CS781: Course Project Report

Enhancing Robustness of Non-Probabilistic Shielding Under Conditions of Environmental Uncertainty

Dion Reji(22B0029) & Dhvanil Gheewala (22B0923)
Department of Computer Science,
Indian Institute of Technology Bombay

October 2024

Abstract

Reinforcement learning (RL) algorithms often rely on environmental feedback to adapt and make decisions, yet such feedback can be unreliable or unavailable in real-world scenarios. To address this, we propose a framework that extends the robustness of the non-probabilistic shield introduced in "Safe Reinforcement Learning via Shielding." Our approach enhances the shield's ability to safeguard the RL agent in situations where the environmental responses to its actions are intermittently absent. By introducing constraints on the agent's level of unawareness, defined by a maximum number of oblivious steps within a fixed time horizon, we enable the shield to preemptively compute safe action strategies. The synthesized shield ensures the agent avoids unsafe states under these conditions, terminating only if no viable safe actions remain. This project demonstrates how informed decision-making and preemptive safety strategies can extend the utility of shields in dynamic and uncertain environments, paving the way for more resilient safe RL systems.

1 Introduction

Reinforcement learning (RL) has emerged as a powerful paradigm for solving complex decision-making tasks, with applications ranging from robotics to autonomous systems. Despite its successes, the safety of RL systems remains a significant concern, particularly in high-stakes environments where failures can have severe consequences. Shielding mechanisms have been introduced as a solution to this challenge, providing an additional layer of protection by ensuring that the agent avoids unsafe states during learning and execution.

The concept of shielding is particularly critical in scenarios where the environment is unpredictable or partially observable. While existing shielding techniques have demonstrated effectiveness under ideal conditions, real-world systems often encounter challenges such as unreliable sensors, communication delays, or unforeseen disruptions. In such cases, the shield may fail to perceive the environmental responses to the agent's actions, potentially compromising its ability to ensure safety.

This project focuses on enhancing the robustness of non-probabilistic shields in scenarios characterized by partial environmental observability. By introducing constraints on the level of unawareness and precomputing safety information under these conditions, we aim to extend the shield's protective capabilities. Our approach addresses a critical gap in current shielding methodologies, enabling them to function reliably even when environmental feedback is intermittent or unavailable. Through this work, we aim to contribute to the development of safer, more resilient RL systems capable of operating in dynamic and uncertain environments.

2 Problem Statement

Let S be the set of states in the environment, and A be the set of actions available to the agent. The transition function of the environment is defined as $T : S \times A \to \Delta S$, which maps a state-action pair to a distribution over states, ΔS. A subset $S_{\text{safe}} \subseteq S$ represents the set of safe states such that any state $s \notin S_{\text{safe}}$ is unsafe.

The agent operates in an episodic environment where it perceives feedback from the environment after executing an action. However, the agent may encounter situations where the environmental feedback is unavailable for a finite number of steps, referred to as *oblivious steps*. Such *oblivious steps* are when we call the shield to *blank out*. The shield must ensure the agent avoids unsafe states despite these interruptions in feedback.

To formalize this, we define a **window parameter** n, representing the total number of steps considered. Let $k \leq n$ represent the maximum number of oblivious steps allowed within the window of n steps. For each step $t \in \{1, 2, \ldots, n\}$, the agent may experience a state o_t where $o_t \in \{0, 1\}$:

- $o_t = 0$ implies feedback from the environment is available.
- $o_t = 1$ implies feedback is unavailable (oblivious step).

The **sequence of obliviousness** within n steps is given as $O = [o_1, o_2, \ldots, o_n]$, where $\sum_{i=1}^{n} o_i \leq k$.

Our shield, Shield operates over the inputs

- Parameters k and n.
- An abstraction of the MDP of the environment, (S, A, T).
- A safety specification , φ which defines the safe states safe states S_{safe}.

At each step t, based on the current state s_t and the sequence O up to step t, compute a set of safe actions $A_{\text{safe}}(s_t, O_t) \subseteq A$ that ensures:

$$\forall a \in A_{\text{safe}}(s_t, O_t),\ T(s_t, a) \in S_{\text{safe}}, \quad \text{for all possible future oblivious sequences within } n.$$

If $A_{\text{safe}}(s_t, O_t) = \emptyset$, terminate with the message: "No safe steps found". At any step, the shield should ensure that $\sum_{i=1}^{n} o_i \leq k$

The problem requires the synthesis of a shield that computes $A_{\text{safe}}(s_t, O_t)$ dynamically while adhering to the constraints on obliviousness and termination conditions.

3 Choice of tool

For this project we have chosen `tempest_in_action`.

Some of the major shortcomings of the tool, which posed challenges for us, are outlined below:

1. **Insufficient Comments and Documentation:** The provided code lacked adequate comments and clear documentation, making it difficult to understand the functionality of various components. This resulted in significant time spent understanding the implementations and adapting them to our needs.
2. **Ambiguity in Internal Implementation:** Many internal implementations were not well-documented, leaving their functionality unclear. This made it challenging to interact with the environment and extract the desired states and actions effectively.
3. **Challenges in Building the Notebook:** We encountered considerable difficulties in setting up the Docker image and building the Jupyter Notebook file, which delayed our progress.
4. **Debugging Difficulties:** The lack of transparency in the tool's internal decision-making process hindered debugging and verification of correctness. Additionally, the use of Python's `try-except` blocks to handle errors suppressed error messages, making it harder to identify and resolve issues.

4 Approach

Our project builds upon the repository `tempest_in_action`, extending its functionalities to enable the synthesis of the shield under conditions of partial environmental observability. The repository provides a baseline environment where the agent always perceives environmental feedback without interruptions. To simulate the oblivious steps described in our problem statement, we introduce new functions and classes into the existing framework.

4.1 Simulation of Oblivious Steps

To handle scenarios where the agent experiences loss of perception, we implement the class `PerceptionLossSimulator`. This class is responsible for simulating oblivious steps and integrates seamlessly into the existing environment provided by the repository.

The primary responsibilities of the `PerceptionLossSimulator` class include:

- **Initialization:** The class takes two parameters, k (the maximum number of oblivious steps allowed) and n (the total window size), during initialization. It validates that $k \leq n$ and both values are non-negative.
- **Tracking Perception Loss:** The simulator maintains a fixed-length dequeue to track the last n steps, recording whether each step was normal or oblivious. It also keeps a count of the current number of oblivious steps within this window.
- **Validation of Constraints:** At each step, the simulator checks whether adding a new oblivious step would exceed the maximum allowed number k. If k is already reached, further oblivious steps are not allowed.
- **Simulating Steps:** For each new step, the simulator decides whether it will be oblivious or normal. If the maximum oblivious limit has not been reached, a probabilistic function determines the likelihood of the step being oblivious.
- **Reset and Statistics:** The simulator provides methods to reset its state to the initial configuration and to retrieve the current statistics, such as the number of steps taken, the current window state, and the count of oblivious steps.

4.2 Integrating simulator with the Shield

The shield is integrated into the environment through the `MiniGridSbShieldingWrapper`, which has been extended to accept k and n as parameters. The `PerceptionLossSimulator` is incorporated as a member of `MiniGridSbShieldingWrapper` to simulate partial observability conditions.

The `PerceptionLossSimulator` acts as an abstraction layer that introduces intermittent perception loss. At each step, when the shield wrapper receives a response from the environment, the simulator determines whether the observation should be disregarded based on whether the current step is oblivious. The shield is then informed about whether the step was blanked, enabling it to dynamically adapt its decisions in real-time.

By including the `PerceptionLossSimulator` in the shielding framework, the system can test and evaluate the robustness of the shield in environments with varying levels of partial observability. This ensures that the shield performs effectively even when the agent cannot reliably perceive its surroundings.

4.3 Shield Operation and Action Recommendation Strategy

The Shield employs the `MiniGridShieldHandler`, which computes a pre-emptive shield and stores it in the `shield` field of the `MiniGridSbShieldingWrapper`. This base shield is used to determine the final set of recommendable actions by selecting suitable actions based on the shield's logic. The algorithm is in 1. We explain this briefly here.

Case 1: Perception is Successfully Retained by the Shield

When the response from the environment is successfully perceived, the shield directly utilizes the action recommendations provided by the base shield. In this case, no perception loss has occurred, and the shield does not need to make additional adjustments.

Case 2: Perception is Lost and the Response is Not Received by the Shield

When the response from the environment is lost, the shield evaluates the current state using statistical information provided by the simulator. It calculates the maximum possible number of continuous oblivious states to assess the safety of future actions.

In such cases, the shield recommends actions by considering the current state s_t and the perception loss history. The goal is to ensure that the recommended actions remain safe, even when the agent lacks full visibility of the environment.

Determining Maximum Consecutive Blank States: The shield calculates the maximum possible number of consecutive oblivious steps by iterating from 1 to k and evaluating the feasibility of such scenarios using the past n steps. The maximum value derived from this process is used for further decision-making.

Let $\mathcal{S}^k(s_t)$ denote the set of states reachable from s_t under the assumption of up to k consecutive oblivious steps. The shield's decision-making process can be outlined as follows:

1. **Check Perception Loss:** The shield determines whether the current step is blanked:

   $$\texttt{is\_blanked} = \texttt{PerceptionLossSimulator.step()}$$

   If the step is not blanked, the shield directly uses the current state s_t to recommend actions.

2. **Conservative Action Selection:** If the step is blanked, the shield selects actions conservatively based on the last known safe state s_{last}. The total consecutive blanked states are calculated as the sum of:

   - The number of blanked steps since s_{last} (denoted as τ).
   - The additional possible blank states (k) derived from the simulation statistics.

   If s_{last} is unavailable or the uncertainty in the system is too high, the shield defaults to the most conservative action set, ensuring safety across all possible states.

   The set of safe actions, $\mathcal{A}^k_{\text{safe}}(s_t)$, is defined as the set of actions that are safe for a depth k across all reachable states:

   $$\mathcal{A}^k_{\text{safe}}(s_t) = \mathcal{A}^k_{\text{safe}}(\mathcal{S}^\tau(s_{\text{last}})) = \bigcap_{s_i \in \mathcal{S}^\tau(s_{\text{last}})} \mathcal{A}^k(s_i)$$

   Here, $\mathcal{A}^k_{\text{safe}}(s_t)$ represents the set of safe actions for the current state s_t, which will be output by the final shield.

3. **Adaptation Based on Perception Loss:** When perception is lost, the shield adapts by calculating the set of states reachable within the next k steps, given the safe actions. The shield then computes the intersection of action sets across these states to ensure robust and safe recommendations under uncertainty.

4. **Return Safe Action List:** The shield returns the recommended actions based on the perception state:

   - If perception is lost, the shield recommends a conservative set of actions that are safe across all possible states.
   - If perception is retained, the shield uses the base shield's recommendations for the current state s_t.

Algorithm 1 Shield Action Recommendation Process

```
1: Input: Current state $s_t$, maximum oblivious steps $k$, total window size $n$
2: Output: Action mask $\mathcal{A}_{\text{safe}}(s_t)$
3: if PerceptionLossSimulator.step() is False then          ▷ Step is not blanked
4:     return $\mathcal{S}(s_t)$                             ▷ Return safe actions for current state
5: else
6:     if last_known_state is None then                      ▷ No last known state available
7:         return conservative action                        ▷ Return conservative actions
8:     end if
9:     $s_{\text{last}}$ ← last_known_state
10:    continued_blank_steps ← total_steps − last_known_step − 1
11:    possible_current_states ← get_possible_states($s_{\text{last}}$, continued_blank_steps + possible_blank_states)
12:    safe_actions ← [1.0] * len($\mathcal{S}(s_{\text{last}})$)
13:    for each state in possible_current_states do
14:        if state in shield then
15:            for each action in safe_actions do
16:                safe_actions[action_idx] = min(safe_actions[action_idx], $\mathcal{S}$(state)[action_idx])
17:            end for
18:        else
19:            return conservative actions                   ▷ Fallback to conservative actions
20:        end if
21:    end for
22:    return safe_actions
23: end if
```

5 Files Modified

We have modified the files `sb3utils.py` and `utils.py`. The following functions have been modified:

5.1 utils.py

In `utils.py`, we have created a class `PerceptionLossSimulator` with the following functions:

Class: PerceptionLossSimulator

This class simulates perception loss (or blanked steps) within a specified window, ensuring that the number of blanked steps does not exceed a given threshold k. It maintains a window of recent steps and tracks the number of blanked steps in that window.

Methods:

- `__init__(self, k: int, n: int)`: Initializes the perception loss simulator.
  - **Parameters:**
    - k (int): Maximum number of blanked (oblivious) steps allowed.
    - n (int): Total window size to consider.
  - Ensures that k is not larger than n and both values are non-negative.
- `can_blank_next(self) -> bool`: Checks if the next step can be blanked while respecting the constraints of k and n. If the window is full, it checks if blanking the next step would violate the allowed number of blanked steps.
- `step(self) -> bool`: Simulates one step and determines if perception is lost (blanked).
  - **Returns:**
    - `True` if the step is blanked.
    - `False` if the step is normal (perception is not lost).
- `get_window(self) -> list`: Returns the current window state as a list of Boolean values indicating whether each step was blanked.
- `get_stats(self) -> dict`: Returns the current statistics of the simulator, including the window size, count of blanked steps, and the current window state.
- `reset(self)`: Resets the simulator to its initial state, clearing the window and resetting the count of blanked steps.

5.2 sb3utils.py

In `sb3utils.py`, we have modified the class `MiniGridSbShieldingWrapper` and extended it to incorporate the new parameters k and n. This extension enables the shield to handle perception loss by leveraging the functionality of the `PerceptionLossSimulator` class. Below are the main changes introduced:

- **Constructor Modification:** The constructor of `MiniGridSbShieldingWrapper` was updated to accept two additional parameters:
  - k: Maximum number of blanked (oblivious) steps allowed in the perception loss window.
  - n: Total window size for tracking perception loss.

  These parameters are used to initialize an instance of the `PerceptionLossSimulator`.
- **Modified Methods:** Several methods were updated or introduced to incorporate perception loss handling:
  - `__init__()`: Initializes the wrapper and sets up the perception loss simulator.
  - `_get_conservative_actions()`: Returns the most conservative set of actions by intersecting the allowed actions across all states in the shield.
  - `get_possible_states()`: Computes all possible states reachable within a specified depth by considering safe actions at each step.

  - `get_shield_action()`: Computes the allowed actions for a given state while taking perception loss into account. This method uses the simulator to decide whether perception is lost and adapts the safe action set accordingly.
  - `create_action_mask()`: Integrates the shield's output with perception loss handling, ensuring no unsafe actions are taken even under uncertainty.
  - `reset()`: Resets the environment and re-creates the shield if needed.
  - `step()`: Extends the `step` method to include perception loss-related checks and updates the shield information.

- **Comments and Debugging Statements:** Several print statements were added for debugging purposes. These statements help in understanding the shield's behavior during perception loss, such as:
  - `Stats of simulator {...}`: Displays the current status of the perception loss simulator.
  - `Agent blanked out at <step_number>`: Indicates when and at which step the agent loses perception.
  - `Old Shield's Action -------->`: Prints the original safe actions from the shield.
  - `New Shield's Safe actions -------->`: Shows the updated safe actions after handling perception loss.

The modifications enable the shield to adapt dynamically to perception loss scenarios while ensuring robust and safe behavior.

6 Experiments Conducted and Results

Running Instructions

To run our modified shield, follow these steps:

1. Build the original shield, Tempest, along with its dependencies.
2. Replace the files `utils.py` and `sb3utils.py` with the modified versions provided in the submission folder.
3. Open the Jupyter notebook file of the environment to work with.
4. Add the extra parameters k and n when invoking `MiniGridSbShieldingWrapper` to enable the functionality of the updated shield (change their value in the function call).

Interpreting the Results

To facilitate result interpretation, print statements have been added to the code. Below are the details of the outputs and comments you may encounter:

- **Safe Actions Calculation:** Inside the function `get_shield_actions()`, the shield calculates the safe actions based on the simulator's statistics. For example:

```
Inside get_shield_actions() calculating safe actions
Stats of simulator
```

- **Agent Perception Loss:** When the agent experiences perception loss, a message is displayed indicating the step number where it occurs:

```
Agent blanked out at 2
```

- **Possible State Simulation:** The shield simulates possible states symbolically for finding feasible actions. Each simulated state is displayed with its attributes. For example:

```
symbolically simulated new state State(colAgent=np.int64(1), rowAgent=np.int64(7),
viewAgent=1, carrying='', adversaries=(), balls=(), boxes=(), keys=(),
doors=(), lockeddoors=())
```

- **Shield Actions:** The safe actions recommended by the original shield (`Old Shield's Action`) and the updated shield (`New Shield's Safe Actions`) are displayed for comparison. For example:

```
Old Shield's Action-------->
[1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0]
New Shield's Safe Actions -------->
[True, True, False, False, False, False, False]
```

- **Action Mask Creation:** Each time an action mask is called, it is logged in the output:

```
create_action_mask() --- 1
```

The output provides a detailed breakdown of the shield's functioning at each step, enabling effective debugging and analysis.

7 Experiments

We conducted our experiments in four different environmental settings which was given as a part of the `tempest_in_action` repository:

- Slippery Cliff
- Hello Lava Gap
- Playground
- Faulty Actions

We used our modified Shield wrapper in these environments, which uses the new robust shield. The major results are outlined below:

- Our shield recommends only safe actions. This is because it is built on top of the existing pre-emptive shield handler and selects out the relevant actions as per the state input from the simulator.
- Our shield recommends actions which are a subset of that recommended by the original handler. This is because we take the intersection of safe states up to a depth when we blank out. In non-oblivious steps, the shield recommends the same state set as the original.
- However, in experiments we noticed that most of the time the recommendations of our shield are the same as that of the original one. This is because, in the original shield most of the action recommendations are the same. (There was a recurring pattern of always recommending the first three action in almost all the cases). So, when our shield takes intersection it doesn't cause much change. Nevertheless, we got situations where the recommended actions are a strict subset of the original action set.
- Our shield stops after some steps in some cases. This is because due to Perception Loss, it might not get enough info to decide the actions. In such cases, the intersection of safe actions might be empty and our shield couldn't recommend any actions.

8 Remaining Shortcomings and Future Work

Despite the positive results, several shortcomings remain, providing opportunities for improvement and future research:

1. **Handling of Prolonged Uncertainty:** In scenarios where the system transitions through prolonged periods of uncertain states (e.g., extended blank steps), the shield's conservative approach may overly restrict actions or halt unnecessarily. Future work could use probabilistic shields to better handle such uncertainties without compromising safety.
2. **Optimizing the shield for essential recommendations:** The current implementation builds on the original pre-emptive shield, running its handler to identify a base shield and determine the set of recommendable actions. Efficiency could be improved by streamlining this process to avoid unnecessary computations for state recommendations that are not required.

3. **Fault Tolerance in Perception Loss:** The current shield halts when no recommendation can be made. While this ensures safety, it does not address potential recovery strategies. Future work could involve integrating recovery planning, enabling the system to actively attempt to regain reliable state information. In future work, we could try some strategies to get a safe action without halting [1].

4. **Explainability of Recommendations:** The shield currently functions as a black box in terms of why certain actions are restricted. Improving the explainability of its decision-making process could foster trust and provide insights for debugging and iterative improvements.

By addressing these shortcomings, the shield can be developed into a more adaptive, efficient, and universally applicable safety mechanism, opening avenues for its deployment in a broader range of applications.

[1] We haven't thought of anything along this line