheating. In *Conference on Robot Learning*, pages 1–15. PMLR, 2019.

[8] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. Carla: An open urban driving simulator. *Conference on Robot Learning (CoRL)*, pages 1–16, 2017.

[9] Vincent Francois-Lavet, Peter Henderson, Riashat Islam, Marc G. Bellemare, and Joelle Pineau. An introduction to deep reinforcement learning. *arXiv preprint arXiv:1811.12560v2*, 2018. Submitted on 30 Nov 2018 (v1), last revised 3 Dec 2018 (this version, v2).

[10] David González, Joshué Pérez, Vicente Milané, and Fawzi Nashashibi. A review of motion planning techniques for automated vehicles. *IEEE Transactions on Intelligent Transportation Systems*, 17(4):1135–1145, 2016.

[11] Kaiming He et al. Mask r-cnn. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 40(2):298–314, 2017.

[12] SAE International. Taxonomy and definitions for terms related to driving automation systems for on-road motor vehicles. *SAE Standard J3016*, 2018.

[13] Sanskar Jadhav, Vedant Sonwalkar, Shweta Shewale, Pranav Shitole, and Samarth Bhujadi. Deep reinforcement learning for autonomous driving systems. *International Journal for Multidisciplinary Research (IJFMR)*, 6(5), September-October 2024. IJFMR240528518.

[14] Alex Kendall, James Hawke, David Janz, et al. Learning to drive in a day. *arXiv preprint arXiv:1807.00412*, 2018.

[15] Diederik P. Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[16] Jens Kober, J. Andrew Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32(11):1238–1274, 2013.

[17] Jan Koutník, Jürgen Schmidhuber, and Faustino Gomez. Evolving deep unsupervised convolutional networks for vision-based reinforcement learning. *arXiv preprint arXiv:1406.6366*, 2014.

[18] Jesse Levinson, Jacob Askeland, Jonathan Becker, Jennifer Dolson, David Held, Sebastian Kammel, J. Zico Kolter, Dirk Langer, Oliver Pink, Vadim Pratt, and Sebastian Thrun. Towards fully autonomous driving: Systems and algorithms. *IEEE Transactions on Intelligent Transportation Systems*, 13(4):1355–1369, 2011.

[19] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[20] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Tim Harley, Timothy Lillicrap, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *International conference on machine learning*, pages 1928–1937, 2016.

[21] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[22] Ramesh Mohan, Robert Kolodziejski, and Henry Lee. Towards secure and efficient edge computing for autonomous driving: Challenges and solutions. *IEEE Vehicular Technology Magazine*, 14(2):27–35, 2019.

[23] Pavlo Molchanov, Arun Mallya, Stephen Tyree, Iuri Frosio, and Jan Kautz. Importance estimation for neural network pruning. *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 11264–11272, 2019.

[24] Andrew Y Ng, Daishi Harada, and Stuart J Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *ICML*, pages 278–287, 1999.

[25] Brian Paden, Michal Čáp, Sze Zheng Yong, Denis Yershov, and Emilio Frazzoli. A survey of motion planning and control techniques for self-driving urban vehicles. *IEEE Transactions on Intelligent Vehicles*, 1(1):33–55, 2016.

[26] Asad Idrees Razak. Implementing a deep reinforcement learning model for autonomous driving. *IEEE Transactions on Intelligent Transportation Systems*, 2024.

[27] Joseph Redmon et al. You only look once: Unified, real-time object detection. *CVPR*, 2016.

[28] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[29] John Schulman, Sergey Levine, Philipp Moritz, Michael Jordan, and Pieter Abbeel. Trust region policy optimization. *arXiv preprint arXiv:1502.05477v5*, Apr 2017. Department of Electrical Engineering and Computer Sciences, University of California, Berkeley.

[30] John Schulman, Philipp Moritz, Sergey Levine, Michael I Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. In *International Conference on Learning Representations (ICLR)*, 2016.

[31] Wilko Schwarting, Javier Alonso-Mora, and Daniela Rus. Planning and decision-making for autonomous vehicles. *Annual Review of Control, Robotics, and Autonomous Systems*, 1:187–210, 2018.

[32] Arjit Sharma and Sahil Sharma. Wad: A deep reinforcement learning agent for urban autonomous driving. *arxiv.org/abs/2108.12134*, 2021. [Online]. Available: `https://arxiv.org/abs/2108.12134`.

[33] Rishabh Sharma and Prateek Garg. Optimizing autonomous driving with advanced reinforcement learning: Evaluating dqn and ppo. *IEEE*, 2024. IEEE Xplore Part Number: CFP24V90-ART.

[34] Weisong Shi, Jie Cao, Qun Zhang, Youhu Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016.

[35] Sanjna Siboo, Anushka Bhattacharyya, Rashmi Naveen Raj, and S. H. Ashwin. An empirical study of ddpg and ppo-based reinforcement learning algorithms for autonomous driving. *arXiv preprint*, November 2023. Corresponding author: Rashmi Naveen Raj (rashmi.naveen@manipal.edu), Supported by Manipal Academy of Higher Education.

[36] Alexander Stens, Iversen Szewczyk, Frank Lindseth, and Gabriel Kiss. Ai-agents trained using deep reinforcement learning in the carla simulator. *Journal of Autonomous Systems Research*, 2022.

[37] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2nd edition, 2018.

[38] Charles Thorpe et al. Toward autonomous driving: The cmu navlab. part i. *IEEE Expert*, 1991.

[39] Sebastian Thrun et al. Stanley: The robot that won the darpa grand challenge. *Journal of Field Robotics*, 2006.

[40] Marcus L. Vergara and Frank Lindseth. Accelerating training of deep reinforcement learning-based autonomous driving agents through comparative study of agent and environment designs. *arXiv*, 2019. [Online]. Available: `https://arxiv.org/abs`.

[41] Fei Wang, Meng Jiang, Kai Qian, et al. Efficient and interpretable neural attention mechanisms for autonomous driving. *IEEE Transactions on Neural Networks and Learning Systems*, 31(11):4859–4872, 2019.

[42] Christopher John Cornish Hellaby Watkins. *Q-learning*. Phd thesis, King's College, Cambridge, 1992.

[43] Bernhard Wymann, Eric Espié, Christian Guionneau, et al. Torcs, the open racing car simulator. *Open-Source Software*, 4(1), 2014.

[44] D. A. Yudin, A. Skrynnik, A. Krishtopik, I. Belkin, and A. I. Panov. Object detection with deep neural networks for reinforcement learning in the task of autonomous vehicles path planning at the intersection. *Optical Memory and Neural Networks*, 28(4):283–295, 2019.

[45] Yiming Zeng, Zhongli Yan, and Shuang Wang. Distributed deep learning framework for autonomous vehicles with edge computing. *IEEE Access*, 7:102108–102116, 2019.

[46] Julius Ziegler et al. Making bertha drive—an autonomous journey on a historic route. In *IEEE Intelligent Transportation Systems*, 2014.