ENPM 667 - Project 1

A Technical Report on Deep Multi-agent Reinforcement Learning for Highway On-Ramp Merging in Mixed Traffic

(Dong Chen , Mohammad R. Hajidavalloo , Zhaojian Li , Kaian Chen , Yongqiang Wang , Longsheng Jiang3 , Yue Wang)

Kunj Golwala UID: 121118271

Shreya Kalyanaraman UID: 121166647

A. James Clark School of Engineering (Robotics)
University of Maryland, College Park, MD - 20742

Table of Contents

List of Figures

List of Tables

A Technical Review of Deep Multi-agent Reinforcement Learning for Highway On-Ramp Merging in Mixed Traffic

Dong Chen[1], Mohammad R. Hajidavalloo[2], Zhaojian Li[3], Kaian Chen[4], Yongqiang Wang[5], Longsheng Jiang[6], and Yue Wang[7]

ABSTRACT

The results of Dong Chen et al.'s "Deep Multi-agent Reinforcement Learning for Highway On-Ramp Merging in Mixed Traffic" investigation are presented and replicated in this work. The report addresses on-ramp merging in surroundings populated by human-driven vehicles (HDVs) and autonomous vehicles (AVs). Viewed as a multi-agent reinforcement learning (MARL) problem, a distributed architecture is used to encourage adaptive cooperation among autonomous cars under different traffic conditions. Important improvements cover parameter sharing and a priority-based safety supervisor to enable safe, effective interactions among agents, hence lowering crash rates and enhancing general traffic flow. Furthermore, curriculum learning is used to progressively raise traffic complexity so that autonomous cars may control more challenging environments as the learning process advances. This framework shows great safety improvement over current MARL methods, hence improving traffic efficiency. This paper attempts to reproduce and explain their techniques.

Keywords: Reinforcement Learning, Multi-agent Systems, Autonomous Vehicles

I. INTRODUCTION

Technological advancements like Tesla's Autopilot [1] have rendered autonomous vehicles (AVs) more prevalent and sophisticated. These vehicles, equipped with sensors, machine learning algorithms, and decision-making frameworks, have the potential to significantly enhance road safety and improve traffic efficiency. Public road acceptance of AVs has, nevertheless, accompanied an increase in AV-related incidents, particularly in demanding traffic conditions. Sometimes AVs' inability to dynamically and securely adapt to changing driving conditions causes these incidents. The issue gets more crucial in mixed traffic conditions, as AVs have to live with and interact with human-driven vehicles (HDVs).

Highway on-ramp merging calls for AVs to negotiate a merging lane in coordination with cars in the through lane. This coordination consists of complex interactions between AVs modifying their speed to squeeze into gaps and HDVs may be changing their behavior to allow merging cars. In a perfect world, AVs and HDVs would work in perfect harmony; AVs would use these gaps to merge effectively while HDVs changed their speed or location to produce merging gaps. Achieving this degree of synchronizing is difficult in reality, especially in mixed traffic where the presence of human drivers adds variable and uncertainty in driving actions. Rule-based have also been studied and these methods rely on pre-defined heuristics or protocols, such as simple if-then rules (e.g., speed upon observed gaps). These techniques have complicated and dynamic characteristics of actual traffic that test them even if they operate well in controlled or basic conditions. Conversely, optimization-based methods—such as Model Predictive Control (MPC)—predict and maximize vehicle interactions across a limited time horizon using mathematical models. MPC seeks to identify an ideal series of activities to get desired outcomes, such as decreasing travel time or avoiding crashes, viewing model traffic as a dynamic system and These methods are less useful for real-time decision-making under mixed traffic conditions, though, because they depend on extremely precise models of traffic dynamics and entail large computing overhead.

More recently, data-driven methods leveraging reinforcement learning (RL) have shown promise as a fix for problems with highway on-ramp merging. Agents in the field of machine learning sometimes referred to as RL learn to make decisions by interacting with an environment and receiving feedback. Iteratively through trial and error, an RL agent finds a policy—a mapping from observable states to actions—that maximizes its cumulative reward over time. Although traditional RL methods show potential, they usually rely on single-agent situations, seeing other automobiles on the road as either static or dynamic environmental obstructions rather than as interactive players with their intentions and behaviors. These approaches thus miss the complexity of vehicle-to-vehicle interactions under mixed traffic conditions. This work extends single-agent RL to a multi-agent reinforcement learning (MARL) paradigm addressing its limits. Many agents in MARL, including autonomous cars, learn policies not in an isolated manner but rather collectively so that their activities complement one another. Within the framework of highway on-ramp merging, this method enables several AVs to learn cooperative merging techniques considering the presence and behaviors of nearby HDVs. Parameter sharing—where all AVs employ a shared policy and value network—allows consistent decision-making and scalability across many traffic conditions, hence defining a fundamental component of this MARL system. Furthermore, included in the framework are localized rewards that motivate AVs to maximize interactions with their nearby neighbors, thereby fostering cooperation while preserving computational economy.

A priority-based safety supervisor is proposed to improve the MARL framework's safety and learning efficiency even more. This system serves as protection during the training process since it controls the agents' behavior depending on traffic circumstances and predefined safety limits, therefore reducing collision chances. Safety first guarantees that, even in simulated surroundings, learning does not affect the physical well-being of vehicles or passengers. Using curriculum learning—a training approach modeled on human learning processes—is another essential element. Starting with basic, low-traffic surroundings and subsequently adding more difficult settings, training scenarios in curriculum learning are progressively made more complex. This steady increase in complexity lets AVs develop strong decision-making skills gradually, adjusting to ever-challenging merging conditions as training advances.

The suggested approach is compared to state-of-the-art MARL methods, and testing reveals notable increases in traffic flow and safety criteria. The framework solves the important difficulties of highway on-ramp merging in mixed traffic by allowing AVs to develop adaptive, cooperative merging behaviors while considering the uncertain character of HDVs. This paper emphasizes the possibilities of MARL methods, improved with safety monitoring and curriculum learning, to open the path for safer and more effective integration of autonomous vehicles into challenging traffic systems.

II. BACKGROUND

II.1. Preliminaries of Reinforcement Learning

For any discrete time interval t:

- The agent observes the current state $s_t \in S$, where S denotes the state space encompassing all feasible environmental configurations.
- The agent selects an action $a_t \in A$, where A represents all permitted actions.
- The surroundings provide a reward $r_t \in \mathbb{R}$, which measures the instantaneous gain (or loss) of performing action a_t in state s_t.
- The agent transitions to the next state s_{t+1}, determined by the external dynamics.

The agent aims to maximize the total reward R_t, defined as follows:

$$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}, \tag{1}$$

- r_{t+k} is the reward received k steps into the future.
- $\gamma \in (0,1]$ is the discount factor, which determines the importance of future rewards relative to immediate rewards.

II.1.1. State-Action Value Function and Bellman Equation

A key concept in reinforcement learning is the **state-action value function**, also known as the Q-function. This function $Q^\pi(s_t, a_t)$ represents the expected cumulative reward starting from state s_t, taking action a_t, and then following the policy π thereafter:

$$Q^\pi(s_t, a_t) = \mathbb{E}\big[r_t + \gamma \mathbb{E}_{s_{t+1}\sim P(\cdot|s_t,a_t)}\big[\max_{a_{t+1}} Q^\pi(s_{t+1}, a_{t+1})\big]\big]. \tag{2}$$

- $P(s_{t+1} \mid s_t, a_t)$ is the transition probability that determines the likelihood of transitioning to s_{t+1} given s_t and a_t.
- $\mathbb{E}[\cdot]$ represents the expected value, which accounts for the stochasticity of rewards and state transitions.

The state value function, $V^\pi(s_t)$, measures the expected cumulative reward starting from state s_t and following policy π:

$$V^\pi(s_t) = \mathbb{E}[R_t \mid s_t = s]. \tag{3}$$

The relationship between V^π and Q^π is:

$$V^\pi(s_t) = \mathbb{E}_{a_t\sim\pi}[Q^\pi(s_t, a_t)]. \tag{4}$$

II.1.2. Actor-Critic Framework

Modern RL algorithms often use the actor-critic [2]framework, which combines two neural networks:

1. Critic Network: This network, parameterized by ϕ, learns the value function $V_\phi^\pi(s_t)$, which estimates the value of a given state.
2. Actor Network: This network, parameterized by θ, learns the policy $\pi_\theta(a_t \mid s_t)$, which maps states to actions.

The actor network is trained to maximize the following objective function. The objective function for the actor network in reinforcement learning is given as:

$$J_\theta^\pi = \mathbb{E}_{\pi_\theta}\left[\log \pi_\theta(a_t|s_t) A_t\right], \tag{5}$$

- J_θ^π (Objective Function): This represents the *objective function* for the actor network. The goal of the actor is to maximize J_θ^π, which corresponds to improving the policy π_θ to choose actions that yield higher rewards.
- $\mathbb{E}_{\pi_\theta}$ (Expectation): This denotes the *expectation* over the policy $\pi_\theta(a_t|s_t)$. It indicates averaging over all possible actions a_t that the policy could take in a given state s_t, weighted by their probability under the policy.

- $\log \pi_\theta(a_t|s_t)$ (Log-Probability of the Action): This is the *logarithm of the probability* of taking action a_t in state s_t under the policy π_θ. The logarithm helps stabilize and smooth the optimization process, aiding in gradient calculations.

- $\pi_\theta(a_t|s_t)$ (Policy): This represents the *policy* parameterized by θ. It is the probability distribution over actions a_t given a state s_t. The reinforcement learning process aims to optimize this policy to maximize rewards.

- A_t (Advantage Function): The advantage function quantifies how much better or worse the chosen action a_t is compared to the average action the policy would take in the same state s_t. It is defined as:

$$A_t = Q^\pi(s_t, a_t) - V^\pi(s_t), \tag{6}$$

- $Q^\pi(s_t, a_t)$: The state-action value function, which represents the expected return for taking action a_t in the state s_t and following policy π thereafter.

- $V^\pi(s_t)$: The state value function, which represents the expected return from state s_t under the policy π.

$$J_\theta^\pi = \mathbb{E}_{\pi_\theta}\left[\log \pi_\theta(a_t \mid s_t) A_t\right], \tag{7}$$

- $A_t = Q^\pi(s_t, a_t) - V^\pi(s_t)$ is the advantage function, which quantifies how much better taking action a_t is compared to the average action at state s_t.

The critic network is trained to minimize the value function loss, which measures the error in estimating the value of states:

$$J^\phi = \min_\phi \mathbb{E}_D\left[(R_t + \gamma V_\phi(s_{t+1}) - V_\phi(s_t))^2\right], \tag{8}$$

- D is an experience replay buffer that stores previously encountered experiences for training [3].

By combining the actor and critic networks, the agent can learn both the optimal policy and an accurate value function simultaneously, leading to more efficient learning.

II.2. Multi-agent Reinforcement Learning (MARL)

Over the past decade, the field of reinforcement learning has made remarkable progress, successfully addressing various sequential decision-making problems within machine learning. Most of the RL applications, such as game playing, robotics, and autonomous driving, naturally fall under the category of multi-agent reinforcement learning (MARL). In MARL, we have multiple agents interacting with the environment and among themselves. Largely, MARL algorithms can be classified as cooperative, non-cooperative, and partially cooperative based on the nature of interactions between the agents.

In a cooperative setting, all the agents collectively work towards a shared goal, whereas in a non-cooperative setting, they compete against each other to reach their goals first. The partial setting is a mix of both cooperative and competitive agents. In this paper, the focus is set on a cooperative setting, where the agents work together to achieve a shared goal. The agents learn that the best solution is to coordinate with each other rather than act in isolation.

1. **Independent Q-Learning:** An independent MARL framework where each agent learns independently and simultaneously treats every other agent as part of the environment. The agent will ignore the presence of other agents and conduct its learning and updates its policies as if it were the only agent in the environment. The algorithm is fully scalable, but it suffers from non-stationarity and partial observability. Let's break down and understand the three terms better in terms of independent Q-learning. Scalability here means that the algorithm can handle a large number of agents without much impact on the efficiency. Adding more agents doesn't complicate the learning process because the independent learning process doesn't involve any communication between agents. Non-stationarity is a problem relevant in the context of decentralized learning algorithms, which can be better illustrated through the following illustration. Imagine two robots that are learning to navigate in a shared environment. Each robot updates its Q values based on its experiences, and as each robot learns, it affects the environment in ways that influence the other robot's learning process. Partial observability is a fundamental challenge in MARL, because the agent perceives only a limited part of the environment based on its observations made locally. The agent must make decisions without knowing what the other agents can see or knowing the full state.

2. **Off-policy MARL algorithm:** In this off-policy [5] MARL algorithm, collaboration is achieved by utilizing a centralized critic network that estimates the state-action value using global observations and actions. A centralized critic network has access to the global state, which allows the critic to judge how the agents' joint action affects the overall objective. It is used during training that handles non-stationary behavior , by stabilizing learning considering the joint impact of all agents' actions. This network is not used during deployment, because once trained, all the agents use their own local policy based on their local observations.

3. **Adaptive Communication and Reward Scaling:** These terms refer to techniques that improve the effectiveness of agent interactions during training. What adaptive communication means is the agent decides when, what and how to share information. Reward Scaling helps the agent focus on what really matters around them. It helps them silence the rewards from faraway neighbors and focus on those who directly impact them. By incorporating such techniques it is possible to reduce unnecessary communication.

The above examples work well with an increasing number of agents, but they they might not be as efficient in a dynamic real-world environment.The reason behind this being, the MARL algorithms assume a fixed communication topology, where the connection remains constant between the agents. In real-world situations, the communication links may change, and the algorithm has to be retrained to handle the changes. There is an increasing need for algorithms that prioritize decentralized adaptability, transferable policies, and online learning to handle dynamic situations. In MARL, parameter sharing is widely used with homogeneous agents, which are agents with similar roles and perform the same functions.Single agents RL algorithms such as PPO (Proximal Policy Optimization) and ACKTR (Actor-Critic using Kronecker-Factored Trust Region) are extended to MARL with parameter sharing as MAPPO and MAACKTR. Lets learn how these algorithms work.

1. Proximal Policy Optimization [6]: PPO is a policy-gradient algorithm that learns a policy $\pi_\theta(a \mid s)$ which maps states (s) to actions (a) and the policy is updated to maximize the cumulative reward.It makes the algorithm more stable by preventing too many changes in a single update.It uses a clipped objective function to control how much a policy can change.

The final objective for Proximal Policy Optimization (PPO) is to maximize the clipped surrogate objective:

$$L^{CLIP}(\theta) = \mathbb{E}_t \left[\min \left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\varepsilon, 1+\varepsilon)\hat{A}_t \right) \right] \tag{9}$$

- $\mathbb{E}_t$: The expectation over time steps t.
- $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\text{old}}(a_t|s_t)}$: The importance of sampling ratio, comparing the probability of action a_t under the new policy π_θ and the old policy π_{old}.
- $\hat{A}_t$: The advantage function, which measures how much better an action a_t is compared to the average action in state s_t.
- $\text{clip}(r_t(\theta), 1-\varepsilon, 1+\varepsilon)$: A clipping function that restricts $r_t(\theta)$ to the range $[1-\varepsilon, 1+\varepsilon]$, where ε is a small hyper parameter (e.g., $\varepsilon = 0.2$).
- min: The minimum function ensures that the objective does not increase beyond the clipped range, thus preventing large policy updates.

If $r_t(\theta) \in [1-\varepsilon, 1+\varepsilon]$, the objective uses $r_t(\theta)\hat{A}_t$. If $r_t(\theta)$ goes outside this range, the clipped value $clip(r_t(\theta), 1-\varepsilon, 1+\varepsilon)\hat{A}_t$ is used to suppress large updates.

MAPPO is extended to multi agent environments by using parameter sharing. Sharing the parameters allows MAPPO to handle varying number of agents without requiring changes to the architecture or training process.The core equation of MAPPO is the clipped objective function, but the represents shared parameters,meaning the policy and value functions are being shared across all agents. The value function is being updated to minimize the loss. MAPPO ensures sufficient multi-agent learning by sharing both the policy and the value function.

$$\mathscr{L}_{\text{value}}(\phi) = \mathbb{E}_t \left[\left(V_\phi(s_t) - G_t \right)^2 \right] \tag{10}$$

Where:

- $V_\phi(s_t)$: The predicted value of state s_t using shared value function parameters ϕ.

- G_t: Discounted return from time step t.

2. Actor-Critic using Kronecker-Factored Trust Region [7]: ACKTR is a method for training that uses an advanced optimization technique to learn faster and more precisely. It has an actor-critic system where the actor decides what action to take based on the current state and the critic evaluates how good the action is and provides feedback.

- Actor: Learns the policy $\pi(a \mid s)$, which defines the probability of taking an action a given the state s
- Critic: Learns the value function $V(s)$, which estimates how good it is to be in a certain state s.

ACKTR improves its learning process by using natural gradients, which are smarter when compared to regular or first-order gradients. While the first-order gradients treat all parameters equally and do not consider how each parameter affects the environment, natural gradients take the geometry of the environment using the Fisher information matrix. Natural gradients adjust the size and direction of improvement, treating all the parameters usually. The FIM measures how sensitive the outputs of the model are to changes in its parameters, but calculating it exactly is computationally expensive for large models. K-FAC makes it easier through the calculation of the Fisher matrix by assuming the Kronecker product of two smaller matrices.

$$\nabla_\theta^{\text{natural}} J(\theta) = F^{-1} \nabla_\theta J(\theta) \tag{11}$$

$$F \approx A \otimes G \tag{12}$$

$$\nabla_\theta^{\text{natural}} J(\theta) \approx (A^{-1} \otimes G^{-1}) \nabla_\theta J(\theta) \tag{13}$$

- $\nabla_\theta J(\theta)$: The standard gradient of the objective function $J(\theta)$ with respect to the parameters θ. It represents the steepest direction for improving the policy.
- F: The Fisher Information Matrix (FIM), which measures how sensitive the policy's output is to changes in the parameters. It captures the geometry of the parameter space.
- F^{-1}: The inverse of the Fisher Information Matrix. It scales the gradient to adjust for parameter sensitivity, ensuring efficient updates in different directions.
- $\nabla_\theta^{\text{natural}} J(\theta)$: The natural gradient, which is the scaled version of the standard gradient. It accounts for the geometry of the parameter space and provides more efficient updates.
- A: The covariance matrix of activations, calculated as $A = \mathbb{E}[aa^T]$, where a represents the activations.
- G: The covariance matrix of gradients, calculated as $G = \mathbb{E}[gg^T]$, where g represents the gradients of the loss with respect to the layer's outputs.
- $\otimes$: Kronecker product, a matrix operation that combines two matrices into a larger block matrix.

3. Multi-Agent Advantage Actor-Critic: The Parameter-Sharing Advantage Actor-Critic (MA2C) algorithm is a multi-agent reinforcement learning (MARL) approach that extends the traditional Actor-Critic (A2C) framework for cooperative multi-agent environments. In MA2C, all agents share the same policy and value networks (parameter sharing), which reduces computational overhead and ensures consistency among agents. Each agent uses its local observations and receives localized rewards, enabling decentralized decision-making. This makes MA2C scalable for environments with many agents, such as fleet management, where vehicles (agents) need to allocate resources efficiently. In the fleet management problem, MA2C optimizes fleet performance by maximizing efficiency (e.g., reducing wait times or fuel consumption) through cooperative behavior. Parameter sharing allows the algorithm to scale without requiring individual networks for each agent. Experiments confirm MA2C's strong performance due to its efficiency, scalability, and ability to handle decentralized tasks. In the paper, MA2C is used also as a benchmark algorithm to compare against the proposed method in the on-ramp merging problem.

The paper introduces a **novel on-policy Multi-Agent Reinforcement Learning (MARL)** algorithm tailored for the challenging *on-ramp merging problem* in autonomous driving. This algorithm is designed to address gaps in efficiency and safety by incorporating the following key features:

- **Action Masking:** Prevents invalid or unsafe actions (e.g., lane changes to non-existent lanes or exceeding speed limits), improving stability and reducing training noise.
- **Priority-Based Safety Supervisor:** Ensures safety-critical decisions by regulating actions based on traffic scenarios and system constraints.
- **Parameter Sharing:** All agents share a common policy network, which enhances scalability, ensures consistency and reduces computational costs.
- **Local Reward Shaping:** Rewards are shaped to reflect local interactions (e.g., with neighboring vehicles), enabling efficient credit assignment and reducing communication overhead.

The paper compares the performance of this proposed algorithm against **benchmarks like MA2C** (from Section III) in Section V, showcasing its improvements in safety, efficiency, and overall merging behavior. This novel approach combines scalability, safety, and real-time decision-making for autonomous vehicle coordination.

III. RAMP MERGING AS MARL

The on-ramp merging problem is formulated as a partially observable Markov decision process (POMDP) in this section. To solve the defined POMDP, the actor-critic-based MARL algorithm is presented, including action masking, effective reward function design, and parameter-sharing mechanisms.

III.1. Problem Formulation

This study models the on-ramp merging scenario in mixed traffic as a model-free multi-agent network. It is assumed that the graph $G = (V, E)$ represents the network, where each agent $i \in V$ interacts with its neighbors $N_i := \{j \mid e_{ij} \in E\}$ through edge connections e_{ij}. Let $S := \{x_i \mid x_i \in S\}$ and $A := \{x_i \mid x_i \in A\}$ signify the global state space and action space, respectively. The system's underlying dynamics can be described using the state transition distribution $P : S \times A \times S \to [0, 1]$. A decentralized MARL framework is adopted, where each agent i observes only a portion of the environment, such as surrounding vehicles. This design aligns with real-world traffic dynamics, where AVs can only perceive or interact with vehicles in their immediate vicinity. As a result, the overall system is effectively modeled as a POMDP M_G, which is defined by the tuple $(\{A_i, S_i, R_i\}_{i \subseteq V}, T)$.

• **Action Space** $Based on the designs described in the paper^{[9]}, an agent i$ has a defined action space A_i that consists of high-level control instructions like turn left, turn right, cruise, acceleration, and deceleration. Lower-level controllers produce the necessary steering and throttle signals needed to navigate autonomous vehicles (AVs) once a high-level action is chosen. Figure (to be inserted) shows the general system configuration comprising these hierarchical control systems. The joint action space of all the AVs captures the overall action space for the system, implying,

$$A = A_1 \times A_2 \times \cdots \times A_N. \tag{14}$$

• **State Space** $The agent i$ is described as a matrix with dimensions $N_{N_i} \times W$ where N_{N_i} is the number of cars observed by the agent and W defines the number of attributes utilized to characterize every observed vehicle. These attributes comprise:

- ispresent: A binary variable denoting whether a vehicle is observable close to the ego vehicle.
- x: The observed vehicle's longitudinal relative location to the ego vehicle.
- y: The observed vehicle's lateral relative location to the ego vehicle.
- v_x: The observed vehicle's longitudinal velocity relative to the ego vehicle.
- v_y: The observed vehicle's lateral velocity relative to the ego vehicle.

Based on the agent's local sensing capabilities, it is expected that the ego vehicle can only observe "neighboring vehicles", defined as the N_{N_i} nearest cars located within a longitudinal distance of 150 meters from the ego vehicle. Setting $N_{N_i} = 5$ produced the highest performance in the case of the on-ramp merging situation shown in Fig. 1. The global state of the system is expressed as the Cartesian product of the states of every individual agent, obtained from:

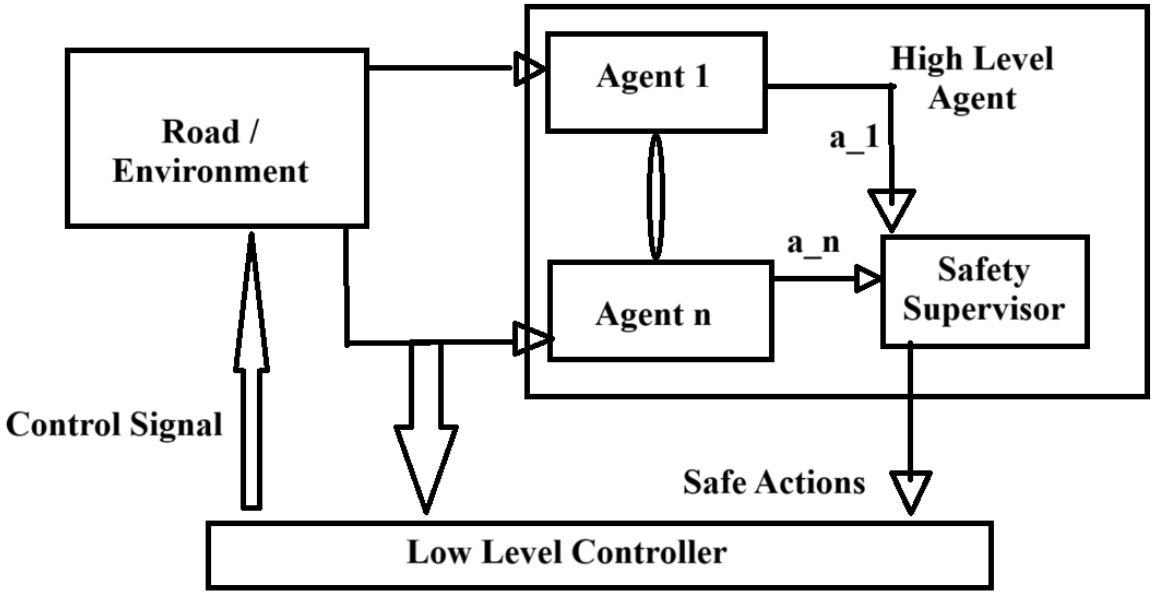

Figure 1. Flow chart of system and simulation setup

$$S = S_1 \times S_2 \times \cdots \times S_N. \tag{15}$$

- **Reward Function**

Training reinforcement learning agents to show the intended behavior depends critically on the reward function R_i. The aim is to let agents negotiate the merging space effectively and safely. Agent i gets the reward at time t from:

$$r_{i,t} = w_c r_c + w_s r_s + w_h r_h + w_m r_m, \tag{16}$$

where w_c, w_s, w_h, and w_m are positive weighting factors assigned to the collision penalty (r_c), speed evaluation (r_s), headway time evaluation (r_h), and merging cost evaluation (r_m), respectively. Safety is the priority here, hence w_c gets a far higher value than the other weights. The reward components are defined as follows:

- Collision evaluation (r_c): Assigned a value of -1 in case of a collision and 0 otherwise.
- Speed evaluation (r_s): Defined as:

$$r_s = \min\left\{\frac{v_t - v_{\min}}{v_{\max} - v_{\min}}, 1\right\}, \tag{17}$$

  where v_t is the current speed of the ego vehicle. Following recommendations from the US Department of Transportation (20–30 m/s) and the speed range observed in the Next Generation Simulation (NGSIM) dataset (minimum speed 6–8 m/s), the minimum and maximum speeds are set as $v_{\min} = 10\,\text{m/s}$ and $v_{\max} = 30\,\text{m/s}$, respectively.

- Headway time evaluation (r_h): Calculated as:

$$r_h = \log \frac{d_{\text{headway}}}{t_h v_t}, \tag{18}$$

where d_{headway} is the distance between the current car and the vehicle in front, t_h is a preset safe headway period. The ego vehicle is penalized if t_h is violated, that is, if the headway time is too short, while else it is rewarded. This work sets t_h as advised in [10].

- Merging cost (r_m): This component is designed to penalize excessive waiting time on the merging lane, which can lead to traffic deadlocks. The merging cost is defined as:

$$r_m = -\exp\left(-\frac{(x-L)^2}{10L}\right), \tag{19}$$

where x represents the distance traveled by the ego vehicle on the ramp, and L denotes the total length of the ramp (refer to Fig. 1).

As shown in Fig. 3, the penalty increases as the ego vehicle gets closer to the merging endpoint. This formulation encourages efficient use of the ramp by minimizing delays and avoiding congestion.

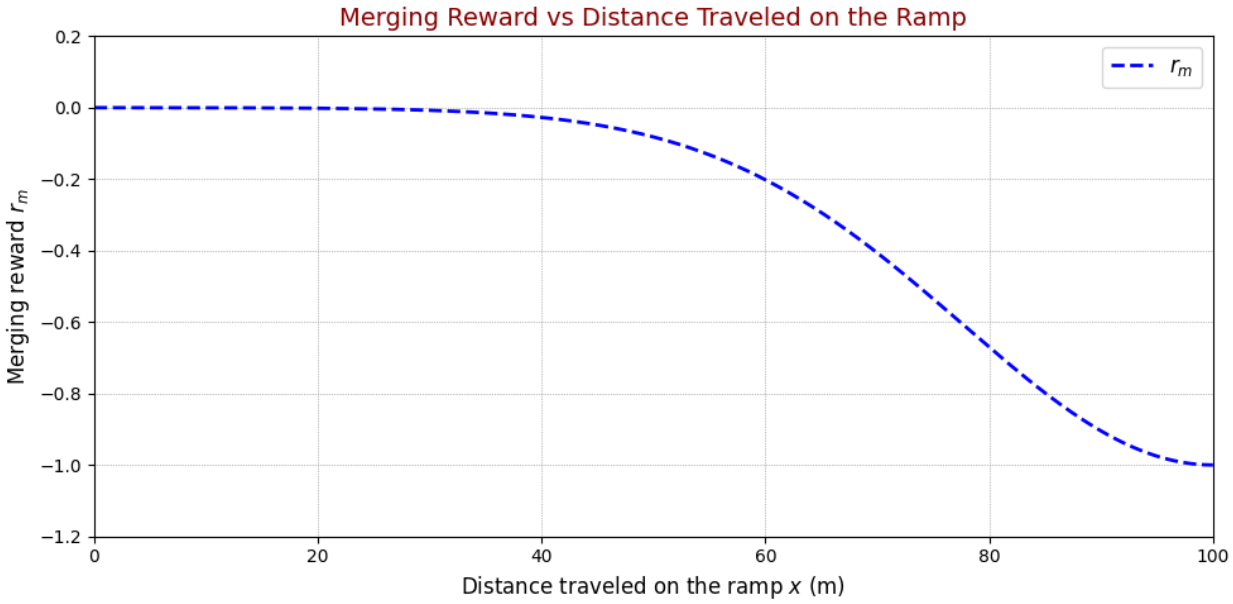

Figure 2. Variation of merging reward with distance x

III.2. MA2C for CAVs

Maximizing the worldwide reward $R_{g,t}$ defined as the total of individual agent rewards is the main goal of the cooperative MARL framework:

$$R_{g,t} = \sum_{i=1}^{N} r_{i,t}. \tag{20}$$

Under ideal circumstances, every agent would get the same average worldwide reward computed as:

$$r_{i,t} = \frac{1}{N} R_{g,t}, \quad \text{for } i = 1, 2, \ldots, N. \tag{21}$$

This common reward system does, however, have major drawbacks. First, aggregating the global reward brings significant latency and increases communication overhead, which might be troublesome in systems with real-time restrictions, including autonomous vehicles (AVs). Second, the credit assignment issue results from a single global reward failing to represent the unique efforts of every vehicle adequately. This can restrict scalability to a small number of agents and compromise learning efficiency.

A localized incentive assignment method is proposed to address these issues. Under this approach, the reward of every ego vehicle is found only based on those of its surrounding neighbors. The agent i gets reward at time t as follows:

$$r_{i,t} = \frac{1}{|v_i|} \sum_{j \in v_i} r_{j,t}, \tag{22}$$

where $v_i = i \cup N_i$ represents the set of the ego vehicle i and its neighboring vehicles, and $|v_i|$ is the size of this set.

This localized reward system guarantees that agents concentrate mostly on interactions with nearby cars, therefore assigning rewards more relevant to the success or failure of a task. For tasks involving road-based interactions, this approach is especially successful since vehicles mostly affect their immediate surroundings.

The network backbone used in this study is depicted in Fig. 4, where the actor and critic networks share the same low-level representations. The policy loss and value function error are integrated into a single overall loss function, utilizing shared network parameters. The overall loss function is defined as:

$$J(\theta_i) = J^{\pi_{\theta_i}} - \beta_1 J^{V_{\phi_i}} + \beta_2 H(\pi_{\theta_i}(s_t)), \tag{23}$$

where β_1 and β_2 are weighting coefficients for the value function loss and the entropy regularization term, respectively. The entropy regularization term is expressed as:

$$H(\pi_{\theta_i}(s_t)) = \mathbb{E}_{\pi_{\theta_i}}[-\log(\pi_{\theta_i}(s_t))], \tag{24}$$

which encourages agents to explore new states, thereby improving learning efficiency.

The policy loss, $J^{\pi_{\theta_i}}$, is given by:

$$J^{\pi_{\theta_i}} = \mathbb{E}_{\pi_{\theta_i}}\left[\log(\pi_{\theta_i}(a_{t,i} \mid s_{t,i}))A_{t,i}^{\pi_{\theta_i}}\right], \tag{25}$$

where $A_{t,i}^{\pi_{\theta_i}}$ represents the advantage function, defined as:

$$A_{t,i}^{\pi_{\theta_i}} = r_{t,i} + \gamma V_{\phi_i}(s_{t+1,i}) - V_{\phi_i}(s_{t,i}). \tag{26}$$

The loss for updating the state value function, $J^{V_{\phi_i}}$, is:

$$J^{V_{\phi_i}} = \min_{\phi_i} \mathbb{E}_t\left[\left(r_{t,i} + \gamma V_{\phi_i}(s_{t+1,i}) - V_{\phi_i}(s_{t,i})\right)^2\right]. \tag{27}$$

Separate experience replay buffers are maintained for each agent, while the shared policy network parameters are updated collectively across agents. This approach is especially effective in cooperative multi-agent reinforcement learning scenarios involving mixed traffic. Mini batches of sampled trajectories are employed to iteratively update the network parameters using Equation (23), ensuring reduced variance and enhanced training stability.

III.3. DNN Settings

Fig. 4 shows the deep neural network (DNN) architectural layout. To enhance scalability and robustness, the observation $s_{i,t}$ is divided into three sub-groups based on their physical characteristics: $s_{i,t}^1 \cup s_{i,t}^2 \cup s_{i,t}^3$ where $s_{i,t}^1$, $s_{i,t}^2$, and $s_{i,t}^3$ correspond to *ispresent states*, *position states*, and *speed states*, respectively. Every sub-group passes an independent fully connected (FC) layer, and the resultant encoded representations are concatenated into a single vector. Actor and critic networks share a 128-neuron FC layer derived from this concatenated vector. In the standard setting, the actor-network produces logits l_i that are passed through a Softmax layer to generate action probabilities: $\pi_{\theta_i}(s_i) = \text{softmax}([l_1, l_2, l_3, l_4, l_5])$ where $\pi_{\theta_i}(s_i)$ represents the probability distribution over actions. Next, actions from this distribution—that is , $a_i \sim \pi_{\theta_i}(s_i)$ are sampled. However, this sampling process introduces some challenges:

- Invalid/unsafe actions: Actions judged hazardous or invalid still carry non-zero probability. These dangerous behaviors could be chosen during training depending on the stochastic character of sampling, therefore causing possibly negative behaviors or system instability.
- Policy updates with invalid actions: Execution of undesirable actions affects and impedes policy training since the gathered experience is useless and misleading.

The proposed invalid action masking technique in [8] is used to help reduce these problems. This method "masks out" illegal behavior, therefore guaranteeing that only legitimate actions are gathered. This erroneous action mask, which is included in the actor network for strong policy training, is shown in Figure 3.

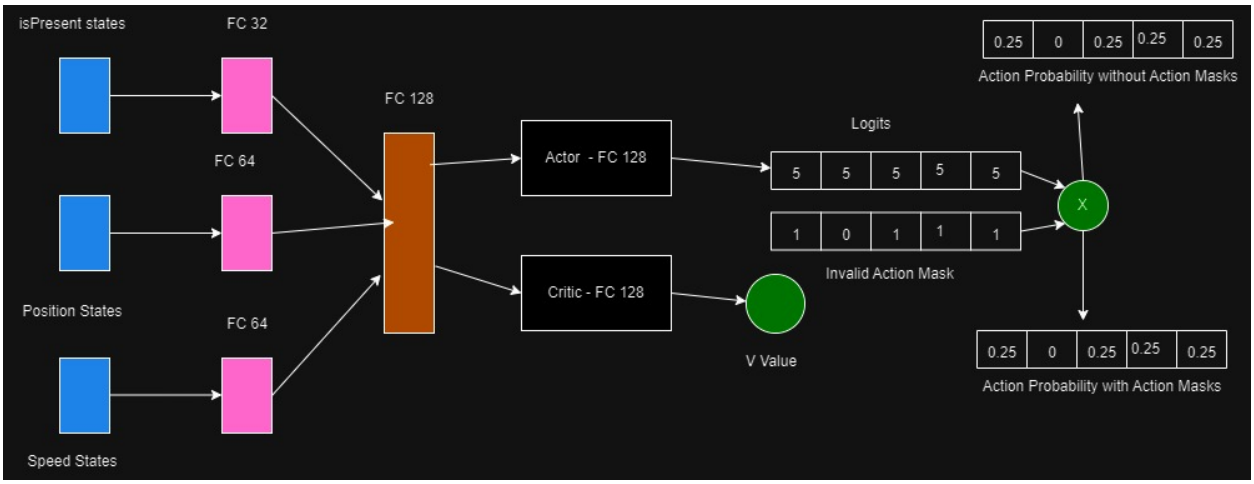

Figure 3. Schematic representation of the proposed DNN architecture

IV. PRIORITY-BASED SAFETY ENHANCEMENT

The rule-based action masking system outlined earlier is effective at preventing improper actions, such as invalid lane changes or unsafe acceleration. However, it does not fully address the risks of inter-vehicle or vehicle-obstacle collisions. To manage these risks in complex, dynamic, and congested mixed-traffic environments, a more comprehensive safety-enhancing approach is proposed. This system leverages vehicle dynamics and multi-step predictions over a defined *prediction horizon (T_n)* to ensure collision-free behavior. The primary objective of the safety supervisor is to predict potential collisions within the prediction horizon and adjust unsafe actions proactively. This involves modeling the behavior of both Human-Driven Vehicles (HDVs) and Autonomous Vehicles (AVs) to ensure safe and efficient navigation. **1. Human-Driven Vehicles (HDVs)** For HDVs, the safety supervisor uses the following models to predict behavior:

- **Intelligent Driver Model (IDM):** This model estimates longitudinal acceleration (a_i) of HDVs based on their current speed (v_i) and headway distance ($d_{headway}$) to the vehicle ahead:

$$a_i = a_{max}\left(1 - \left(\frac{v_i}{v_{des}}\right)^4 - \left(\frac{s^*(v_i,\Delta v)}{d_{headway}}\right)^2\right), \tag{28}$$

  where $s^*(v_i,\Delta v)$ represents the desired safe distance, which depends on speed, time headway (T), and braking deceleration:

$$s^*(v_i,\Delta v) = s_0 + v_i T + \frac{v_i \Delta v}{2\sqrt{a_{max} b}}. \tag{29}$$

- **MOBIL Lane-Change Model:** This model predicts when lane changes are safe and beneficial by evaluating the acceleration gain (a_{gain}) associated with the maneuver:

$$a_{gain} > a_{threshold}. \tag{30}$$

  Lane changes occur only if they provide a significant benefit and are safe for all vehicles involved.

2. Autonomous Vehicles (AVs) For AVs, high-level decisions such as acceleration or lane changes are determined by a *Multi-Agent Reinforcement Learning (MARL) agent*. These high-level actions are then implemented through *low-level PID controllers*, which generate precise control signals. The PID controller computes these signals as:

$$u(t) = K_p e(t) + K_i \int e(t)\,dt + K_d \frac{de(t)}{dt}, \tag{31}$$

where $e(t)$ is the error between the desired and actual state, and K_p, K_i, and K_d are proportional, integral, and derivative gains, respectively. This ensures that the AVs execute decisions smoothly and reliably.

IV. 1. Trajectory Propagation and Motion Primitives

Vehicle trajectories are propagated using the *kinematic bicycle model*, which simulates vehicle motion based on velocity (v) and steering angle (δ):

$$\dot{x} = v\cos(\theta), \quad \dot{y} = v\sin(\theta), \quad \dot{\theta} = \frac{v}{L}\tan(\delta), \tag{32}$$

where (x, y) is the vehicle's position, θ is the orientation, L is the wheelbase, and δ is the steering angle. These predicted trajectories, referred to as *motion primitives*, allow the supervisor to evaluate future paths and ensure collision-free navigation.

The safety supervisor integrates predictive models for both HDVs and AVs and uses motion primitives to evaluate potential risks within the prediction horizon. The framework provides:

- Proactive collision avoidance by simulating future vehicle trajectories.
- Accurate modeling of HDV behavior using IDM and MOBIL.
- Efficient decision-making for AVs using MARL and PID controllers.

The proposed framework and simulation setup are illustrated in **Fig.1** . This system ensures safe, efficient, and reliable operation in high-density, mixed-traffic environments by effectively addressing both safety and efficiency.

IV. 2. Priority Assignment

Based on the joint motion primitives of all AVs, it is possible to forecast if a collision might arise inside the next T_n using the motion models for HDVs. This framework designs the safety-enhancing mechanism using the cooperative efforts of all AVs. While determining a joint safe action to prevent collisions is computationally difficult, spotting possible collisions from a given joint activity is rather easy. This results from the size of the action space, $|A_i|^N$ where N is the overall count of AVs. Particularly in cases with several AVs, the size of the action space expands exponentially as N increases, rapidly rendering the job computationally intractable. To address time constraints effectively, we propose a sequential, priority-based safety enhancement scheme that balances computational efficiency with real-time applicability. The core idea is to coordinate AVs in a sequential order, giving higher priority to AVs with smaller safety margins. For instance, AVs near the end of the merging lane or close to the defined safety boundary (e.g., with small time headway) are assigned higher priorities. The following rationales are considered for assigning priorities:

1. Vehicles on the merging lane are given higher priority compared to vehicles on the through lane, as merging is a time-critical process, especially near the lane end.
2. Vehicles closer to the merging lane's end are prioritized higher because they are more likely to encounter collisions or cause deadlocks [11].
3. Vehicles with shorter time headway are assigned higher priority since they are at greater risk of colliding with preceding vehicles.

Based on these principles, the priority index p_i for the ego vehicle i is defined as:

$$p_i = \alpha_1 p_m + \alpha_2 p_d + \alpha_3 p_h + \sigma_i, \tag{33}$$

where α_1, α_2, and α_3 are positive weighting factors for the merging priority metric (p_m), the distance-to-end metric (p_d), and the time headway metric (p_h), respectively. The term $\sigma_i \sim \mathcal{N}(0, 0.01)$ is a small random variable added to avoid ties between two vehicles with the same priority index.

The metrics are defined as follows: **Merging priority metric (p_m):**

$$p_m = \begin{cases} 0.5, & \text{if on the merging lane,} \\ 0, & \text{otherwise.} \end{cases} \tag{34}$$

Distance-to-end priority metric (p_d):

$$p_d = \begin{cases} \frac{x}{L}, & \text{if on the merging lane,} \\ 0, & \text{otherwise.} \end{cases} \tag{35}$$

Here, x represents the distance the ego vehicle has traveled on the merging lane, and L is the total length of the ramp.

Time headway priority metric (p_h): The time headway priority is computed using the time headway definition:

$$p_h = -\log\left(\frac{d_{\text{headway}}}{t_h v_t}\right), \tag{36}$$

where d_{headway} is the headway distance, t_h is the predefined headway time, and v_t is the ego vehicle's speed.

IV. 3. Priority-based Safety Supervisor

The Priority-Based Safety Supervisor is an advanced system designed to ensure the safe operation of Autonomous Vehicles (AVs) in dynamic, mixed-traffic environments. It works by analyzing predicted vehicle motions, assigning priorities to AVs based on safety-critical factors, and adjusting their actions to prevent collisions. The supervisor processes vehicles sequentially, starting with those in the most critical situations, and ensures both computational efficiency and real-time applicability.

At each time step t, the supervisor generates a priority list P_t for all AVs, ranked by their priority scores. These scores are determined using metrics such as proximity to the merging lane endpoint and time headway to the vehicle ahead. The AV with the highest priority ($P_t[0]$) is selected first for a safety check. This involves analyzing its exploratory action, which is the proposed action generated by its action network, to determine if it could lead to a collision.

Collision prediction is based on the motions of both AVs and Human-Driven Vehicles (HDVs). The motions of HDVs are estimated using the *Intelligent Driver Model (IDM)* for acceleration and the *MOBIL model* for lane changes. For lower-priority AVs, their motions are assumed to remain the same as in the last time step. Using these predictions, the supervisor checks if the trajectories of any two vehicles come within a safety threshold over a prediction horizon (T_n). If no collision is detected, the exploratory action of $P_t[0]$ is deemed safe and approved.

If a collision is predicted, the exploratory action is marked as unsafe. The supervisor then evaluates alternative actions from the set of valid actions (A_{valid}) for the AV. Each action is assessed based on its safety margin ($d_{\text{sm},k}$), which is the minimum safe distance to surrounding vehicles. For lane-changing actions, the safety margin is calculated as the minimum distance to preceding and following vehicles in both the current and target lanes. For speed adjustments, the safety margin is the minimum headway distance to the vehicle ahead. The action with the highest safety margin is selected as the safest alternative, using the formula:

$$a_t' = \arg\max_{a_t \in A_{\text{valid}}} \left(\min_{k \in T_n} d_{\text{sm},k}\right) \tag{37}$$

After selecting a safe action, the trajectory of $P_t[0]$ is updated, and the vehicle is removed from the priority list. The process then moves to the next-highest-priority vehicle ($P_t[1]$), and the same safety-check procedure is repeated. For lower-priority AVs, the updated trajectories of higher-priority AVs are considered during the safety checks. This process continues until all vehicles in the priority list have been processed.

The system is designed to ensure collision-free motion by addressing unsafe exploratory actions and selecting the safest alternatives dynamically. It balances safety and efficiency by processing vehicles in order of priority and focusing on the most critical scenarios. For instance, in the case of lane changes, the supervisor evaluates the minimum distance to vehicles in both the current and target lanes, while for speed adjustments, it considers the headway distance to the vehicle ahead.

By employing this structured approach, the Priority-Based Safety Supervisor ensures that AVs can make safe and cooperative decisions in real-world traffic scenarios. It proactively detects and mitigates potential collisions, enhances computational efficiency, and enables real-time decision-making in complex environments, making it a robust solution for traffic safety management.

This algorithm ensures the safe operation of autonomous vehicles (AVs) in dynamic and complex traffic scenarios. It prioritizes vehicles based on their risk levels and iteratively ensures their actions are safe, modifying unsafe actions when necessary. The steps of the algorithm are as follows:

IV. 4. Algorithm 1 - Priority-Based Safety Supervisor

Algorithm 1 Priority-Based Safety Supervisor

Parameter : L, α_1, α_2, α_3, t_h, w, T_n
Output : a_i, $i \in \nu$

for $i = 0$ ***to*** N **do**
  Compute the priority scores according to Eq. (10) Rearrange ego vehicles to list P_t according to their priority scores
for $j = 0$ ***to*** $|P_t|$ **do**
  Obtain the highest-priority vehicle $P_t[0]$ Find its neighboring vehicles $N_{P_t[0]}$ Predict trajectories ζ_v, $v \in P_t[0] \cup N_{P_t[0]}$ for T_n time steps **if** *trajectories are overlapped* **then**
    Replace the risky action as $a_t \leftarrow a_t'$ according to Eq. (13) Replace the trajectory $\zeta_{P_t[0]}$ with $\zeta'_{P_t[0]}$
  Remove $P_t[0]$ from P_t Update $P_t[i] \leftarrow P_t[i+1]$, $i = 1, 2, \ldots$

IV.4. 1 Input Parameters and Output

1. Parameters:

- L: Total length of the merging ramp.
- $\alpha_1, \alpha_2, \alpha_3$: Weighting factors for merging priority, distance-to-end priority, and time headway priority, respectively.
- t_h: Time headway threshold.
- w: Safety margin parameter.
- T_n: Time horizon for collision prediction.

2. Output:

- Safe actions a_i for each AV i in the system.

IV. 4. 2. Algorithm Steps

Step 1: Compute Priority Scores: For each vehicle i, calculate the priority score p_i using the formula:

$$p_i = \alpha_1 p_m + \alpha_2 p_d + \alpha_3 p_h + \sigma_i \tag{38}$$

- p_m: Merging priority.
- p_d: Distance-to-end priority.
- p_h: Time headway priority.
- σ_i: Small random value to avoid ties.

Step 2: Create a Priority List: Arrange vehicles in a priority list P_t in descending order of their priority scores. The highest-priority vehicle is placed at the top.

Step 3: Process Each Vehicle in Priority Order:

(a) Select the highest-priority vehicle $P_t[0]$.
(b) Identify its neighboring vehicles $N_{P_t[0]}$, which include other AVs and human-driven vehicles (HDVs).

Step 4: Predict Future Trajectories: Predict the trajectories ζ_v for T_n steps for the selected vehicle $P_t[0]$ and its neighboring vehicles $N_{P_t[0]}$.

Step 5: Check for Potential Collisions:

- If no trajectory overlap is detected, the exploratory action a_t is deemed safe and selected.
- If a trajectory overlap is detected, replace the exploratory action a_t with a safe action a_t' determined by:

$$a_t' = \arg \max_{a_t \in A_{\text{valid}}} \left(\min_{k \in T_n} d_{sm,k} \right) \tag{39}$$

- A_{valid}: Set of valid actions.
- $d_{sm,k}$: Safety margin at step k.

Step 6: Update Trajectory: Update the trajectory $\zeta_{P_t[0]}$ with the new safe trajectory $\zeta'_{P_t[0]}$.

Step 7: Remove Processed Vehicle:

- Remove $P_t[0]$ from the priority list.
- Update the priority list by moving the next highest-priority vehicle to the top.

Step 8: Repeat Until All Vehicles are Processed: Continue steps 3 to 7 until the priority list is empty.

Remark 1: Implementation with Vehicle-to-Infrastructure (V2I) Communication
The priority-based safety supervision scheme utilizes *Vehicle-to-Infrastructure (V2I)* communication to coordinate Autonomous Vehicles (AVs) and Human-Driven Vehicles (HDVs) near merging ramps. A central communication station monitors traffic conditions and calculates priority scores for AVs based on real-time observations. At each discrete time step t, the priority score for an AV i is computed as:

$$p_i = \alpha_1 p_m + \alpha_2 p_d + \alpha_3 p_h + \sigma_i, \tag{40}$$

p_m: Merging priority metric, defined as:

$$p_m = \begin{cases} 0.5, & \text{if on the merging lane,} \\ 0, & \text{otherwise.} \end{cases} \tag{41}$$

p_d: Distance-to-end metric, calculated as:

$$p_d = \begin{cases} \frac{x}{L}, & \text{if on the merging lane,} \\ 0, & \text{otherwise.} \end{cases} \tag{42}$$

Here, x is the distance traveled on the merging lane, and L is the total length of the ramp.

p_h: Time headway priority metric, computed as:

$$p_h = -\log\left(\frac{d_{\text{headway}}}{t_h v_t}\right), \tag{43}$$

- d_{headway}: Headway distance to the vehicle ahead.
- t_h: Predefined safe time headway.
- v_t: Vehicle speed.
- $\alpha_1, \alpha_2, \alpha_3$: Positive weighting factors that balance the impact of p_m, p_d, and p_h, respectively.
- σ_i: A small random variable ($\sigma_i \sim \mathcal{N}(0, 0.01)$) to prevent ties between vehicles with identical priority scores.

Once the priority scores are computed:

- AVs send their exploratory actions (proposed movements) to the central station.
- The station applies Algorithm 1 to process AVs sequentially based on their priority scores and generate safe actions.

The sequential nature of Algorithm 1 ensures computational efficiency. For instance, with a *prediction horizon* $T_n = 8$, the system processes actions in approximately 28.13 milliseconds per decision in complex traffic conditions (refer to Table II in Section V-C). This computational speed makes real-time implementation feasible, enabling updated control commands within one sampling step. Future optimizations will further enhance computational efficiency.

Remark 2: Importance of the Prediction Horizon T_n

The prediction horizon T_n is a critical hyperparameter that governs how far into the future the safety supervisor predicts potential collisions. The effects of small and large T_n are as follows:

Impact of Small T_n

- A small T_n makes the safety supervisor "short-sighted," potentially failing to anticipate imminent collisions.
- This limitation could lead to no feasible solutions after a few steps.

Impact of Large T_n

- A large T_n accumulates uncertainties in human-driven vehicle (HDV) behavior due to noise and variations in actual vehicle motion.
- This results in overly conservative decisions, sacrificing performance to guarantee safety over extended horizons.

Optimal Value of T_n

To strike a balance, cross-validation experiments indicate that $T_n = 8$ or $T_n = 9$ provides the optimal trade-off. This value ensures:

- Adequate foresight to predict and resolve potential collisions.
- Computational efficiency for real-time decision-making.

IV. 5 . Algorithm 2 - MARL for AVs with Safety Supervisor

- **Parameters:** γ, η, T, M, β_1, β_2.

- **Outputs:** θ.

The algorithm begins by initializing the necessary parameters, such as the initial state of the system, the current time step, and an empty replay buffer to store experiences. The training process is divided into multiple epochs, each consisting of a fixed number of time steps. For each time step within an epoch, every agent (vehicle) collects information about its current state and determines its next action using a policy network enhanced with action masking to prevent invalid moves. These exploratory actions are then sent to the priority-based safety supervisor, which checks if they are safe. If an action is deemed unsafe, the safety supervisor replaces it with a predefined safer alternative based on a specific rule. The agent then executes the safe action, and the experience, including the state, action, reward, and resulting state, is stored in the replay buffer.

This process continues until the end of the time step sequence for the epoch. Afterward, the algorithm evaluates whether the episode is complete, either because the task has been finished successfully or because a collision or unsafe event occurred (signaled by the "DONE" flag). Once the episode is complete, the policy network of each agent is updated using the collected experiences stored in the buffer. The agents then reset to their initial states, and the next epoch begins. This cycle of exploration, safety validation, action execution, experience storage, and learning repeats for the specified number of epochs, allowing the agents to iteratively refine their policies and improve their performance over time.

The combination of action masking, safety supervision, and reinforcement learning ensures that the agents can explore the environment safely while progressively learning better strategies. The safety supervisor guarantees that risky actions are intercepted and replaced with safe ones, preventing accidents while still enabling the agents to learn from their experiences. This structured approach balances exploration and exploitation while maintaining the safety of the learning process.

Algorithm 2 MARL for AVs with Safety Supervisor

Initialize s_0, $t \leftarrow 0$, $D \leftarrow \emptyset$;
for $j = 0$ to $M-1$ **do**
  for $t = 0$ to $T-1$ **do**
    for $i \in \nu$ **do**
      Observe s_i;
      Update $a_{i,t} \sim \pi_{\theta_i}(\cdot|s_i)$ with action masking;
    end for
    for $i \in \nu$ **do**
      Check the actions by Algorithm 1;
      if safe **then**
        Execute $a_{i,t}$;
        Update $D_i \leftarrow (s_{i,t}, a_{i,t}, r_{i,t}, v_{i,t})$;
      else
        Update $a_{i,t} \leftarrow a'_{i,t}$ and execute $a'_{i,t}$;
        Update $D_i \leftarrow (s_{i,t}, a'_{i,t}, r_{i,t}, v_{i,t})$;
      end if
    end for
    Update $t \leftarrow t+1$;
    if DONE **then**
      for $i \in \nu$ **do**
        Update $\theta_i \leftarrow \theta_i + \eta \nabla_{\theta_i} J(\theta_i)$;
      end for
    end if
    Initialize $D_i \leftarrow \emptyset$, $i \in \nu$;
  end for
  Update $j \leftarrow j+1$;
end for
Update s_0, $t \leftarrow 0$; =0

Step 1: Initialization: Initialize the initial state s_0. Set the time $t \leftarrow 0$. Initialize the replay buffer $D \leftarrow \emptyset$, which will store experiences for learning.

Step 2: Outer Loop Over Epochs: The algorithm runs for M epochs. Each epoch represents a complete training cycle.

Step 3: Inner Loop Over Time Steps: For each epoch, the algorithm progresses through T time steps. At each time step:
Step 3.1: Agent Observation and Action Selection

- For each agent $i \in \nu$, observe the current state s_i.
- Select an action $a_{i,t}$ using the policy $\pi_{\theta_i}(\cdot|s_i)$ with action masking.
- Action masking ensures that invalid or impractical actions are excluded from consideration.

Step 4: Safety Check: The priority-based safety supervisor checks the proposed actions $a_{i,t}$ for all agents:

- **If the action is safe:**
  - The agent executes $a_{i,t}$.
  - The experience tuple $(s_{i,t}, a_{i,t}, r_{i,t}, v_{i,t})$, which includes the state, action, reward, and next state, is stored in the replay buffer D_i.
- **If the action is unsafe:**
  - Replace the unsafe action $a_{i,t}$ with a safe action $a'_{i,t}$, which is executed instead.
  - Store the experience $(s_{i,t}, a'_{i,t}, r_{i,t}, v_{i,t})$ in D_i.

Step 5: Update Time Step: Increment the time t to proceed to the next step.

Step 6: Policy Update (When Episode Ends)

- Check if the episode has ended, which can occur if:
  - The task is completed, or
  - A collision happens (flagged by the *DONE* signal).
- If the episode ends, update the policy parameters θ_i of each agent using the collected experiences:

$$\theta_i \leftarrow \theta_i + \eta \nabla_{\theta_i} J(\theta_i),$$

  where $J(\theta_i)$ is the objective function and η is the learning rate.

Step 7: Reset Buffers: After the episode ends, clear the replay buffers D_i for all agents to prepare for the next episode.

Step 8: Epoch Update: Once all time steps in an epoch are completed, increment the epoch counter j and begin the next epoch.

Step 9: Final Reset: After all epochs are completed, reset the initial state s_0 and the time t to their default values.

V. NUMERICAL EXPERIMENTS

This section evaluates the performance of the proposed Multi-Agent Reinforcement Learning (MARL) algorithm in a simulated road merging scenario. The evaluation focuses on two key aspects: training efficiency (how quickly and effectively the algorithm learns) and collision rate (its ability to avoid accidents during the merging process).

The road used for the simulation is 520 meters long, with a merging lane that begins at 320 meters and stretches for 100 meters. There are 12 evenly spaced spawn points along the main lane and the ramp lane between 0 and 220 meters. Vehicles that move beyond the end of the road are removed from the display, but their motion dynamics continue to be updated in the simulation. The scenario is tested under three levels of traffic density: easy, medium, and hard. In easy mode, 1-3 autonomous vehicles (AVs) and 1-3 human-driven vehicles (HDVs) are introduced. Medium mode includes 2-4 AVs and 2-4 HDVs, while hard mode features 4-6 AVs and 3-5 HDVs.

Each training episode begins with a randomized number of AVs and HDVs, which are assigned starting positions with slight variations (noise of ±1.5 meters) around their spawn points. The vehicles' initial speeds are randomly selected between 25 and 27 meters per second. The AVs make decisions and take actions every 0.2 seconds, corresponding to a control frequency of 5 Hz. To simulate real-world conditions, HDVs' acceleration and steering inputs are perturbed with 5% random noise.

The MARL algorithm is trained over 2 million steps, corresponding to approximately 20,000 episodes, with each episode lasting up to 100 steps. Training is conducted with three different random seeds to ensure the robustness of the results, with the same seeds shared among all agents. The algorithm is evaluated every 200 training episodes by running three evaluation episodes, allowing periodic assessment of its performance and progress.

The reward function used for training is designed to encourage safe and efficient driving. Agents are rewarded based on their ability to avoid collisions (weight of 200), drive safely (weight of 1), maintain a safe distance from other vehicles (weight of 4), and merge successfully (weight of 4). The algorithm uses a discount factor (γ) of 0.99 to balance immediate and future rewards and a learning rate (η) of 5×10^{-4} to control the speed of updates to the agents' policies. Additional parameters, including priority coefficients ($\alpha_1, \alpha_2, \alpha_3$) set to 1 and loss function weights ($\beta_1 = 1$ and $\beta_2 = 0.01$), are used to balance learning priorities.

The simulation is built on a modified version of the highway-env simulator, an open-source tool designed for traffic scenarios. This simulator incorporates default settings from the Intelligent Driver Model (IDM) and MOBIL models, which handle vehicle dynamics and lane-changing behaviors. The experiments are conducted on an Ubuntu 18.04 server with an AMD 9820X processor and 64 GB of memory.

The proposed MARL algorithm is compared against a baseline version that does not include the safety supervisor introduced in this work. This comparison demonstrates the improvements in collision avoidance and overall performance brought by the safety supervisor. Additionally, a video demonstration of the training process is available online, providing a visual representation of the algorithm in action.

V. 1. Reward Function Designs

In this subsection, we evaluate the performance of the proposed Multi-Agent Reinforcement Learning (MARL) framework by analyzing two aspects: (1) the comparison between local and global reward designs, and (2) the impact of the safety penalty weight w_c in the reward function.

1. Local vs. Global Reward Designs

The proposed local reward design is compared to the global reward design used in previous works.

- **Local Reward:** Each agent receives a reward based on its own actions and outcomes.
- **Global Reward:** Each agent receives the average reward of all agents, defined as:

$$r_{i,t} = \frac{1}{N} \sum_{j=1}^{N} r_{j,t},$$

  where N is the total number of agents, and $r_{j,t}$ is the reward of agent j at time t.

The evaluation results (shown in Fig. 8) demonstrate that the proposed local reward design outperforms the global reward design in terms of:

- **Higher Rewards:** Agents achieve better performance with local rewards.
- **Faster Convergence:** The learning process is faster with local rewards.

This difference is particularly evident in **Hard mode**, where the global reward design fails, achieving evaluation rewards below zero. The poor performance of global rewards in challenging scenarios is due to:

1. **Credit Assignment Issues:** Global rewards make it difficult to link an agent's actions to the overall outcome as the number of agents increases.
2. **Lack of Correlation:** The averaged global rewards do not align well with the actions of individual agents, reducing their effectiveness in learning.

In contrast, local rewards allow agents to optimize their own behaviors, resulting in better outcomes across all traffic scenarios, especially in complex conditions.

2. Impact of Safety Penalty Weight w_c

The effect of the safety penalty weight w_c in the reward function (Eq. 14) is studied using different values of w_c in the **Medium traffic mode**, while keeping other coefficients unchanged. The reward function is given as:

$$r_i = w_c \cdot \text{safety metric} + w_s \cdot \text{smoothness metric} + w_h \cdot \text{headway metric} + w_m \cdot \text{merging success metric}.$$

Effect of w_c:

- **High w_c:** Encourages safer behavior by penalizing unsafe actions more heavily. For example:
  - When $w_c \geq 100$, there are no collisions, indicating a strong emphasis on safety.
  - As w_c increases further, agents become overly conservative, reducing the overall traffic speed.
- **Low w_c:** Prioritizes traffic efficiency over safety, which can lead to collisions.

The testing results (shown in Table I) confirm that selecting $w_c \geq 100$ eliminates collisions. However, very high values of w_c slow down traffic flow as agents become excessively cautious.

Optimal Trade-Off: To balance safety and efficiency, w_c is set to 200 in the following experiments. This value provides a reasonable trade-off, ensuring a safe margin while maintaining traffic efficiency.

These findings emphasize the importance of reward function design in ensuring both safety and efficiency in multi-agent traffic scenarios.

V. 2. Curriculum Learning for Hard Traffic Mode

Curriculum Learning has been adopted to improve both the training speed and inference accuracy in high-traffic density scenarios, often referred to as the "Hard" traffic mode. The way people pick up difficult tasks—beginning with simpler ideas and working gradually toward more difficult ones—inspired this learning paradigm. Curriculum learning in this context refers to fine-tuning pre-trained models from simpler traffic modes—that is, Easy and Medium—for more difficult conditions, including the Hard traffic mode.

Curriculum learning lets us build on a basis laid in easier environments rather than directly from scratch for the Hard mode. This procedure greatly improves training effectiveness since the model gains from the knowledge already learned in previous phases. The model adjusts more successfully to the complexity of high-density traffic by spreading acquired representations and actions.

Safety-critical uses like autonomous driving especially benefit from this method. Direct training in a complicated environment might result in many "blind" explorations, in which the agent acts untrained and maybe dangerously. Such investigations can cause training inefficiencies or crashes. Starting from a well-trained baseline model, curriculum learning reduces the frequency of dangerous activities and increases the general safety and stability of the training process, therefore mitigating these risks.

The outcomes clearly show how successful curricula are. By greatly accelerating the convergence rate, pre-trained models from simpler traffic modes help the model to learn optimal policies more rapidly. Better final performance results from this method as well as from beginning training from scratch.

Moreover, the benefits of curriculum learning help to deal with traffic efficiently. This development emphasizes the capacity of the method to increase not only the operational effectiveness of the trained model but also the safety and stability of training.

Curriculum learning is implemented in the next tests for the Hard traffic mode considering these major advantages. This method guarantees that, in complicated, high-density traffic conditions, the model keeps improving while reducing training risks and boosting efficiency.

V. 3. Performance of the Priority-based Safety Supervisor

We assess in this subsection the performance of the suggested priority-based safety supervisor and its effects on traffic flow, training efficiency, and assessment incentives. As shown, the suggested safety supervisor exhibits better sample efficiency by faster convergence over all three traffic densities. Moreover, even in demanding Hard traffic conditions, the safety supervisor gets better assessment incentives. This development results from its capacity to substitute safe activities for dangerous ones, especially in the early stages of discovery. Early termination brought on by dangerous behavior is avoided by the safety supervisor, therefore improving the efficiency of learning.

The average vehicle speed during training shown in Figure 12 indicates traffic flow. The results unequivocally reveal that algorithms with the safety supervisor retain faster training speeds than the baseline technique, that is, without safety supervision. This underlines the two advantages of the suggested strategy: it increases traffic efficiency and training stability. Due to more frequent contacts in dense traffic, where lower speeds maintain safety by lowering accident risks, vehicle speeds decrease with increasing traffic density—26 m/s, 24 m/s, and 22 m/s for Easy, Medium, and Hard traffic modes, respectively.

The MARL algorithms for every traffic density are tested over three random seeds for thirty epochs following training. With a suitable forecast horizon ($T_n \geq 7$ the MARL framework removes collisions in all traffic modes, according to the results. By contrast, the baseline approach shows collision rates of 0.07 and 0.16 for Medium and Hard traffic categories respectively. Shorter prediction horizons $T_n = 0$ or $T_n = 3$ the safety supervisor finds difficult in demanding situations. For example, the limited horizon results in "short-sighted" decisions, which causes infeasibility after a few steps, hence the collision rate in the Hard mode stays at 0.03 even $T_n = 6$. On the other hand, too wide prediction horizons $T_n = 10, 12, 14$ could spread uncertainty and cause agents to adopt too cautious behavior to preserve safety resulting in higher accident rates and slower speeds.

The safety supervisor achieves best traffic efficiency with a suitable prediction horizon $T_n = 7, 8$. For the Easy mode, for instance, the optimum average speed is obtained with the safety supervisor ($T_n = 8$ at 27.72 m/s, vs 23.52 m/s for the baseline approach. Although wider prediction horizons raise inference time, the suggested framework is still possible for real-time implementation with enough processing capacity at the infrastructure.

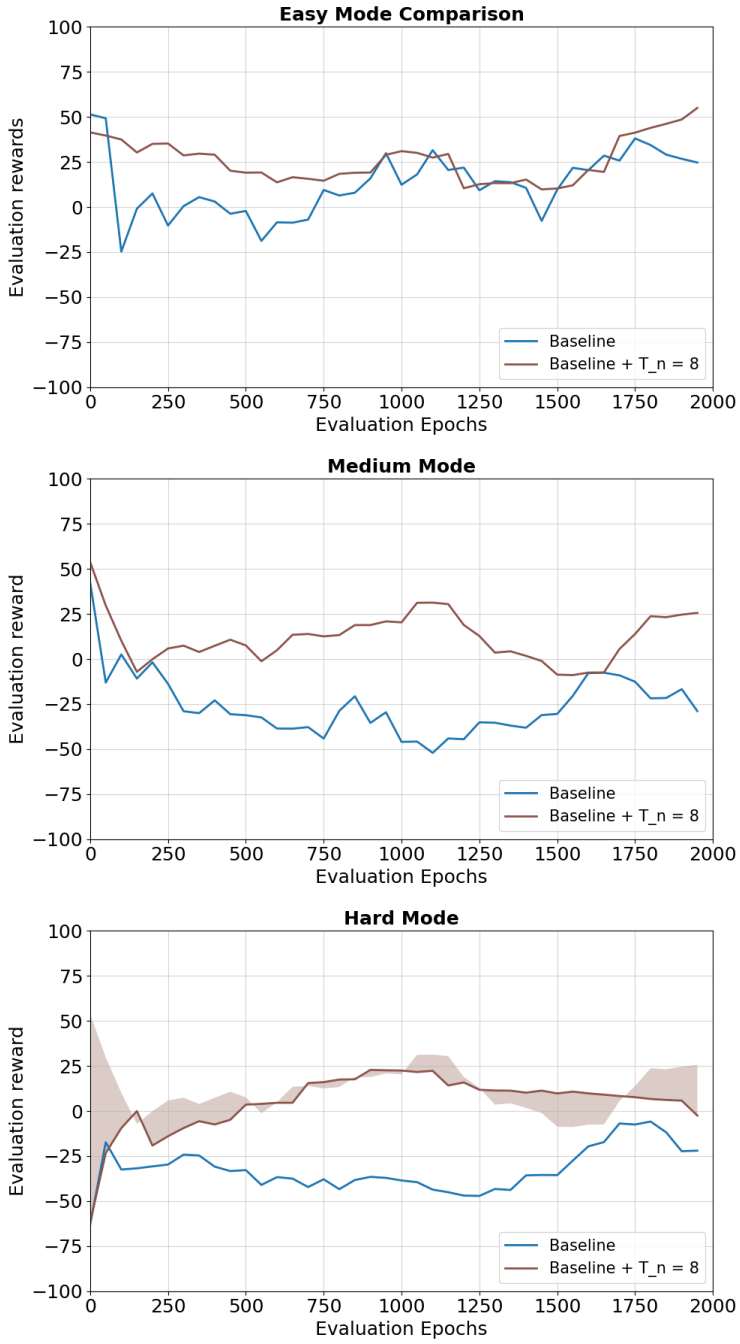

Figure 4. Training curves for the n-step priority-based safety supervisor.

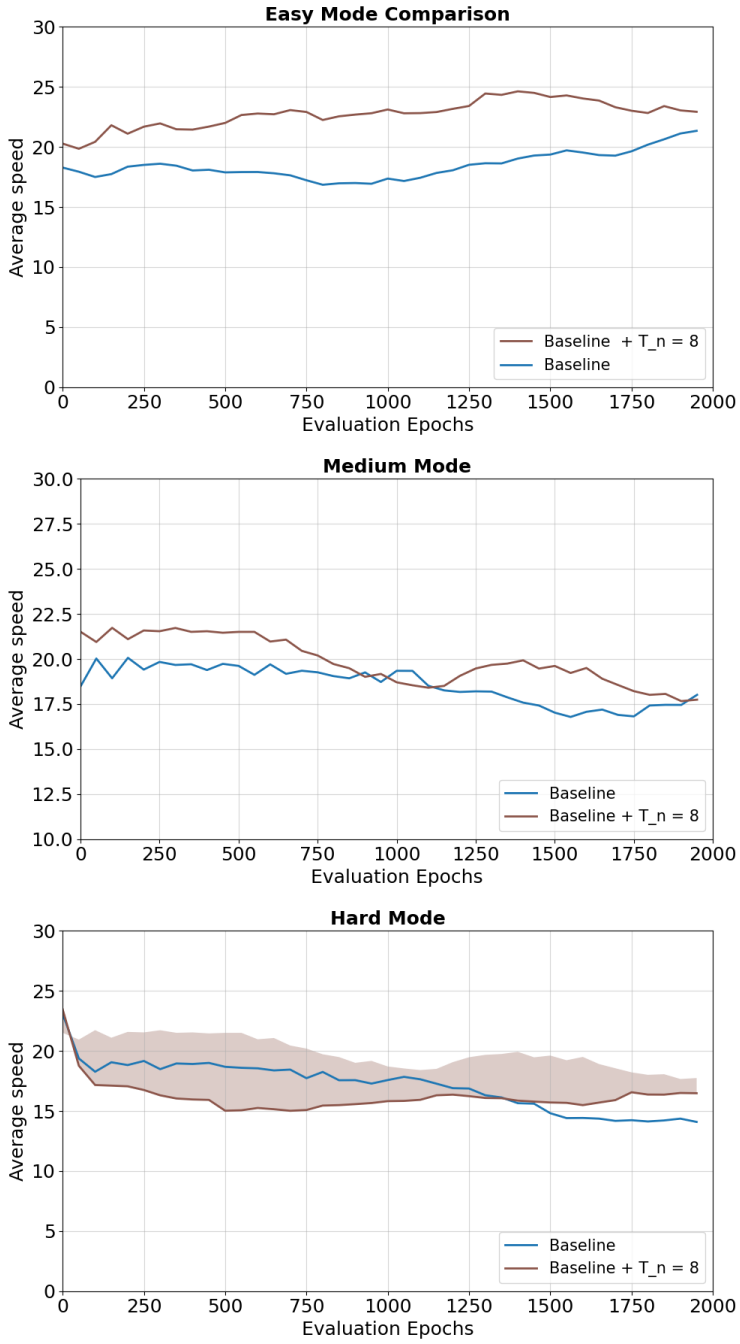

Figure 5. Average speed during training for the n-step priority-based safety supervisor

V. 4. Comparison with Benchmark Algorithms

In this subsection, we evaluate the proposed method against a state-of-the-art MARL benchmark, including MAACKTR, introduced in Section II. All MARL benchmarks rely on parameter sharing among agents to handle dynamic numbers of agents and utilize global rewards and discrete action spaces. The 8-step safety supervisor (baseline + $T_n = 8$) is applied across all traffic levels, as it offers a balanced trade-off between collision rates and prediction efficiency.

Figure 6 presents the evaluation results during training for the described MARL algorithm and MAACKTR algorithm. The proposed method (baseline + $T_n = 8$) consistently outperforms the benchmarks across all traffic levels. Notably, in the Hard traffic mode, the proposed method demonstrates significant improvements in sample efficiency and

Table 1. Testing performance comparison of collision rate and average speed (m/s) between n-step safety supervisor based on the baseline (bs) method.

Scenarios	Metrics	bs	bs + Tn = 8
Easy Mode	collision rate	0	0
	avg. speed	23.53	27.72
Medium Mode	collision rate	0.07	0
	avg. speed	20.30	24.08
Hard Mode	collision rate	0.16	0
	avg. speed	21.71	22.73

training performance over the benchmarks. Moreover, the proposed method achieves relatively higher training speeds, which enhances training efficiency.

After training, all algorithms were tested across three random seeds for 2000 epochs each, with average collision rates and vehicle speeds summarized in Table III. The results reveal that the proposed method effectively avoids collisions and achieves superior efficiency compared to other benchmark algorithms.

Furthermore, the study highlights the impact of system complexity as the number of vehicles increases. Higher traffic density exacerbates model mismatches, making it challenging to generate collision-free policies in such scenarios. These findings emphasize the robustness of model-free approaches in handling complex tasks like on-ramp merging under dense traffic conditions.

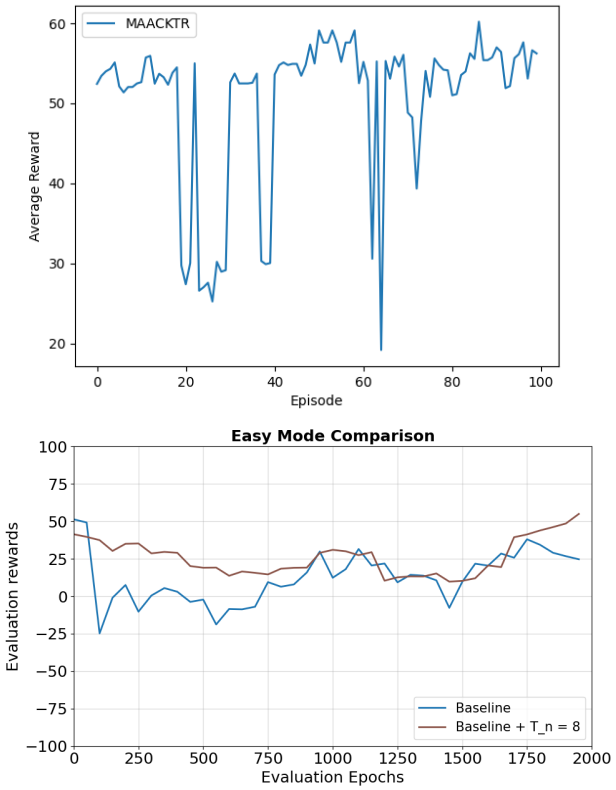

Figure 6. Training curves comparison between the proposed MARL policy (baseline (bs) + Tn = 8) and state-of-the-art MARL benchmark

VI. CONCLUSIONS AND DISCUSSIONS

From Figure 4, we observe that the baseline + Tn = 8 approach with 8 safety supervisors outperforms the MARL approach without the safety supervisor for all. - easy, medium, and hard traffic density scenarios. This is especially seen in the high traffic density mode where the performance of the 8 safety supervisors mode is significantly better than the baseline method. This can also be verified from Figure 5, where the average speed plots vs epochs is plotted, where the safety supervisor shows its potential to achieve higher average speeds and hence ensure faster navigation and higher throughput through the traffic with minimal collision rate.

In Figure 6, the performance of a state-of-the-art algorithm- MAACKTR is compared with the designed algorithm in the paper. It can be observed from this plot, that the rewards for the MAACKTR algorithm vary abruptly with frequency oscillation. This may lead to undesirable behavior and movement of the ego vehicle as compared to the proposed algorithm which showcases reasonable stability. As the training is extended further to a higher number of epochs, the proposed algorithm is expected to return even better results.

This report replicates the work of Chen et al., implementing their deep multi-agent reinforcement learning approach for autonomous highway on-ramp merging in mixed traffic. By using MARL, curriculum learning, and a priority-based safety supervisor, the framework achieves significant improvements in safety and traffic flow, addressing the challenges of dynamic and mixed-traffic environments. Further experiments and simulations can be conducted to evaluate the robustness of the approach in different traffic conditions.

REFERENCES

[1] “Future of driving,” https://www.tesla.com/autopilot, accessed: 2021-03-31.

[2] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, “Asynchronous methods for deep reinforcement learning,” in International conference on machine learning, 2016, pp. 1928–1937.

[3] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, “Playing atari with deep reinforcement learning,” arXiv preprint arXiv:1312.5602, 2013.

[4] T. Chu, J. Wang, L. Codecà, and Z. Li, “Multi-agent deep reinforcement learning for large-scale traffic signal control,” IEEE Transactions on Intelligent Transportation Systems, vol. 21, no. 3, pp. 1086–1095, 2019

[5] R. Lowe, Y. I. Wu, A. Tamar, J. Harb, O. P. Abbeel, and I. Mordatch, “Multi-agent actor-critic for mixed cooperative-competitive environments,” in Advances in neural information processing systems, 2017, pp. 6379–6390.

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

[7] Y. Wu, E. Mansimov, S. Liao, R. Grosse, and J. Ba, “Scalable trust region method for deep reinforcement learning using kronecker-factored approximation,” arXiv preprint arXiv:1708.05144, 2017.

K. Lin, R. Zhao, Z. Xu, and J. Zhou, “Efficient large-scale fleet management via multi-agent deep reinforcement learning,” in Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery Data Mining, 2018, pp. 1774–1783.

[9] N. Li, H. Chen, I. Kolmanovsky, and A. Girard, “An explicit decision tree approach for automated driving,” in Dynamic Systems and Control Conference, vol. 58271. American Society of Mechanical Engineers, 2017, p. V001T45A003

[10] T. Ayres, L. Li, D. Schleuning, and D. Young, “Preferred time-headway of highway drivers,” in ITSC 2001. 2001 IEEE Intelligent Transportation Systems. Proceedings (Cat. No. 01TH8585). IEEE, 2001, pp. 826–829.

[11]] M. Bouton, A. Nakhaei, K. Fujimura, and M. J. Kochenderfer, “Cooperation-aware reinforcement learning for merging in dense traffic,” in 2019 IEEE Intelligent Transportation Systems Conference (ITSC). IEEE, 2019, pp. 3441–3447.

APPENDIX

PYTHON CODE: MULTI-AGENT REINFORCEMENT LEARNING TRAINING AND EVALUATION

```python
import torch as torch_lib
import os
import configparser
from torch import nn, optim
import numpy as np
from common.Agent import BaseAgent
from common.Model import PolicyNetwork, ValueNetwork, SharedActorCritic
from common.utils import calculate_entropy, convert_to_tensor, encode_action, VideoLogger

# Load configurations
config_path = 'configs/configs.ini'
config = configparser.ConfigParser()
config.read(config_path)
random_seed = config.getint('MODEL_CONFIG', 'torch_seed')
torch_lib.manual_seed(random_seed)
torch_lib.backends.cudnn.benchmark = False
torch_lib.backends.cudnn.deterministic = True
os.environ['PYTHONHASHSEED'] = str(random_seed)


class MARL(BaseAgent):
    """
    Multi-Agent Reinforcement Learning (MARL) using Advantage Actor-Critic (A2C)
    """

    def __init__(self, environment, state_size, action_size, **kwargs):
        super(MARL, self).__init__(environment, state_size, action_size, **kwargs)

        # Configuration Parameters
        self.env = environment
        self.state_dim = state_size
        self.action_dim = action_size
        self.rollout_steps = kwargs.get('rollout_steps', 10)
        self.gamma = kwargs.get('discount_factor', 0.99)
        self.entropy_weight = kwargs.get('entropy_weight', 0.01)
        self.gradient_clipping = kwargs.get('max_gradient_norm', 0.5)
        self.batch_limit = kwargs.get('batch_size', 64)
        self.actor_lr = kwargs.get('actor_learning_rate', 0.0005)
        self.critic_lr = kwargs.get('critic_learning_rate', 0.001)
        self.shared_policy = kwargs.get('shared_policy', True)
        self.use_gpu = torch_lib.cuda.is_available()

        # Define Actor and Critic Networks
        if self.shared_policy:
            self.actor_critic = SharedActorCritic(self.state_dim, self.action_dim, kwargs.get('
                hidden_layer_size', 128))
            self.optimizer = optim.Adam(self.actor_critic.parameters(), lr=self.actor_lr)
        else:
            self.policy_network = PolicyNetwork(self.state_dim, self.action_dim)
            self.value_network = ValueNetwork(self.state_dim)
            self.policy_optimizer = optim.Adam(self.policy_network.parameters(), lr=self.actor_lr
                )
            self.value_optimizer = optim.Adam(self.value_network.parameters(), lr=self.critic_lr)

        if self.use_gpu:
            if self.shared_policy:
                self.actor_critic.cuda()
            else:
                self.policy_network.cuda()
                self.value_network.cuda()

        # Storage for tracking performance
```

```
        self.training_rewards = []
        self.episode_lengths = []
        self.average_speeds = []

    def collect_experience(self):
        """
        Collects experience by interacting with the environment over a series of steps.
        """
        states, actions, rewards, policies, masks = [], [], [], [], []
        total_reward = 0
        done = True

        for _ in range(self.rollout_steps):
            if done:
                state = self.env.reset()
            state_tensor = convert_to_tensor(state, self.use_gpu)

            # Get action and policy probabilities
            if self.shared_policy:
                policy_distribution, _ = self.actor_critic(state_tensor)
            else:
                policy_distribution = self.policy_network(state_tensor)

            action = torch_lib.multinomial(policy_distribution, 1).item()
            next_state, reward, done, _ = self.env.step(action)

            # Record the results
            states.append(state)
            actions.append(encode_action(action, self.action_dim))
            rewards.append(reward)
            policies.append(policy_distribution)
            masks.append(1 - int(done))

            state = next_state
            total_reward += reward

        return states, actions, rewards, policies, masks, total_reward

    def compute_loss(self, states, actions, rewards, policies, masks):
        """
        Computes the actor and critic losses based on collected experiences.
        """
        states_tensor = convert_to_tensor(states, self.use_gpu)
        actions_tensor = convert_to_tensor(actions, self.use_gpu)
        rewards_tensor = convert_to_tensor(rewards, self.use_gpu)
        masks_tensor = convert_to_tensor(masks, self.use_gpu)

        if self.shared_policy:
            policy_distribution, value_estimate = self.actor_critic(states_tensor)
        else:
            policy_distribution = self.policy_network(states_tensor)
            value_estimate = self.value_network(states_tensor)

        log_probs = policy_distribution.log_prob(actions_tensor)
        advantages = rewards_tensor - value_estimate
        policy_loss = -(log_probs * advantages.detach()).mean()
        value_loss = nn.MSELoss()(value_estimate, rewards_tensor)
        entropy_loss = -calculate_entropy(policy_distribution) * self.entropy_weight

        return policy_loss, value_loss, entropy_loss

    def train_step(self, memory):
        """
        Trains the model for one step using sampled experiences.
        """
        experiences = memory.sample(self.batch_limit)
        states, actions, rewards, policies, masks = experiences

```

```
        policy_loss, value_loss, entropy_loss = self.compute_loss(states, actions, rewards,
            policies, masks)
        total_loss = policy_loss + value_loss + entropy_loss

        self.optimizer.zero_grad()
        total_loss.backward()
        if self.gradient_clipping:
            nn.utils.clip_grad_norm_(self.actor_critic.parameters(), self.gradient_clipping)
        self.optimizer.step()

    def evaluate(self, environment, episodes=10):
        """
        Evaluates the model over a number of episodes.
        """
        total_rewards = []
        for _ in range(episodes):
            state = environment.reset()
            done = False
            episode_reward = 0
            while not done:
                state_tensor = convert_to_tensor(state, self.use_gpu)
                if self.shared_policy:
                    action = self.actor_critic(state_tensor).argmax().item()
                else:
                    action = self.policy_network(state_tensor).argmax().item()
                state, reward, done, _ = environment.step(action)
                episode_reward += reward
            total_rewards.append(episode_reward)
        return np.mean(total_rewards)
```

Listing 1. Python Code for Proposed algorithm

SCRIPT FOR TRAINING THE PROPOSED MULTI-AGENT REINFORCEMENT LEARNING (MARL) MODEL IN A DYNAMIC ENVIRONMENT

```python
from __future__ import print_function, division
from MARL import MARL
from common.utils import agg_double_list, copy_file, init_dir
from datetime import datetime

import argparse
import configparser
import sys
import os
import gym
import numpy as np
import matplotlib.pyplot as plt
import highway_env  # Import the highway environment

# Add the highway environment path
sys.path.append("../highway-env")


def parse_args():
    """
    Parses command-line arguments for training or evaluation.
    """
    default_base_dir = "./results/"
    default_config_dir = 'configs/configs.ini'

    parser = argparse.ArgumentParser(description='Train or evaluate a policy on an RL environment
        using MARL.')

    parser.add_argument('--base-dir', type=str, default=default_base_dir,
                        help="Base directory to save results.")
    parser.add_argument('--option', type=str, default='train',
                        help="'train' to train the model, 'evaluate' to evaluate the model.")
    parser.add_argument('--config-dir', type=str, default=default_config_dir,
                        help="Path to the configuration file.")
    parser.add_argument('--model-dir', type=str, default='',
                        help="Path to a pretrained model directory.")
    parser.add_argument('--evaluation-seeds', type=str, default=','.join([str(i) for i in range
        (0, 600, 20)]),
                        help="Random seeds for evaluation, separated by commas.")

    return parser.parse_args()


def train(args):
    """
    Train the MARL model using the configuration provided in the config file.
    """
    base_dir = args.base_dir
    config_dir = args.config_dir
    config = configparser.ConfigParser()
    config.read(config_dir)

    # Create an experiment folder
    timestamp = datetime.now().strftime("%b-%d_%H_%M_%S")
    output_dir = os.path.join(base_dir, timestamp)
    dirs = init_dir(output_dir)
    copy_file(dirs['configs'])

    model_dir = args.model_dir if os.path.exists(args.model_dir) else dirs['models']

    # Load model and training configuration
    BATCH_SIZE = config.getint('MODEL_CONFIG', 'BATCH_SIZE')
    MEMORY_CAPACITY = config.getint('MODEL_CONFIG', 'MEMORY_CAPACITY')
    ROLL_OUT_N_STEPS = config.getint('MODEL_CONFIG', 'ROLL_OUT_N_STEPS')
```

```
    reward_gamma = config.getfloat('MODEL_CONFIG', 'reward_gamma')
    training_strategy = config.get('MODEL_CONFIG', 'training_strategy')
    actor_hidden_size = config.getint('MODEL_CONFIG', 'actor_hidden_size')
    critic_hidden_size = config.getint('MODEL_CONFIG', 'critic_hidden_size')
    MAX_GRAD_NORM = config.getfloat('MODEL_CONFIG', 'MAX_GRAD_NORM')
    ENTROPY_REG = config.getfloat('MODEL_CONFIG', 'ENTROPY_REG')
    epsilon = config.getfloat('MODEL_CONFIG', 'epsilon')
    alpha = config.getfloat('MODEL_CONFIG', 'alpha')
    shared_network = config.getboolean('MODEL_CONFIG', 'shared_network')
    reward_type = config.get('MODEL_CONFIG', 'reward_type')

    # Training configurations
    actor_lr = config.getfloat('TRAIN_CONFIG', 'actor_lr')
    critic_lr = config.getfloat('TRAIN_CONFIG', 'critic_lr')
    MAX_EPISODES = config.getint('TRAIN_CONFIG', 'MAX_EPISODES')
    EPISODES_BEFORE_TRAIN = config.getint('TRAIN_CONFIG', 'EPISODES_BEFORE_TRAIN')
    EVAL_INTERVAL = config.getint('TRAIN_CONFIG', 'EVAL_INTERVAL')
    EVAL_EPISODES = config.getint('TRAIN_CONFIG', 'EVAL_EPISODES')
    reward_scale = config.getfloat('TRAIN_CONFIG', 'reward_scale')

    # Initialize environment
    env = gym.make('merge-multi-agent-v0')
    env_eval = gym.make('merge-multi-agent-v0')

    for param in ['seed', 'simulation_frequency', 'duration', 'policy_frequency', '
        COLLISION_REWARD',
                  'HIGH_SPEED_REWARD', 'HEADWAY_COST', 'HEADWAY_TIME', 'MERGING_LANE_COST', '
                      traffic_density',
                  'safety_guarantee', 'n_step']:
        env.config[param] = config.get('ENV_CONFIG', param, fallback=None)

    env_eval.config.update(env.config)

    # Model initialization
    state_dim = env.n_s
    action_dim = env.n_a
    marl = MARL(env, state_dim, action_dim,
                memory_capacity=MEMORY_CAPACITY, roll_out_n_steps=ROLL_OUT_N_STEPS,
                reward_gamma=reward_gamma, reward_scale=reward_scale,
                actor_hidden_size=actor_hidden_size, critic_hidden_size=critic_hidden_size,
                actor_lr=actor_lr, critic_lr=critic_lr,
                entropy_reg=ENTROPY_REG, batch_size=BATCH_SIZE,
                training_strategy=training_strategy, shared_network=shared_network,
                reward_type=reward_type, max_grad_norm=MAX_GRAD_NORM)

    # Load existing model if available
    marl.load(model_dir, train_mode=True)

    episodes, eval_rewards = [], []
    best_eval_reward = -float('inf')

    while marl.n_episodes < MAX_EPISODES:
        marl.explore()
        if marl.n_episodes >= EPISODES_BEFORE_TRAIN:
            marl.train()

        # Evaluate periodically
        if marl.episode_done and (marl.n_episodes % EVAL_INTERVAL == 0):
            rewards, *_ = marl.evaluation(env_eval, dirs['train_videos'], EVAL_EPISODES)
            avg_reward = np.mean(rewards)
            print(f"Episode {marl.n_episodes}, Avg. Reward: {avg_reward:.2f}")
            episodes.append(marl.n_episodes)
            eval_rewards.append(avg_reward)

            if avg_reward > best_eval_reward:
                marl.save(dirs['models'], marl.n_episodes)
                best_eval_reward = avg_reward

```

```
        np.save(os.path.join(output_dir, 'eval_rewards.npy'), np.array(eval_rewards))

    # Save training progress and final model
    marl.save(dirs['models'], MAX_EPISODES)
    plt.figure()
    plt.plot(eval_rewards)
    plt.xlabel("Episodes")
    plt.ylabel("Evaluation Rewards")
    plt.savefig(os.path.join(output_dir, "training_rewards.png"))
    plt.show()


def evaluate(args):
    """
    Evaluate a pretrained MARL model.
    """
    if not os.path.exists(args.model_dir):
        raise FileNotFoundError("Pretrained model not found.")

    config = configparser.ConfigParser()
    config.read(os.path.join(args.model_dir, 'configs.ini'))

    env = gym.make('merge-multi-agent-v0')
    for param in ['seed', 'simulation_frequency', 'duration', 'policy_frequency', '
        COLLISION_REWARD',
                  'HIGH_SPEED_REWARD', 'HEADWAY_COST', 'HEADWAY_TIME', 'MERGING_LANE_COST', '
                      traffic_density',
                  'safety_guarantee', 'n_step']:
        env.config[param] = config.get('ENV_CONFIG', param, fallback=None)

    marl = MARL(env, state_dim=env.n_s, action_dim=env.n_a)
    marl.load(args.model_dir)

    rewards, *_ = marl.evaluation(env, args.model_dir, eval_episodes=config.getint('TRAIN_CONFIG'
        , 'EVAL_EPISODES'))
    print(f"Average Rewards: {np.mean(rewards):.2f}")


if __name__ == "__main__":
    args = parse_args()
    if args.option == 'train':
        train(args)
    elif args.option == 'evaluate':
        evaluate(args)
    else:
        raise ValueError("Invalid option. Use 'train' or 'evaluate'.")
```

Listing 2. Script for training the proposed Multi-Agent Reinforcement Learning (MARL) model in a dynamic environment

CONFIGURATION FILE

The following is the configuration file used in the project:

```
[MODEL_CONFIG]
epsilon = 1e-5;
alpha = 0.99;
reward_gamma = 0.99
MAX_GRAD_NORM = 5
ROLL_OUT_N_STEPS = 100
MEMORY_CAPACITY = 100
; only use the latest ROLL_OUT_N_STEPS for training A2C
BATCH_SIZE = 100
ENTROPY_REG = 0.01
; seeds for pytorch, 0, 2000, 2021
torch_seed = 2021

; concurrent
training_strategy = concurrent
actor_hidden_size = 128
critic_hidden_size = 128
shared_network = True
action_masking = True
state_split = True
; "greedy", "regionalR", "global_R"
reward_type = regionalR

[TRAIN_CONFIG]
MAX_EPISODES = 2000
EPISODES_BEFORE_TRAIN = 1
EVAL_EPISODES = 3
EVAL_INTERVAL = 50
reward_scale = 20.
actor_lr = 5e-4
critic_lr = 5e-4
test_seeds =
    0,25,50,75,100,125,150,175,200,325,350,375,400,425,450,475,500,525,550,575

[ENV_CONFIG]
; seed for the environment, 0, 2000, 2021
seed = 2021
simulation_frequency = 15
duration = 20
policy_frequency = 5
COLLISION_REWARD = 200
HIGH_SPEED_REWARD = 1
HEADWAY_COST = 4
HEADWAY_TIME = 1.2
MERGING_LANE_COST = 4
traffic_density = 3
safety_guarantee = True
n_step = 8
```

Listing 3. Configuration File: configs.ini

CUSTOM HIGHWAY MERGING ENVIRONMENT

```python
import numpy as np
from gym.envs.registration import register
from typing import Tuple

from highway_env import utils
from highway_env.envs.common.abstract import AbstractEnv, MultiAgentWrapper
from highway_env.road.lane import LineType, StraightLane, SineLane
from highway_env.road.road import Road, RoadNetwork
from highway_env.vehicle.controller import ControlledVehicle, MDPVehicle
from highway_env.road.objects import Obstacle
from highway_env.vehicle.kinematics import Vehicle


class MergeEnvironment(AbstractEnv):
    """
    A simulation environment for highway merging scenarios.

    The focus vehicle navigates a highway segment that includes a merge point, where vehicles
    enter from an on-ramp. Rewards are based on maintaining high speeds, avoiding collisions,
    and facilitating smooth merging interactions.
    """
    n_a = 5
    n_s = 25

    @classmethod
    def default_config(cls) -> dict:
        config = super().default_config()
        config.update({
            "observation": {
                "type": "Kinematics"},
            "action": {
                "type": "DiscreteMetaAction",
                "longitudinal": True,
                "lateral": True},
            "controlled_vehicles": 1,
            "screen_width": 600,
            "screen_height": 120,
            "centering_position": [0.3, 0.5],
            "scaling": 3,
            "simulation_frequency": 15,  # [Hz]
            "duration": 20,  # total duration
            "policy_frequency": 5,  # [Hz]
            "reward_speed_range": [10, 30],
            "COLLISION_REWARD": -200,  # Penalty for collisions
            "HIGH_SPEED_REWARD": 1,  # Reward for maintaining high speed
            "HEADWAY_COST": 4,  # Cost for insufficient spacing
            "HEADWAY_TIME": 1.2,  # [s]
            "MERGING_LANE_COST": 4,  # Cost for staying in merging lane
            "traffic_density": 1,  # Levels of traffic density (easy/hard)
        })
        return config

    def _reward(self, action: list) -> float:
        """Calculate the overall reward for all controlled vehicles."""
        return sum(self._vehicle_reward(action, vehicle) for vehicle in self.controlled_vehicles)
              \
               / len(self.controlled_vehicles)

    def _vehicle_reward(self, action: int, vehicle: Vehicle) -> float:
        """
        Compute the reward for an individual vehicle.
        :param action: The action taken by the vehicle.
        :param vehicle: The vehicle for which the reward is calculated.
        :return: A reward value for the given state-action pair.
        """
        speed_ratio = utils.lmap(vehicle.speed, self.config["reward_speed_range"], [0, 1])
```

```python

        # Penalize for being in the merging lane
        if vehicle.lane_index == ("b", "c", 1):
            merging_penalty = -np.exp(-(vehicle.position[0] - sum(self.ends[:3])) ** 2 / (10 *
                self.ends[2]))
        else:
            merging_penalty = 0

        # Penalize for inadequate headway
        headway_dist = self._compute_headway_distance(vehicle)
        headway_penalty = np.log(headway_dist / (self.config["HEADWAY_TIME"] * vehicle.speed)) if
             vehicle.speed > 0 else 0

        # Compute total reward
        reward = (
            self.config["COLLISION_REWARD"] * (-1 * vehicle.crashed)
            + (self.config["HIGH_SPEED_REWARD"] * np.clip(speed_ratio, 0, 1))
            + self.config["MERGING_LANE_COST"] * merging_penalty
            + self.config["HEADWAY_COST"] * (headway_penalty if headway_penalty < 0 else 0)
        )
        return reward

    def _regional_reward(self):
        """Calculate rewards for vehicles within a regional context."""
        for vehicle in self.controlled_vehicles:
            neighbor_vehicles = []

            # Determine vehicles in the vicinity
            if vehicle.lane_index in [("a", "b", 0), ("b", "c", 0), ("c", "d", 0)]:
                v_front, v_rear = self.road.surrounding_vehicles(vehicle)
                if self.road.network.side_lanes(vehicle.lane_index):
                    v_side_front, v_side_rear = self.road.surrounding_vehicles(vehicle, self.road
                        .network.side_lanes(vehicle.lane_index)[0])
                else:
                    v_side_front, v_side_rear = None, None
            else:
                v_side_front, v_side_rear = self.road.surrounding_vehicles(vehicle)
                v_front, v_rear = None, None

            # Collect all neighbor vehicles
            for v in [v_front, v_side_front, vehicle, v_rear, v_side_rear]:
                if isinstance(v, MDPVehicle) and v:
                    neighbor_vehicles.append(v)

            regional_reward = sum(v.local_reward for v in neighbor_vehicles)
            vehicle.regional_reward = regional_reward / len(neighbor_vehicles)

    def step(self, action: int) -> Tuple[np.ndarray, float, bool, dict]:
        """Perform an environment step."""
        obs, reward, done, info = super().step(action)
        obs = np.array(obs).reshape((len(obs), -1))
        info["agents_rewards"] = tuple(v.local_reward for v in self.controlled_vehicles)
        return obs, reward, done, info

    def _reset(self, num_CAV=0) -> None:
        """Reset the environment for a new episode."""
        self._make_road()
        self._populate_road(num_CAV)
        self.action_is_safe = True
        self.T = int(self.config["duration"] * self.config["policy_frequency"])

    def _make_road(self) -> None:
        """Construct the road network."""
        net = RoadNetwork()

        c, s = LineType.CONTINUOUS_LINE, LineType.STRIPED
        net.add_lane("a", "b", StraightLane([0, 0], [sum(self.ends[:2]), 0], line_types=[c, c]))
```

```
        net.add_lane("b", "c", StraightLane([sum(self.ends[:2]), 0], [sum(self.ends[:3]), 0],
            line_types=[c, s]))
        net.add_lane("c", "d", StraightLane([sum(self.ends[:3]), 0], [sum(self.ends), 0],
            line_types=[c, c]))

        road = Road(network=net, np_random=self.np_random, record_history=self.config["
            show_trajectories"])
        self.road = road

    def _populate_road(self, num_CAV: int) -> None:
        """Populate the road with vehicles."""
        road = self.road
        self.controlled_vehicles = []

        # Spawn vehicles
        for _ in range(num_CAV):
            vehicle = self.action_type.vehicle_class(road, road.network.get_lane(("a", "b", 0)).
                position(10, 0), speed=25)
            self.controlled_vehicles.append(vehicle)
            road.vehicles.append(vehicle)


register(
    id='merge-v1',
    entry_point='highway_env.envs:MergeEnvironment',
)

register(
    id='merge-multi-agent-v0',
    entry_point='highway_env.envs:MergeEnvironment',
)
```

Listing 4. Script for training the proposed Multi-Agent Reinforcement Learning (MARL) model in a dynamic environment