R&D Project Report (SC 691)

Control Barrier Certificates for Multi-agent System

Submitted By:

Raj Kumar

Guided By:

Prof. Mayank Baranwal

Systems and Control Engineering
Indian Institute of Technology Bombay
2024-25

Abstract

This project builds upon the framework introduced in the paper *Learning Safe Multi-Agent Control with Decentralized Neural Barrier Certificates* by Qin et al. (ICLR 2021). The project aims to adapt the decentralized learning methodology for control barrier functions (CBFs) and control policies to a PyTorch framework. This adaptation enhances the modularity and scalability of the codebase, enabling future integration of additional features and experiments. The implementation demonstrates improved safety and task performance in multi-agent navigation scenarios, validated through extensive simulations by implementing different types of configurations different from the original implementation.

Introduction

Related Work

Multi-agent systems have been extensively studied in the context of safe navigation and control. Classical methods for ensuring safety rely on centralized control strategies or handcrafted barrier functions, which can be computationally expensive and inflexible in dynamic environments. Recent approaches, such as reinforcement learning, have focused on policy optimization, but they often lack rigorous safety guarantees. The framework introduced by Qin et al. uniquely combines control barrier functions and decentralized learning to provide both scalability and safety guarantees, making it a significant advancement in this field.

Control Lyapunov Functions (CLFs)

Control Lyapunov Functions (CLFs) are a powerful tool used in control theory to design feedback controllers that stabilize dynamic systems. A CLF is a scalar function $V : \mathbb{R}^n \to \mathbb{R}$ that serves as a measure of the system's energy or deviation from a desired equilibrium point. The main idea is to ensure that V decreases over time under the influence of a suitable control input.

Given a dynamical system described by:

$$\dot{x}(t) = f(x(t), u(t)),$$

where $x \in \mathbb{R}^n$ represents the state and $u \in \mathbb{R}^m$ the control input, a function $V(x)$ is a CLF if:

1. $V(x) > 0$ for all $x \neq 0$, and $V(0) = 0$ (positive definiteness).
2. There exists a control input $u(x)$ such that the derivative of V along the system's trajectory satisfies:

$$\dot{V}(x,u) = \nabla V(x) \cdot f(x,u) < 0 \quad \forall x \neq 0.$$

The existence of a CLF guarantees that a stabilizing control law $u(x)$ can be designed, leading the system's state to the equilibrium point $x = 0$.

Control Barrier Functions (CBFs)

Control Barrier Functions (CBFs) extend the concept of Lyapunov functions to enforce safety constraints on dynamical systems. While CLFs ensure stability, CBFs ensure that the system remains within a predefined safe set $S \subset \mathbb{R}^n$. The safe set is defined as:

$$S = \{x \in \mathbb{R}^n \mid h(x) \geq 0\},$$

where $h : \mathbb{R}^n \to \mathbb{R}$ is a continuously differentiable function. A CBF is a function $h(x)$ that satisfies the following conditions:

1. $h(x) > 0$ for all $x \in S$, and $h(x) < 0$ for $x \notin S$ (safety conditions).
2. There exists a control input $u(x)$ such that:

   $$\dot{h}(x,u) = \nabla h(x) \cdot f(x,u) + \alpha(h(x)) \geq 0 \quad \forall x \in S,$$

   where $\alpha(\cdot)$ is a class-$\mathcal{K}$ function ensuring that $h(x)$ does not decrease in value.

CBFs are particularly useful for designing controllers that guarantee safety by preventing the system's state from entering unsafe regions, while allowing task-specific goals to be achieved.

Integration of CLFs and CBFs

By combining CLFs and CBFs, it is possible to design controllers that achieve both stability and safety. This is often done by formulating a Quadratic Programming (QP) problem that seeks to minimize the control effort while satisfying the conditions imposed by the CLFs and CBFs:

$$\min_u \|u\|^2 \quad \text{subject to } \dot{V}(x,u) < 0 \text{ and } \dot{h}(x,u) \geq 0.$$

This unified framework provides a principled approach for synthesizing controllers that respect both stability and safety requirements in complex dynamical systems.
However, formulating CBF's for a complex dynamics systems is a very difficult task, which stems our interest toward learning based CBF and the referenced paper explores that.

Paper Review

The paper presented jointly learning safe multi-agent control policies and CBF certificates, in a decentralized fashion. The control policy is decentralized (i.e., each agent has its own control policy and there does not exist a central controller to coordinate all the agents)

The goal of this project is to implement the decentralized framework proposed in the paper for jointly learning control policies and Control Barrier Functions (CBFs) to ensure safety in multi-agent systems. Below, we explain the key components and process of the learning framework.

Key Notations

- $\pi_i(s_i, o_i)$: Control policy for agent i, mapping state s_i and observation o_i to action u_i.
- $h_i(s_i, o_i)$: Control Barrier Function (CBF) for agent i, ensuring safety.
- $\mathcal{H}_i$: Set of candidate CBFs parameterized by neural networks.
- $\mathcal{V}_i$: Set of candidate control policies parameterized by neural networks.
- τ_i: Trajectory of state and observation pairs for agent i over a time interval T.
- γ: Safety margin to ensure robust satisfaction of CBF conditions.

Joint Learning of Policies and Control Barrier Functions

The learning framework aims to train two key components for each agent i:

- The **control policy** $\pi_i(s_i, o_i)$, which determines the action u_i for agent i based on its state s_i and local observation o_i.
- The **control barrier function** (CBF) $h_i(s_i, o_i)$, which acts as a safety certificate, ensuring that the agent avoids unsafe states during operation.

The policy and CBF are jointly learned such that the **CBF conditions** are satisfied, guaranteeing the safety of the multi-agent system. These conditions are:

$$\begin{aligned} h_i(s_i, o_i) &\geq 0 \quad \text{for all states in the safe set,} && (1)\\ h_i(s_i, o_i) &< 0 \quad \text{for states in the dangerous set,} && (2)\\ \dot{h}_i(s_i, o_i, u_i) = \nabla h_i \cdot f(s_i, u_i) + \alpha(h_i) &\geq 0 \quad \text{to ensure safety under dynamics.} && (3) \end{aligned}$$

Optimization Objective

The optimization objective is to jointly train the control policy π_i and CBF h_i by ensuring that all safety conditions are satisfied. For a given trajectory $\tau_i = \{(s_i(t), o_i(t)) : t \in T\}$, the framework defines the following function:

$$y_i(\tau_i, h_i, \pi_i) := \min \left\{ \inf_{X_{i,0} \cap \tau_i} h_i(s_i, o_i),\ \inf_{X_{i,d} \cap \tau_i} -h_i(s_i, o_i),\ \inf_{X_{i,h} \cap \tau_i} \left(\dot{h}_i + \alpha(h_i) \right) \right\},$$

where:

- $X_{i,0}$ is the set of initial states that must be safe.
- $X_{i,d}$ is the dangerous set that must be avoided.
- $X_{i,h}$ is the set of states where $h_i \geq 0$, corresponding to the safe set.

The optimization problem can then be formulated as:

$$\text{For all agents } i, \quad \max_{h_i \in \mathcal{H}_i,\, \pi_i \in \mathcal{V}_i} y_i(\tau_i, h_i, \pi_i) \quad \text{subject to } y_i(\tau_i, h_i, \pi_i) \geq \gamma,$$

where $\gamma > 0$ is a safety margin, $\mathcal{H}_i$ is the set of candidate CBFs, and $\mathcal{V}_i$ is the set of candidate control policies.

Iterative Training Process

The framework employs an iterative process to train the policies and CBFs:

1. **Data Collection:** At each iteration, the current control policy π_i is used to simulate trajectories τ_i for each agent in the environment. These trajectories consist of state-observation pairs (s_i, o_i).

2. **Loss Computation:** A composite loss function is defined to enforce both safety and task completion:
$$L = L_c + \eta L_g,$$
where:

   - L_c: Enforces the CBF conditions (safety). It is defined as:
$$L_c = \sum_i \left(\sum_{s_i \in X_{i,0}} \max(0, \gamma - h_i(s_i, o_i)) + \sum_{s_i \in X_{i,d}} \max(0, \gamma + h_i(s_i, o_i)) + \sum_{s_i \in X_{i,h}} \max(0, \gamma - \dot{h}_i) \right).$$

   - L_g: Encourages goal-reaching by minimizing the deviation between the control input $\pi_i(s_i, o_i)$ and a reference control input u_i^g (e.g., from an LQR or PID controller):
$$L_g = \sum_i \sum_{s_i \in X} \|\pi_i(s_i, o_i) - u_i^g(s_i)\|^2.$$

3. **Parameter Update:** Using the computed loss, the neural networks for π_i and h_i are updated via backpropagation and stochastic gradient descent.

4. **Iteration:** The updated controllers are deployed in simulation to collect new trajectories, and the process is repeated until convergence.

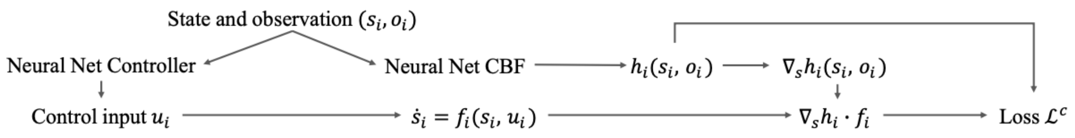

Figure 1: The computational graph of the control-certificate jointly learning framework in multi-agent systems. Only the graph for agent i is shown because agents have the same graph and the computation is decentralized.

Generalization to Testing Scenarios

The iterative process ensures that the learned policies and CBFs generalize well to unseen testing scenarios. By collecting training data on-policy (i.e., using the current controller), the distribution shift between training and testing is minimized. The learned CBFs guarantee safety even under dynamic conditions, and the policies effectively guide the agents toward their goals.

Key Contributions and Results of the Paper

The paper demonstrated the effectiveness of jointly learning decentralized control policies and CBFs in multi-agent systems. Key contributions include:

- A novel decentralized learning framework for scalable and safe multi-agent control.
- Experimental validation with up to 1024 agents, showing near-perfect safety and task completion rates in both static and dynamic environments.
- Strong generalization performance to unseen testing scenarios.

While the results are promising, challenges remain in improving the modularity of the codebase and extending the framework to new environments, which are addressed in this project.

Motivation for PyTorch Implementation

The TensorFlow-based implementation provided in the paper served as the foundation for this project. However, PyTorch offers improved modularity, flexibility, and accessibility, which are essential for incorporating new features and performing custom experiments. This project aimed to replicate the learning framework in PyTorch while ensuring scalability and extensibility.

PyTorch Implementation

Code Structure Overview

The PyTorch implementation is divided into four main components, each encapsulated in a separate file for modularity and clarity:

- **config.py:** Contains all hyperparameters and configurations for the learning framework.
- **core.py:** Defines utility functions, the neural network architectures for control policies and CBFs, and other core functionalities required for the framework.
- **train.py:** Implements the training pipeline, including data collection, loss computation, and parameter updates.
- **evaluate.py:** Provides functions to evaluate the performance of the learned policies and CBFs on testing scenarios.

The modular structure ensures scalability, ease of debugging, and adaptability for additional features.

Core.py

Core Functions and Neural Networks

The `core.py` file is a foundational part of the project that implements key functionalities required for training and evaluation of the control policies and Control Barrier Functions (CBFs). These include:

- Initial Position and goal generation lying in safe set
- Neural network definitions for CBF computation and action selection.
- Loss functions for training the CBF and control policies.
- Helper functions for collision detection and agent filtering.

Neural Networks for CBF and Actions

Two key neural networks are implemented in `core.py`:

- **CBF Network:** Computes the Control Barrier Function $h(x)$, which ensures safety by preventing collisions.
- **Action Network:** Computes the control actions u for each agent, ensuring they navigate efficiently toward their goals while satisfying safety constraints.

Both networks use one-dimensional convolutional layers to process pairwise interactions between agents, followed by fully connected layers to compute outputs.

Code Example: CBF Network

```
def network_cbf(x, r, indices=None):
    x, indices = remove_distant_agents(x=x, k=config.TOP_K,
       indices=indices)
    x = tf.contrib.layers.conv1d(
        inputs=x,
        num_outputs=64,
        kernel_size=1,
        activation_fn=tf.nn.relu
    )
    x = tf.contrib.layers.conv1d(
        inputs=x,
        num_outputs=128,
        kernel_size=1,
        activation_fn=tf.nn.relu
    )
    x = tf.contrib.layers.conv1d(
        inputs=x,
        num_outputs=1,
        kernel_size=1
    )
    return x
```

Loss Functions

Several loss functions are defined in `core.py` to train the CBF and policy networks:

- **Barrier Loss:** Ensures the CBF $h(x)$ satisfies safety conditions, penalizing violations of the safe set.
- **Derivative Loss:** Encourages $\dot{h}(x, u) \geq -\alpha h(x)$, ensuring system dynamics respect the barrier conditions.

- **Action Loss:** Minimizes the deviation of the learned actions from reference actions (e.g., generated by a classical controller).

Code Example: Barrier Loss

```
def loss_barrier(h, s, r, ttc, indices=None):
    dang_mask = ttc_dangerous_mask(s, r=r, ttc=ttc,
       indices=indices)
    dang_h = tf.boolean_mask(h, dang_mask)
    loss_dang = tf.reduce_sum(tf.math.maximum(dang_h + eps[0],
       0)) / num_dang
    return loss_dang
```

The loss penalizes unsafe trajectories, improving the CBF's ability to enforce safety constraints.

Helper Functions

Several helper functions facilitate efficient training:

- **remove_distant_agents:** Filters out agents that are beyond a specified observation radius, reducing computational overhead.
- **ttc_dangerous_mask:** Identifies agents on a collision course by calculating the Time-To-Collision (TTC) based on current positions and velocities.

Code Example: TTC Dangerous Mask

```
def ttc_dangerous_mask(s, r, ttc, indices=None):
    x, y, vx, vy = tf.split(s_diff, 4, axis=2)
    alpha = vx ** 2 + vy ** 2
    beta = 2 * (x * vx + y * vy)
    gamma = x ** 2 + y ** 2 - r ** 2
    ttc_dangerous = tf.logical_or(dist_dangerous,
       has_root_less_than_ttc)
    return ttc_dangerous
```

This function identifies potentially unsafe agents based on their trajectories and speeds.

Train.py

Neural Network Integration

These networks are instantiated using PyTorch and optimized separately, allowing modular and efficient parameter updates.

Code Example:

```
model_cbf = core.NetworkCBF().to(device)
model_action = core.NetworkAction().to(device)
optimizer_cbf = optim.Adam(model_cbf.parameters(),
   lr=config.LEARNING_RATE)
optimizer_action = optim.Adam(model_action.parameters(),
   lr=config.LEARNING_RATE)
```

Separate optimizers ensure that safety-related and action-related parameters are updated independently.

Training Process

The training loop iteratively updates the parameters of the CBF and action networks. Each iteration involves:

1. **Data Loading:** Randomly generating states and goals using the *generate data* function.
2. **Forward Pass:** Computing:
    - Control actions u using the action network.
    - Safety measures $h(s)$ using the CBF network.
    - Losses for safety ($h(s)$), dynamics ($\dot{h}(s)$), and task performance (u).
3. **Backward Pass:** Accumulating gradients over multiple inner loops before updating parameters.
4. **Parameter Updates:** Alternating between optimizing the CBF and action networks.

Code Example: Forward Pass and Loss Computation

```
for i in range(accumulation_steps):
    a_np = model_action(s_np, g_np)
    s_np = s_np + torch.cat([s_np[:, 2:], a_np], dim=1) *
       config.TIME_STEP
    h, mask, indices = model_cbf(s_np, config.DIST_MIN_THRES)
    loss_action = core.loss_actions(s=s_np, g=g_np, a=a_np,
       r=config.DIST_MIN_THRES)
```

Model Saving and Monitoring

The model is periodically saved to checkpoints, enabling resumption of training and model evaluation. Training progress is logged, including:

- Average loss across safety and action objectives.
- Accuracy metrics for safety conditions ($h(s) \geq 0$, $\dot{h}(s) \geq -\alpha h(s)$).

Code Example:

```
if (istep + 1) % config.SAVE_STEPS == 0:
    torch.save({
        'model_cbf_state_dict': model_cbf.state_dict(),
        'model_action_state_dict': model_action.state_dict(),
    }, os.path.join('models', f'model_iter_{istep + 1}.pth'))
```

This ensures that the model's state can be easily restored for further training or evaluation.

Summary of Training Pipeline

The `train.py` script implements the training pipeline for learning safe and efficient decentralized control policies. It:

- Leverages PyTorch's data handling and optimization capabilities for modular and scalable training.
- Alternates between optimizing the Control Barrier Function and action networks.
- Logs and monitors training progress to ensure safety and task performance goals are met.
- Saves model checkpoints for reproducibility and evaluation.

This design ensures an efficient and reproducible learning process for multi-agent systems.

Evaluate.py

Action Refinement via Control Barrier Functions

The function `get_next` refines nominal actions produced by the action network to ensure safety using the CBF network. This process optimizes the input actions by iteratively adjusting them to satisfy safety constraints.
Steps in Action Refinement:

1. Compute CBF values h for the current state s.
2. Iteratively refine the action a using gradient descent to minimize errors in the CBF conditions.
3. After optimization, apply the refined action to compute the next state s_{next}.

Optimization Objective: The refinement process minimizes violations of the CBF condition:

$$\text{Error} = \sum \max(-\text{Derivative Condition}, 0),$$

where the derivative condition is:

$$\dot{h}(s, a) + \alpha h(s) \geq 0.$$

Code Snippet: Refining Actions with CBF

```
a_res = torch.zeros_like(a, requires_grad=True).to(device)
optimizer = torch.optim.SGD([a_res],
   lr=config.REFINE_LEARNING_RATE)

for _ in range(config.REFINE_LOOPS):
    optimizer.zero_grad()
    dsdt = core.dynamics(s, a + a_res)
    s_next = s + dsdt * config.TIME_STEP
    h_next, _, _ = model_cbf(x=s_next.unsqueeze(1) -
       s_next.unsqueeze(0))
    deriv = h_next - h + config.TIME_STEP * config.ALPHA_CBF * h
    error = torch.sum(torch.relu(-deriv))
    error.backward()
    optimizer.step()
```

Output of Refinement:

- a_{opt}: Refined action satisfying CBF conditions.
- s_{next}: Next state computed using the refined action.
- h_{next}: Updated CBF values after applying the refined action.

Evaluation Loop

The script evaluates the model over multiple steps, tracking metrics such as safety ratios and distance errors. Each evaluation step involves:

1. **Data Generation:** Randomly initialize agent states and goals.
2. **Action Computation:** Compute nominal actions using the trained action network.
3. **Action Refinement:** Refine actions using the CBF network to ensure safety.
4. **Metrics Computation:** Compute safety ratios, distance errors, and rewards for both the refined model and a baseline LQR controller.
5. **Visualization:** Render trajectories of agents for qualitative analysis.

Code Snippet:

```
with torch.no_grad():
    a_network = model_action_eval(s_np, g_np)  # Nominal action
h_next, s_next, a_opt, _, _ = get_next(s_np, g_np, a_network,
   model_cbf_eval, device)
```

Metrics

The following metrics are computed to assess performance:

- **Safety Ratio:** Proportion of agents avoiding collisions during their trajectories.
- **Distance Error:** Average distance between agent states and their goals.
- **Rewards:**
  - **Safety Reward:** Penalizes unsafe actions based on the number of collisions.
  - **Distance Reward:** Rewards agents for reaching their goals within a threshold distance.

Code Snippet: Safety Ratio Computation

```
safety_mask_np = core.ttc_dangerous_mask_np(
    s_np.cpu().numpy(), config.DIST_MIN_CHECK,
       config.TIME_TO_COLLISION_CHECK)
safety_ratio = 1 - np.mean(safety_mask_np, axis=1)
```

Summary of Evaluation Pipeline

The `evaluate.py` script evaluates the trained CBF and action networks, focusing on safety and task performance. Key features include:

- Refining nominal actions using the CBF network to ensure safety.
- Comparing safety and task metrics between the trained model and a baseline LQR controller.
- Visualizing agent trajectories to assess performance qualitatively.

This evaluation pipeline provides both quantitative and qualitative insights into the model's ability to navigate multi-agent systems safely and efficiently.

Polygon-Based Agent Initialization and Goal Assignment

To evaluate the decentralized control framework in structured environments, a new initialization strategy was implemented to place agents and goals on the perimeters of regular polygons. These polygons, with 3, 4, or 6 sides (triangles, squares, and hexagons), introduced controlled spatial configurations, making the evaluation more structured and reproducible.

Key Features

The initialization process ensures:

- **Geometric Arrangement:** Agents and goals are placed on the perimeters of regular polygons, creating structured spatial interactions.
- **Minimum Distance Threshold:** A distance threshold (`dist_min_thres`) is enforced to avoid overlap between agents and their goals.
- **Dynamic Polygon Radius:** The radius of each polygon is computed dynamically based on the number of agents and the distance threshold, ensuring sufficient spacing.

Initialization Procedure

The initialization process comprises the following steps:

1. **Select Polygon:** Randomly choose a regular polygon with 3, 4, or 6 sides.
2. **Compute Radius:** Determine the radius required to fit all agents and goals along the perimeter while respecting the minimum distance threshold.
3. **Generate Perimeter Points:** Compute evenly spaced points along the polygon's perimeter using linear interpolation between vertices.
4. **Assign Positions:** Randomly assign agents and goals to non-overlapping positions along the perimeter, ensuring sufficient separation between them.

Pseudocode

The pseudocode for generating agent positions and goals is as follows:

```
function generate_data_polygons(num_agents, dist_min_thres):
    sides = random_choice([3, 4, 6])  # Triangle, Square, Hexagon
    radius = compute_polygon_radius(num_agents, dist_min_thres, sides)
    positions = generate_perimeter_points(sides, radius)

    agent_positions = randomly_assign(num_agents, positions)
    goal_positions = assign_remaining_positions(num_agents, positions, agent_position

    states = combine_positions_and_zero_velocities(agent_positions)
    return states, goal_positions
```

In this pseudocode:

- `compute_polygon_radius`: Calculates the required radius to fit all agents and goals while respecting the minimum distance threshold.
- `generate_perimeter_points`: Computes evenly spaced points along the polygon's perimeter.
- `randomly_assign` and `assign_remaining_positions`: Randomly select positions for agents and goals while ensuring sufficient separation.
- `combine_positions_and_zero_velocities`: Combines agent positions with zero initial velocities to form the initial state array.

Evaluation Benefits

This configuration provides several advantages:

- **Structured Testing:** Evaluates the framework in well-defined geometric settings, enhancing reproducibility.
- **Challenging Interactions:** Creates structured interactions that test collision avoidance under constrained trajectories.
- **Scalability:** Easily scales to varying numbers of agents and polygons of different sizes.

This initialization strategy strengthens the evaluation pipeline by introducing structured and challenging scenarios for assessing the learned control policies and CBFs.

Results

The performance of the decentralized control framework was evaluated under various configurations. This section presents the results, highlighting the system's ability to ensure safety and task completion in structured environments.

Evaluation Configurations

The following configurations were tested:

- **Polygon-Based Initialization:** Agents and goals placed on the perimeters of polygons (triangle, square, hexagon).
- **Random Initialization:** Agents and goals are placed randomly in the workspace.

Polygon-Based Configurations

The framework was tested with agents initialized on polygons with 3, 4, and 6 sides. Screenshots of trajectories are presented below:

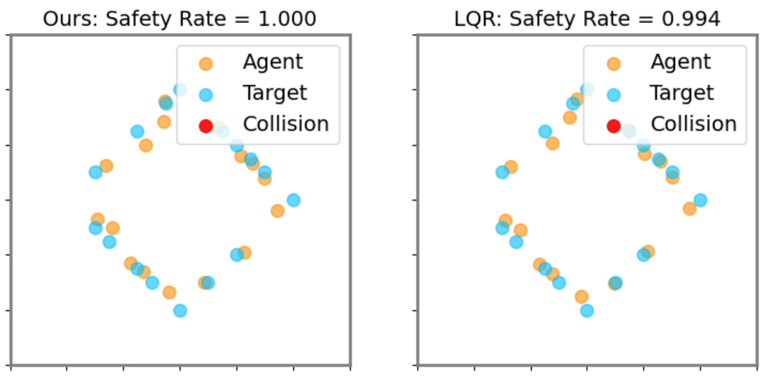

Figure 2: Initialization of agents and goals on Square perimeter

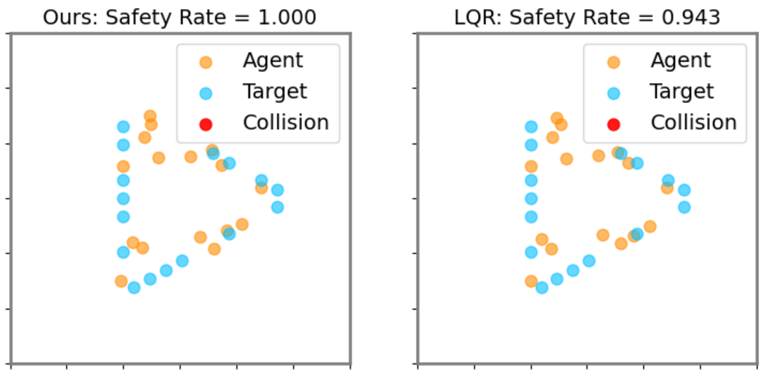

Figure 3: Initialization of agents and goals on Triangular perimeter

In these configurations:

- **Safety:** All agents avoided collisions throughout their trajectories with a Safety Rate=1, however increasing the number of agents decreased the safety slightly.
- **Goal Completion:** Agents successfully reached their assigned goals with error = 0.0255 units in 3*3 frame.
- **Trajectory Efficiency:** The learned policies ensured smooth and direct paths to the goals.

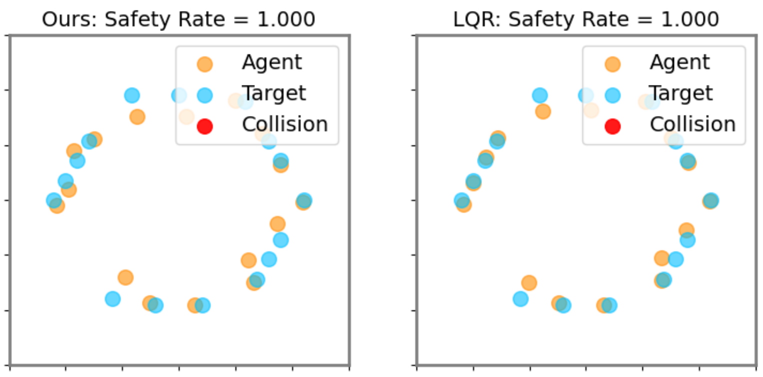

Figure 4: Initialization of agents and goals on hexagonal perimeter

Random Initialization

Agents were initialized in random positions within the workspace. The results demonstrate the framework's ability to generalize to unstructured environments.

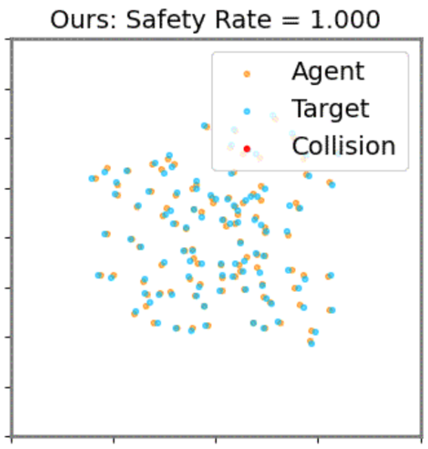

Figure 5: Random initialization of 100 agents and goals.

Key observations include:

- **Collision Avoidance:** The framework maintained safety despite close proximity interactions.
- **Dynamic Adaptability:** The learned policies adapted effectively to diverse configurations.

Performance Metrics

The following metrics summarize the system's performance across all configurations:

- **Safety Ratio:** 99% across all configurations however increasing the number of agents in polygon configuration decreased it, which can be improved in future work.
.

Parameter Tuning

To ensure optimal performance across different configurations, several parameters were tuned to adapt to the specific challenges posed by each scenario. These configurations include polygonal setups (triangle, square, hexagon) and symmetric random placements. Key parameters adjusted include:

- **Observation Radius (`obs_radius`):** Determines the range within which agents perceive nearby obstacles or other agents. For configurations with densely packed agents (e.g., smaller polygons), a smaller observation radius was used to focus on immediate interactions. Conversely, in larger or sparse setups, the radius was increased to account for agents further away.
- **CBF Coefficient (`alpha_cbf`):** Governs the rate at which the Control Barrier Function enforces safety. In symmetric configurations with agents starting closer to their goals, a higher `alpha_cbf` ensured rapid responses to potential collisions. For larger polygons with more spread-out agents, a lower value was sufficient to maintain safety while allowing smoother trajectories.
- **Time-to-Collision Threshold (`ttc`):** Used to determine the urgency of collision avoidance maneuvers. For dense polygonal setups, a lower threshold was used to ensure timely responses.

References

1. Qin, X., Chen, X., Li, X., Sui, X., Kumar, A. (2021). *Learning Safe Multi-Agent Control with Decentralized Neural Barrier Certificates.* Proceedings of the International Conference on Learning Representations (ICLR 2021). `https://arxiv.org/abs/2101.05436`

Note: The complete implementation and additional resources for this project are available on GitHub: https://github.com/Raj9571/Pytorch-Multiagent$_{C}BF.git$.