# HiSaRL: A Hierarchical Framework for Safe Reinforcement Learning

**Zikang Xiong, Ishika Agarwal, Suresh Jagannathan**

Computer Science Department, Purdue University,
West Lafayette, Indiana 47906
xiong84@cs.purdue.edu, agarwali@purdue.edu, suresh@cs.purdue.edu,

**Abstract**

We propose a two-level hierarchical framework for safe reinforcement learning in a complex environment. The high-level part is an adaptive planner, which aims at learning and generating safe and efficient paths for tasks with imperfect map information. The lower-level part contains a learning-based controller and its corresponding neural Lyapunov function, which characterizes the controller's stability property. This learned neural Lyapunov function serves two purposes. First, it will be part of the high-level heuristic for our planning algorithm. Second, it acts as a part of a runtime shield to guard the safety of the whole system. We use a robot navigation example to demonstrate that our framework can operate efficiently and safely in complex environments, even under adversarial attacks.

## 1 Introduction

Although deep reinforcement learning has achieved promising results in various domains, ensuring its safety still is a concern. One line of work (Bastani, Pu, and Solar-Lezama 2018; Zhu et al. 2019; Chang, Roohi, and Gao 2020; Dai et al. 2021) provides rigorous safety guarantees by using analytic approaches. These proposals generally require knowledge of a system's underlying dynamics and constraints, making it challenging to generalize their method to handle complex dynamics. On the other hand, hierarchical reinforcement learning algorithms (Levy, Jr., and Saenko 2017; Nachum et al. 2019; Kreidieh et al. 2019) are attractive because they can support complex tasks without requiring knowledge of an underlying environment structure. However, this lack of knowledge and the non-stationary MDP problem make these algorithms be time-intensive without providing any assurance of safety after training.

In contrast, our framework assumes, in the specific planning setting we consider, that our learning algorithm can assess the map information of one environment. Such information can be a priori modeled with a high-definition map or collected during runtime using techniques such as SLAM. With map information, a high-level planner can generate a safe and efficient path. This planner frees our agents from unsafe and inefficient exploration for generating a high-level navigation policy. However, a safe *plan* cannot guarantee safety at *runtime*. Because the low-level controller is not required to follow the plan perfectly, agents can deviate from a safe plan and produce unsafe behaviors, such as hitting obstacles. To solve this problem, we apply a learned neural Lyapunov function (Berkenkamp et al. 2016; Chang, Roohi, and Gao 2020) as a runtime monitor and harness its stability property to enforce that the agent stays in a region specified by the function. We also incorporate the neural Lyapunov function as part of our planner heuristic. This incorporation fuses our high-level planner and the low-level controller seamlessly. Moreover, our learned neural Lyapunov function does not require any knowledge about system dynamics. Hence, compared with (Bastani, Pu, and Solar-Lezama 2018; Zhu et al. 2019), our approach can extend to complex unknown dynamics.

The backend algorithms of our planner are A$^*$ (Hart, Nilsson, and Raphael 1972) and RRT$^*$ (Urmson and Simmons 2003). When we have accurate map information, both A$^*$ and RRT$^*$ work correctly. However, if map information is inaccurate or outdated, our backend algorithm is unlikely to work as expected. Hence, we further strengthen our planner with a refinement policy. The refinement policy's mission is repairing and refining planning decisions during runtime. Inspired by the elastic band algorithm (Quinlan and Khatib 1993), we propose a learning-based approach to generate the refinement policy.

It is well-known that most deep learning algorithms suffer from robustness issues. Adversarial attacks (Sun et al. 2018; Mankowitz et al. 2020; Chen et al. 2019; Zhang et al. 2020) can effectively test the robustness of a learning-enabled system. Hence, we formulate an attack mechanism against our framework to force an agent (i.e, robot) to deviate outside the region specified by the Lyapunov function. Our results show that unless we apply an unrealistically high attack frequency and force significant perturbation, our framework is robust and can keep the robot within a safe region.

The contributions of this work can be summarized as follows:

- We propose a hierarchical framework that reconciles both efficiency and safety. Our framework can enhance the safety of an agent in complex environments, where the dynamics of the controlled robots are unknown, and the environment information (i.e., map and obstacle in-

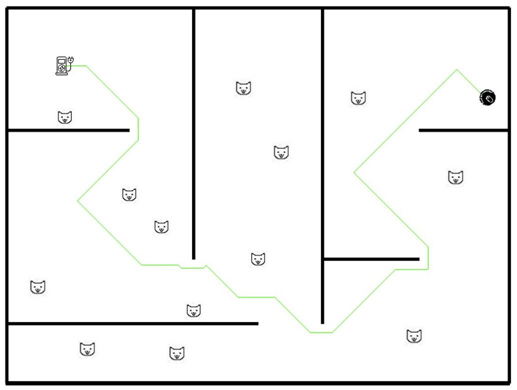


Figure 1: Sweeping robot in room example. We need to control the sweeping robot to reach the charger without hitting cats and walls. The green lines are planed path.

formation) can be imperfect.

- We consider an approach to adapt the high-level planner to deal with imperfect information.
- We demonstrate the robustness of our framework under adversarial attack.

## 2 Motivation Example

### 2.1 Hierarchical Framework

The simple navigation task in Figure 1 requires navigating the sweeping robot from the initial position $s_0$ to the goal position $g_T$ (the charger). We solve this problem with a two-level hierarchical framework. First, a high-level planner finds a safe plan (i.e., no collision with walls and cats) from the initial position $s_0$ to the goal position $g_T$. The plan is a sequence of subgoals shown as the green line in Figure 1. We denote it as $P(s_0, g_T) = (g_0, g_1, \dots, g_T)$, where $g_i$ is a subgoal, $g_0 = s_0$. Then, A low-level controller $\pi_l(s|g)$ is introduced to execute the plan. The low-level controller is conditioned by a subgoal $g$, and it is trained to predict the optimal action under state $s$ to reach $g$. The low-level controller is trained with TD3 (Fujimoto, Hoof, and Meger 2018) using a reward for achieving the given subgoal $g$ with the shortest path.

### 2.2 Runtime Safety

Although the high-level planner can provide a safe plan, the low-level controller does not necessarily follow the plan strictly. Especially when we train the low-level controller with a model-free reinforcement learning algorithm, it is quite common that the agent finds an unexpected approach to achieve the goal. Once the unstable low-level controller leads our robot to deviate from our safety plan, we cannot guarantee the safety of our robot.

**Stability of Low-level Controller** We measure the stability of a low-level controller with the deviation between the actual trajectory and the plan. In Figure 2, the robot's position at time $t$ is $pos_t$, the deviation $d_t$ is defined as the Euclidean distance from $pos_t$ to the plan fragment between 2 subgoals.

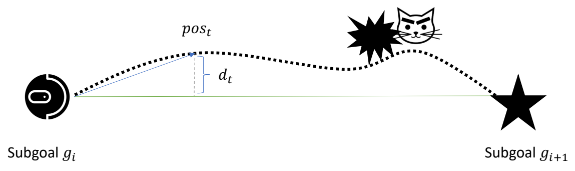


Figure 2: A demonstration on how the low-level controller executes a plan. The star is a subgoal, and the green line is the plan. The low-level controller may drift from the plan and generate the dotted path. Although our plan avoids hitting the cat, the low-level controller still cannot ensure safety during the runtime.

**Lyapunov Function** The large deviation $d_t$ can cause unsafe behaviors. Hence, we hope to constrain the robot around our plan. The Lyapunov function is a positive-definite function for analyzing the stability of a system. A Lyapunov function $V(x)$ characterizes a control system's Region Of Attraction (ROA). The ROA binds the robot to stay around the plan. We will introduce the technical details for the neural Lyapunov in Section 3.2.

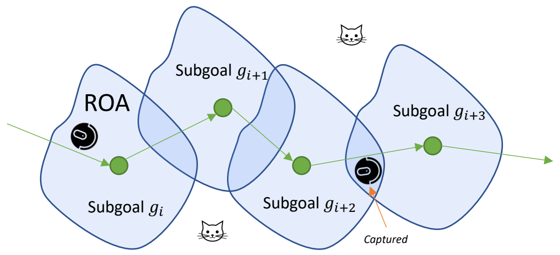
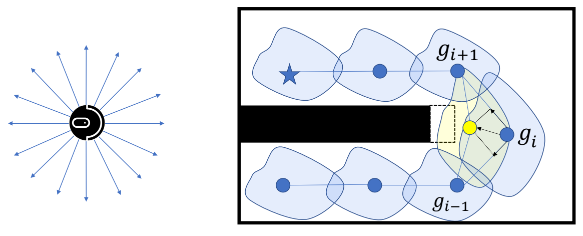


Figure 3: The way to apply the Lyapunov function. We built a sequence of ROAs around the subgoals. One ROA is a region that the robot will stay in when the robot is heading to the corresponding subgoal (i.e., the sink of the ROA). Each ROA has intersections with its neighbors.

**Runtime Shield** With the plan and ROAs built by the Lyapunov function, we can construct a runtime shield to guard the safety of our robot. Unlike the previous method (Bastani, Pu, and Solar-Lezama 2018; Zhu et al. 2019), we do not require an additional safe policy for the recovering purpose. Instead, we construct the runtime shield by switching the robot's heading toward different subgoals. Specifically, we select two consecutive subgoals during the runtime. The first one is the latest subgoals achieved; the second one is the next subgoals. Then, we select the first subgoal's ROA to monitor our robot. While the robot stays in the selected ROA, we set its subgoal to the second subgoal that we have not achieved yet. When the robot goes beyond the selected ROA, we check whether it is "captured" by the next ROA (i.e., the robot enters the intersection part of two ROA, see Figure 3). If the robot is captured, we will change the selected ROA to the next ROA until the robot achieves the second subgoal. Nevertheless, if the robot is not captured by the next ROA and slides out of the selected ROA, we will set its subgoal to the first subgoal to pull it back. The runtime shield binds a robot to stay around its plan. We can consider the ROA while planning and generating a plan with safety boundaries. Consequently, the runtime safety of our system is guarded. A more formal description is provided in Section 3.3.

## 2.3 Planner

The backend algorithm of our planner is A$^*$ and RRT$^*$. Both of them require a heuristic guiding the efficient search. The most straightforward heuristic might be the Euclidean distance. However, this heuristic cannot guarantee that the generated path is safe for our system. For example, a planning path may get close to the wall, which does not leave enough space for the shield to switch the subgoal for avoidance. Thus, we have to consider the Lyapunov function as part of the heuristic. We further consider scenarios that we do not have the perfect map data. In this case, we need to collect the sensor data during the runtime and refine our plan. We achieved the refinement by employing a high-level reinforcement learning policy. This refinement policy enables our system to operate safely with imperfect map data.



(a) Robot sensors (b) Refinement in U-maze environment

Figure 4: (a) demonstrates the sensors on the robot. The LiDARs scans 16 positions around, which generates an observation vector with 16 distance elements. (b) is an example for refinement policy. The dashed region does not exist in real word, but the map used for planning marks it as obstacle.

**Heuristic** A simple Euclidean distance heuristic is the $L_2$ distance from current subgoal position to the final goal position, $h_{euc}(g_t) = ||g_T - g_t||$. This heuristic guides the search toward the final goal. For the U-maze example in Figure 4(b), the path generated by $h_{euc}(g_t)$ will stick to the wall in the lower part because this path is closer to the final goal. However, such a path does not provide any space as a safety distance. Thus, we also need to consider a Lyapunov heuristic $h_{lyap}(g_t)$. Given $x_t$, the relative position from the sink of ROA $g_t$ to the closest obstacle, we define the $h_{lyap}(x_t)$ to characterize the safety. If the closest obstacle is located outside the ROA, $h_{lyap}(x_t)$ returns 0. Otherwise, $h_{lyap}(x_t)$ returns $-\infty$ to disable the search on this path. We compose the Euclidean heuristic for efficiency and the Lyapunov heuristic for safety, and guide the planning search with $h_{euc}(g_t) + h_{lyap}(x_t)$.

**Refinement Policy** Naturally, the map will change after the data is collected. Hence, the plan made in the outdated map may cause undesired results. Thus, we consider a repair schema during runtime. The key idea is the elastic band (Quinlan and Khatib 1993) which optimizes a planning "string" with the internal and repulsive force. The internal force, formally, can be described as

$$F_{in} = k_{in} \cdot \left( \frac{g_{i-1} - g_i}{\|g_{i-1} - g_i\|} + \frac{g_{i+1} - g_i}{\|g_{i+1} - g_i\|} \right)$$

where $k_{in}$ is the elasticity coefficient. The internal force for subgoal $g_i$ is computed with the sum of the two direction vectors toward its neighbors. In addition to the internal force, a repulsive force is applied to prevent the ROA from intersecting with obstacles. When we have perfect map data, we can compute the repulsive force easily. However, it can be a problem when the map data is imperfect. In this case, we only have the raw sensor data during runtime. To address this problem, we learn a function $k_{re}(g_i|h_i)$ to predict the repulsive coefficient, given the subgoal $g_i$ and a sequence of sensor data history $h_i$. Finally, the applied force $\Delta f$ for subgoal $g_i$ is

$$\Delta F = (k_{in} - k_{re}(g_i|h_i)) \cdot \left( \frac{g_{i-1} - g_i}{\|g_{i-1} - g_i\|} + \frac{g_{i+1} - g_i}{\|g_{i+1} - g_i\|} \right)$$

We can iteratively update the $g_i$ with $g_i' = g_i + \alpha \Delta F$ to find the equilibrium where $\Delta F = 0$. $\alpha$ here is a small constant. An example is provided in Figure 4(b). The blue $g_i$ was updated to the yellow position with iteratively applying $\Delta F$.

## 2.4 Adversarial Attack

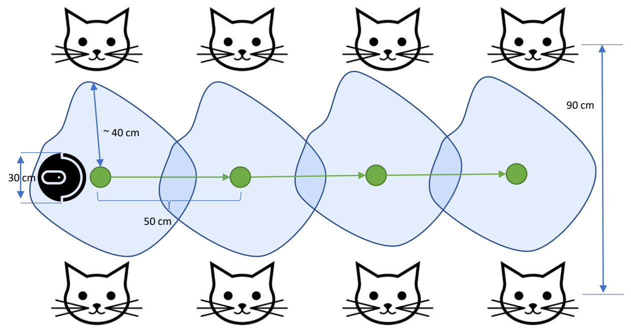


Figure 5: Cat parade environment for robustness evaluation

To better evaluate the robustness of our framework, we consider attacking the framework from two aspects. Firstly, the neural network is criticized for being unrobust to the input perturbations. Thus, we add adversarial noise to the low-level controller and the neural Lyapunov function's inputs in the first type of attack. Secondly, the robustness of a system is not only affected by the upstream perception data fed to the neural network controller, but also the actual action executed by downstream modules. That means even if the controller outputs the right action, the noise in the downstream models can still cause undesired results. Hence, we further attack our framework with noise added to the action.

We evaluate the robustness of our system with this simple but safety-sensitive cat parade environment in Figure 5. The adversary needs to fool our neural Lyapunov function and guide the robot to hit the cats. We assume that we can access the parameters of the neural Lyapunov function and the low-level controller. Because our framework aims to work on the robot with complex dynamics, it is generally challenging to model the robot's dynamics required by a gradient-based attack such as FGSM (Goodfellow, Shlens, and Szegedy 2014) and PGD (Madry et al. 2017). Hence, similar to (Zhao et al. 2020), we learn approximated surrogate dynamics. We provide the technical details in Section 3.5.

## 3 Approach

Our work incorporated a Lyapunov function into a learning-enabled control system. First, because it is challenging to reason about the behavior of a RL-trained controller, we use the Lyapunov function to characterize its stability property. The Lyapunov function provides us with the power to bind a robot to a specified region. Second, we assemble the specified ROA with a high-level planner. By considering the Lyapunov function and the planning heuristic jointly, we can generate a safety plan. When executing the plan, a safety shield guards the controlled robot. Additionally, we enhance our high-level planner with a learning-based high-level policy. This policy refines plans with the sensor data collected during runtime, which gives our high-level planner the ability to work on imperfect map data. Finally, we evaluate the robustness of our system against adversarial attacks.

### 3.1 Low-Level Controller

A low-level controller $\pi_l(s|g)$ is introduced to achieve a subgoal $g$. The controller is trained with TD3. We designed its reward function as

$$r(s_t) = ||s_{t-1} - g_{t-1}|| - ||s_t - g_t||$$

We compute the distance-to-goal at the previous time step $t-1$ and the current time step $t$. The reward function is the difference between the two distances. TD3 maximizes this reward w.r.t the Bellman function. As a result, the trained agent is expected to get closer to the given goal in the fastest direction.

### 3.2 Neural Lyapunov Function and ROA

We use the neural Lyapunov function to characterize the stability property. The Lyapunov function is a positive-definite function satisfying three constraints.

$$V(x_o) = 0 \quad (1)$$

$$\forall x \neq x_0, V(x) > 0 \quad (2)$$

$$V(x_{t+1}) - V(x_t) < 0 \quad (3)$$

In Eq. (1), a Lyapunov function's value is 0 on the origin. [1] Eq. (2) is the position property of a Lyapunov function. Eq. (3) is known as the lie derivative. When the lie derivative is smaller than 0, $V(x_t)$ strictly decreases along with time. We compute the Lyapunov function with a neural network and train the neural network with a loss function based on the Zubov-type equation (Grune and Wurth 2000). The loss function is

$$\mathcal{L}_\theta(x_t, x_{t+1}) = |V_\theta(x_0)| + \left|V_\theta(x_t)(V_\theta(x_{t+1}) - V_\theta(x_t)) - ||x_t||^2\right|$$

The Monte-Carlo estimation of $\mathcal{L}$ is

$$L_\theta = \frac{1}{N}\Big(\sum_{x_t, x_{t+1} \sim \pi_l} |V_\theta(x_0)| + \left|V_\theta(x_t)(V_\theta(x_{t+1}) - V_\theta(x_t)) - ||x_t||^2\right|\Big)$$

To characterize the stability of $\pi_l$, $x_t$ and $x_{t+1}$ are sampled with the $\pi_l$. We can compute the gradient with $L$ and optimize the network's parameters $\theta$. The optimal parameters are $\theta^* = \arg\min_\theta(L_\theta)$.

The neural Lyapunov function specifies the ROA with a constant $C_{ROA}$.

$$\text{ROA} = \{g + x | V(x) < C_{ROA}\}$$

Where $g$ is the sink of a ROA.

### 3.3 Runtime Shield

The Lyapunov function specifies a single ROA that a robot stays in. However, a runtime shield binds a robot to stay around a given plan $P(s_0, g_T) = (g_0, g_1, \ldots, g_T)$. To deal with the sequence of subgoals, we check the ROAs of any two consecutive subgoals. Given the lower-level controller $\pi_l$, previous system state $s_{i-1}$, current system state $s_i$, ROA$_t$ with sink $g_t$, and ROA$_{t+1}$ with sink $g_{t+1}$, the algorithm 1 shows the details of the runtime shield.

Algorithm 1: Sequential Shield

```
function SHIELD($\pi_l, s_{i-1}, s_i, g_t, g_{t+1}$, ROA$_t$, ROA$_{t+1}$)
    if $s_i \in$ ROA$_t \vee s_i \in$ ROA$_{t+1}$ then
        return $\pi_l(s_i|g_{t+1})$
    else if $(s_{i-1}) \in$ ROA$_t \wedge s_i \notin$ ROA$_{t+1}$ then
        return $\pi_l(s_i|g_t)$                ▷ Pull back to ROA$_t$
    else
        return $\pi_l(s_i|g_{t+1})$
```

When $s_i \in$ ROA$_t \vee s_i \in$ ROA$_{t+1}$, the $\pi_i$ is parameterized with the next subgoal $g_{t+1}$, and the robot moves toward $g_{t+1}$. When $(s_{i-1}) \in$ ROA$_t \wedge s_i \notin$ ROA$_{t+1}$, the robot moves from the ROA$_t$ to a region other than ROA$_{t+1}$. We need to pull the robot back to ROA$_t$, thus the $\pi_i$ is parameterized by $g_t$. The last condition is $(s_{i-1}) \in$ ROA$_{t+1} \wedge s_i \notin$ ROA$_{t+1}$, although this is not supposed to happen. If it happens due to any aspect of imprecision, we return action $\pi_i(s_i|g_{t+1})$.

[1]The Lyapunov function's input $x$ and state $s$ are in the same space. The only difference is their coordination origin. The origin of $s$ is determined by the system. The origin of $x$, however, is a selected subgoal during the runtime. $x = s - g$.

## 3.4 Planner

**Heuristic** Our planning heuristic has two parts. The $h_{euc}$ for efficiency and the $h_{lyap}$ for safety. $h_{euc}$ is the Euclidean distance to the final goal.

$$h_{euc}(g_t) = ||g_T - g_t||$$

This heuristic is designed to search for the shortest path, but does not consider safety. Hence, it can generate paths that are close to obstacles, which may cause undesired behaviors during the runtime. To address this problem, we introduce a Lyapunov heuristic.

$$h_{lyap}(x_t) = \begin{cases} 0 & V(x_t) > C_{ROA}, \\ -\infty & V(x_t) \leq C_{ROA} \end{cases}$$

Suppose the closest obstacle position to $g_t$ is $o_t$, $x_t = o_t - g_t$; $C_{ROA}$ is the constant specified the ROA. This heuristic ensures that the ROAs do not intersect with obstacles.

**Refinement Policy** We provided an example for the refinement policy in Section 2.3. Force $\Delta F$ was applied to update the subgoal $g_i$ of a path.

$$\Delta F = (k_{in} - k_{re}(g_i|h_i)) \cdot \left( \frac{g_{i-1} - g_i}{\|g_{i-1} - g_i\|} + \frac{g_{i+1} - g_i}{\|g_{i+1} - g_i\|} \right)$$

The updating is iterative,

$$g_i' = g_i + \alpha \cdot \Delta F$$

The new subgoal $g_i'$ should converge to a fixed point where $k_{in} = k_{re}(g_i'|h_i)$. The $k_{re}(g_i'|h_i)$ was learned using an encoder-decoder structure. First, we train an auto-encoder $ENC$ to encode the history. Then, we concatenate the embedding $ENC(h_i)$ with subgoal $g_i$, and train an MLP to predict the $k_{re}$. The training data was sampled with simulations in different scenarios.

## 3.5 Robustness to Adversarial Attack

**Surrogate Dynamics** The surrogate dynamics is a function

$$f_{dyn}(s_t, a_t|h_t) = s_{t+1}$$

where the $s_t$ is the state and $a_t$ is the action. $h_t$ is the history states of the agent. If a system is fully-observable, we can ignore the $h_t$. The $f_{dyn}$ computes the next state of the system.

**Attack Controller** We define an objective function $f_{obj}$ that measures the distance to the planning path. Maximizing the $f_{obj}(s_{t+1})$ guides the robot to deviate from the plan. We considered a simple FGSM attack. When attacking the input state of the low-level controller, we can compute the attacked state $\hat{s}_t$ with

$$\hat{s}_t = s_t + \varepsilon \cdot \mathtt{sign}(\frac{\partial f_{obj}\left(f_{dyn}(s_t, a_t|h_t)\right)}{\partial s_t})$$

On the other hand, if we want to attack the action, the attacked action $\hat{a}_t$ can be computed with

$$\hat{a}_t = a_t + \varepsilon \cdot \mathtt{sign}(\frac{\partial f_{obj}\left(f_{dyn}(s_t, a_t|h_t)\right)}{\partial a_t})$$

where $\varepsilon$ is the attack noise size. Because we can compute the gradient of $s_t$ and $a_t$, we can also apply PGD or other attack techniques similarly.

**Attack Neural Lyapunov Function** We want to fool the neural Lyapunov function with an attacked input $\hat{x}_t$. In this case, we expect a small $V(\hat{x}_t)$, where $V$ is the Lyapunov function. The attacked input $\hat{x}_t$ can be computed with gradient $\frac{dV(x_t)}{dx_t}$.

$$\hat{x}_t = x_t + \varepsilon \cdot \mathtt{sign}(\frac{dV(x_t)}{dx_t})$$

Note that we optimize the attacked input $\hat{x}_t$ of the Lyapunov function and the attacked state $\hat{s}_t$ of the low-level controller separately. That makes the attack more potent.

## 4 Primary Evaluation

In this section, we report the primary evaluation results on our motivation examples introduced in Section 2.

### 4.1 Low-level controller

We train the low-level controller $\pi_l(s|g)$ with TD3. The training results are provided in Figure 6. Each training iteration indicates updates on the low-level controller. The updates are executed periodically in every 2000 simulation steps. All the reward curves in Figure 6 converge to a reward around -2.5.

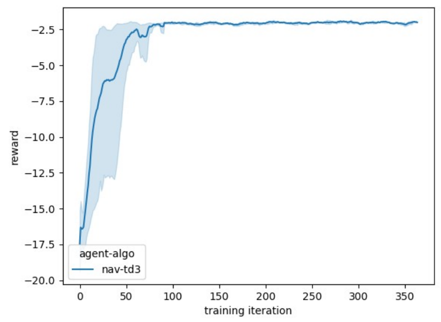


Figure 6: Training reward of $\pi_l(s|g)$. This figure includes results of 5 runs with different random seeds.

### 4.2 Neural Lyapunov Function

We train the neural Lyapunov function with a dataset containing $10^6$ transitions and test the neural Lyapunov with a test dataset with $2 \times 10^5$ transitions. Both the training and test dataset are sampled with the same low-level controller. During the test time, we check the properties in Eq. (1), Eq. (2) and Eq. (3) on the test dataset with five neural Lyapunov functions trained with different splits on the training and testing dataset.

The statistics are in Table 1. The minimum percentage of property violations is 0%, and the maximum is 0.3%. Thus, out of around 10,000 simulations, only 30 simulations are violated.

| Stat. | min | max | mean | std |
|---|---|---|---|---|
| Vio. | 0 | 0.00294 | 0.000656 | 0.00115 |

Table 1: Neural Lyapunov function training results.

## 4.3 Refinement Policy

We generated the sensor data and computed the accurate repulsive constant for training repulsive prediction network $k_{re}(g_i|h_i)$ in different scenarios (i.e., the obstacles appear in different positions). $10,000$ trajectories are generated, which includes $9,500$ training trajectories and $500$ test trajectories. Each trajectory contains 5 seconds of sensor data. However, our repulsive prediction network only uses latest 1 seconds sensor data as the history $h_i$ in $k_{re}(g_i|h_i)$. The range of the $k_{re}$ is $[-1, 1]$, the prediction has error around 0.01. In our toy example provided in Figure 4, we measure the average distance between plan and center obstacle. We hope this distance be small while the ROA should not collide with the obstacle. Before applying our refinement policy, the average distance is $51.32$. After 100 runs and refinements, our refinement policy changed the average distance to $45.27$ and avoided the collision between obstacles and ROAs.

## 4.4 Robustness to Adversarial Attack

We attacked both the state and action with different attack frequency and noise size $\varepsilon$. The Figure 7 and 8 provide the results of attacks on the state and action respectively. Each column is generated with 100 experiments with attacks at random times. The attack frequency ranges from 0.2 to 1.0, indicating the percentage of attacked transitions. The $^*$ in the legend means that we also attack the Lyapunov function in these experiments and the unit of $\varepsilon$ is cm. The $y-$axis is the max deviation to our plan. When the $d_{max} > 30$ cm, the robot will hit cats in the example provided in Figure 5.

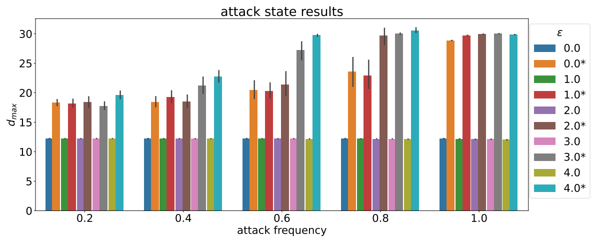


Figure 7: Attack state results

In Figure 7, when we do not attack the Lyapunov function, the $d_{max}$ is always bounded below 15 cm. Hence, the protection provided by the shield works as expected. Nevertheless, when the Lyapunov function is under attack, we notice that all the $d_{max}$ is significantly larger, which means the Lyapunov function and shield can be affected by the attack. On the other hand, the $d_{max}$ grows as the attack frequency and $\epsilon$ increases. The first safety violation happens when the attack frequency is $60\%$, and the noise size is $4.0$ cm, while we also attack the Lyapunov function.

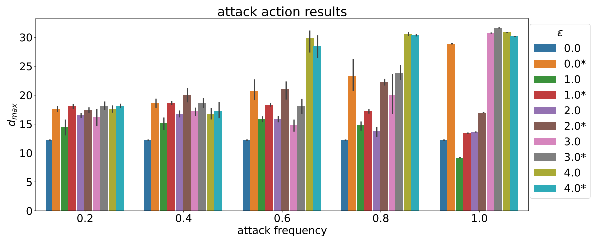


Figure 8: Attack action results

Figure 8 provides the attack action results. Our robot's action range is $[-5, 5]$. The action represents the displacement every 0.1 seconds. We set the attack noise from 0 to 4. The $d_{max}$ is smaller when we do not attack the Lyapunov function. Overall, the $d_{max}$ increases as the $\varepsilon$ and attack frequency raises. We also observe that sometimes larger $\varepsilon$ results in smaller $d_{max}$. This is because a stronger attack sometimes can cause intensive calling on the shield's pull-back action. For example, when the attack frequency is 1.0 when the $\varepsilon = 1.0$, the $d_{max}$ is smaller compared with when $\varepsilon = 0.0$. Finally, we notice that the first safety violation happens when the attack frequency is $60\%$ and $\varepsilon = 4.0$.

## 5 Future work

One issue we want to explore further is the scalability of our framework. There is a sequence of work has shown the scalability (Gaby, Zhang, and Ye 2021; Dawson et al. 2021) of the neural Lyapunov function and deep reinforcement learning (Pérez-Dattari et al. 2019; Fujimoto, Hoof, and Meger 2018). Hence, it is natural to demonstrate the scalability of our framework better on more complicated benchmarks (Achiam and Amodei 2019; Todorov, Erez, and Tassa 2012). On the other hand, the neural Lyapunov function we learned does not provide a verifiable guarantee (Dai et al. 2021; Chang, Roohi, and Gao 2020; Zhu et al. 2019; Bastani, Pu, and Solar-Lezama 2018) because we do not assume the access of the dynamics of a system. However, in the case that the dynamics can be easily modeled, extending our work with the verifiable tools can provide a strong guarantee on the system safety. This is a direction we are working on. Regarding the system robustness, we only evaluated a limited number of adversarial attacks. It is interesting to see how our framework performs under various attack techniques (Weng et al. 2020; Mankowitz et al. 2020; Zhang et al. 2020) while also consider the attack on the high-level planner (Xiang et al. 2018). Moreover, the current framework only considers a single agent and static obstacles. A challenging but fruitful direction is to extend our framework to multiagent planning and control (Guestrin, Koller, and Parr 2001; Nissim, Brafman, and Domshlak 2010). Lastly, our high-level planner is closely related to the safe neural motion planning (Huang et al. 2021; Qureshi et al. 2019). We propose to investigate further on these works and better refine our high-level planner.

## References

Achiam, J.; and Amodei, D. 2019. Benchmarking Safe Exploration in Deep Reinforcement Learning. In *arxiv*.

Bastani, O.; Pu, Y.; and Solar-Lezama, A. 2018. Verifiable Reinforcement Learning via Policy Extraction. *CoRR*, abs/1805.08328.

Berkenkamp, F.; Moriconi, R.; Schoellig, A. P.; and Krause, A. 2016. Safe Learning of Regions of Attraction for Uncertain, Nonlinear Systems with Gaussian Processes. *arXiv e-prints*, arXiv:1603.04915.

Chang, Y.-C.; Roohi, N.; and Gao, S. 2020. Neural lyapunov control. *arXiv preprint arXiv:2005.00611*.

Chen, T.; Liu, J.; Xiang, Y.; Niu, W.; Tong, E.; and Han, Z. 2019. Adversarial attack and defense in reinforcement learning-from AI security view. *Cybersecurity*, 2(1): 1–22.

Dai, H.; Landry, B.; Yang, L.; Pavone, M.; and Tedrake, R. 2021. Lyapunov-stable neural-network control. *arXiv preprint arXiv:2109.14152*.

Dawson, C.; Qin, Z.; Gao, S.; and Fan, C. 2021. Safe Nonlinear Control Using Robust Neural Lyapunov-Barrier Functions. *arXiv preprint arXiv:2109.06697*.

Fujimoto, S.; Hoof, H.; and Meger, D. 2018. Addressing function approximation error in actor-critic methods. In *International Conference on Machine Learning*, 1587–1596. PMLR.

Gaby, N.; Zhang, F.; and Ye, X. 2021. Lyapunov-Net: A Deep Neural Network Architecture for Lyapunov Function Approximation. *arXiv e-prints*, arXiv:2109.13359.

Goodfellow, I. J.; Shlens, J.; and Szegedy, C. 2014. Explaining and Harnessing Adversarial Examples. *arXiv e-prints*, arXiv:1412.6572.

Grune, L.; and Wurth, F. 2000. Computing control Lyapunov functions via a Zubov type algorithm. In *Proceedings of the 39th IEEE Conference on Decision and Control (Cat. No.00CH37187)*, volume 3, 2129–2134 vol.3.

Guestrin, C.; Koller, D.; and Parr, R. 2001. Multiagent Planning with Factored MDPs. In *NIPS*, volume 1, 1523–1530.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1972. Correction to "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". *SIGART Bull.*, 28–29.

Huang, X.; Feng, M.; Jasour, A.; Rosman, G.; and Williams, B. 2021. Risk Conditioned Neural Motion Planning. *arXiv e-prints*, arXiv:2108.01851.

Kreidieh, A. R.; Berseth, G.; Trabucco, B.; Parajuli, S.; Levine, S.; and Bayen, A. M. 2019. Inter-Level Cooperation in Hierarchical Reinforcement Learning. *arXiv preprint arXiv:1912.02368*.

Levy, A.; Jr., R. P.; and Saenko, K. 2017. Hierarchical Actor-Critic. *CoRR*, abs/1712.00948.

Madry, A.; Makelov, A.; Schmidt, L.; Tsipras, D.; and Vladu, A. 2017. Towards deep learning models resistant to adversarial attacks. *arXiv preprint arXiv:1706.06083*.

Mankowitz, D. J.; Levine, N.; Jeong, R.; Abdolmaleki, A.; Springenberg, J. T.; Shi, Y.; Kay, J.; Hester, T.; Mann, T.; and Riedmiller, M. 2020. Robust Reinforcement Learning for Continuous Control with Model Misspecification. In *International Conference on Learning Representations*.

Nachum, O.; Gu, S.; Lee, H.; and Levine, S. 2019. Near-Optimal Representation Learning for Hierarchical Reinforcement Learning. In *International Conference on Learning Representations*.

Nissim, R.; Brafman, R. I.; and Domshlak, C. 2010. A general, fully distributed multi-agent planning algorithm. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 1-Volume 1*, 1323–1330.

Pérez-Dattari, R.; Celemin, C.; Ruiz-del-Solar, J.; and Kober, J. 2019. Continuous Control for High-Dimensional State Spaces: An Interactive Learning Approach. *arXiv e-prints*, arXiv:1908.05256.

Quinlan, S.; and Khatib, O. 1993. Elastic bands: connecting path planning and control. *[1993] Proceedings IEEE International Conference on Robotics and Automation*, 802–807 vol.2.

Qureshi, A. H.; Simeonov, A.; Bency, M. J.; and Yip, M. C. 2019. Motion planning networks. In *2019 International Conference on Robotics and Automation (ICRA)*, 2118–2124. IEEE.

Sun, Y.; Huang, X.; Kroening, D.; Sharp, J.; Hill, M.; and Ashmore, R. 2018. Testing deep neural networks. *arXiv preprint arXiv:1803.04792*.

Todorov, E.; Erez, T.; and Tassa, Y. 2012. MuJoCo: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 5026–5033.

Urmson, C.; and Simmons, R. 2003. Approaches for heuristically biasing RRT growth. In *Proceedings 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003) (Cat. No.03CH37453)*, volume 2, 1178–1183 vol.2.

Weng, T.-W.; Dvijotham*, K. D.; Uesato*, J.; Xiao*, K.; Gowal*, S.; Stanforth*, R.; and Kohli, P. 2020. Toward Evaluating Robustness of Deep Reinforcement Learning with Continuous Control. In *International Conference on Learning Representations*.

Xiang, Y.; Niu, W.; Liu, J.; Chen, T.; and Han, Z. 2018. A PCA-Based Model to Predict Adversarial Examples on Q-Learning of Path Finding. In *2018 IEEE Third International Conference on Data Science in Cyberspace (DSC)*, 773–780.

Zhang, H.; Chen, H.; Xiao, C.; Li, B.; Boning, D. S.; and Hsieh, C.-J. 2020. Robust deep reinforcement learning against adversarial perturbations on observations. *ICLR*.

Zhao, Y.; Shumailov, I.; Cui, H.; Gao, X.; Mullins, R.; and Anderson, R. 2020. Blackbox attacks on reinforcement learning agents using approximated temporal information. In *2020 50th Annual IEEE/IFIP (DSN-W)*, 16–24. IEEE.

Zhu, H.; Xiong, Z.; Magill, S.; and Jagannathan, S. 2019. An inductive synthesis framework for verifiable reinforcement learning. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 686–701.