Safe Deployment of a Reinforcement Learning Robot Using Self Stabilization

Nanda Kishore Sreenivas and Shrisha Rao

June 24, 2021

Abstract

In toy environments like video games, a reinforcement learning agent is deployed and operates within the same state space in which it was trained. However, in robotics applications such as industrial systems or autonomous vehicles, this cannot be guaranteed. A robot can be pushed out of its training space by some unforeseen perturbation, which may cause it to go into an unknown state from which it has not been trained to move towards its goal. While most prior work in the area of RL safety focuses on ensuring safety in the training phase, this paper focuses on ensuring the safe deployment of a robot that has already been trained to operate within a safe space. This work defines a condition on the state and action spaces, that if satisfied, guarantees the robot's recovery to safety independently. We also propose a strategy and design that facilitate this recovery within a finite number of steps after perturbation. This is implemented and tested against a standard RL model, and the results indicate a much-improved performance.

Keywords: safety in robotics, reinforcement learning, self stabilization

1 Introduction

While some early works in reinforcement learning (RL) were restricted to simple environments [1, 2], recent works have used RL to solve problems in real-world settings where the stakes are much higher. Over the past decade, RL has found its way into robotics [3], autonomous vehicles [4], controlling traffic signals [5], and optimizing chemical reactions [6] among many other practical use cases. Therefore, it can be seen that such robots trained using RL (which we refer to as "RL robots") cannot afford to focus

only on performance; they also need to ensure their safety in addition to that of their surroundings. Safety in robotics has been considered to be a major bottleneck in the safe and productive use of robots in industry and manufacturing [7].

Safety in RL has been a topic of interest recently. Achiam *et al.* [8] proposed Constrained Policy Optimization (CPO) as a trust region method, which offered near-constraint satisfaction. A different approach to address the problem of safe exploration is to add a safety layer that corrects the action choice to never violate constraints during training [9]. This is done by pretraining based on past trajectories made up of arbitrary actions. Another class of solutions uses Lyapunov functions to guarantee safety during training [10, 11]. Yet another recent work used a QP solver for ensuring safety [12]. Gehring and Precup [13] propose a different approach based on the notion of controllability computed from temporal difference errors. Human demonstrations have also been used to constrain exploration to ensure safety [14]. Recently, inverse RL has been used to learn human perception of safety and hard safety constraints based on successful demonstrations [15]. For an expanded overview of related work, see the survey by García and Fernández [16] on safe RL.

A common theme across all these works is that they are more focused on ensuring safety during training, which is certainly an important concern. Andersson and Doherty [17] show that popular RL algorithms which generally perform well on simple toy environments fare poorly when random perturbations are introduced. Also, even robots trained with safe exploration methods such as CPO in simulation are unsuccessful at some tasks when deployed in the real world [18]. Dalal *et al.* [9] write, "Safety is a crucial concern: unless safe operation is addressed thoroughly and ensured from the first moment of deployment, RL is deemed incompatible for them." Chow *et al.* [10] also concur: "Besides optimizing performance it is crucial to guarantee the safety of an agent in deployment, as well as during training."

While most agree that safety is as important in deployment as it is in training, there has not been much work on safety in deployment. This could be because of an implicit assumption that a well-trained robot taught to explore safely will also remain within safety in deployment. While that could be true, a robot may enter an unsafe state unintentionally due to some unforeseen external perturbation in the environment. In such an event, the robot should recover to safety as soon as possible, but it may be in a hitherto unseen state and there may exist no learned policy that the robot can readily use.

This has been defined as one of the challenges in AI safety by Amodei

et al. [19], who discuss the scenario where an agent finds itself in a space different from the one it was originally trained in:

> "In general, when the testing distribution differs from the training distribution, machine learning systems may not only exhibit poor performance, but also wrongly assume that their performance is good. More broadly, any agent whose perception or heuristic reasoning processes are not trained on the correct distribution may badly misunderstand its situation, and thus runs the risk of committing harmful actions."

We precisely focus on this aspect of safety: how can a trained RL robot quickly recover when perturbed? Work on RL and its applications suggests that systems using RL may be subject to transient disturbances, and specifically that robots trained using RL may encounter issues during deployment that have not been seen during training. We illustrate that self stabilization, a classical paradigm of distributed computing, can be applied to RL to address these problems.

We first define a recoverability condition on the state space, which if satisfied guarantees the robot's return to safety when unsafe. We offer a prototype of an RL robot that learns the consequence of each action during the training phase, *i.e*, the change in state that occurs. During deployment, when the robot enters an unsafe territory, it determines the state change required to navigate back to safety. The robot now finds a sequence of actions that help lower its distance to safety based on the relationship between action and state change that it had learned during training. Thus, the suggested robot model can stabilize itself when pushed out of the safe space due to some unexpected perturbation.

We implement this model and test it on a simulated maze environment where the robot navigates a maze to reach its goal. There are high negative rewards when the robot is in an unsafe state. The trained RL robot is deployed with and without self stabilization. Perturbations are simulated probabilistically with the same frequency in both cases. While the RL robot without stabilization is found to recover from roughly 50% of perturbations, but the RL robot with self stabilization (abbreviated as RL+SS) achieved a 100% recovery. It is also observed that the average score after 5000 episodes of deployment is significantly higher for the robot using the RL+SS strategy.

Further, the same strategy was implemented in a simulated environment based on the Atari game of Lunar Lander. To find out if RL+SS can achieve better scores even in the absence of systemic safety constraints, different artificial safety constraints that direct the robot toward its goal are tested.

For an already trained RL robot, choosing the right constraint resulted in slightly better scores.

2 Self Stabilization

In distributed computing, self stabilization was conceived by Dijkstra [20] as the property of a system that can return to a valid state in spite of the lack of a centralized control. Equivalently, a self-stabilizing system is one whose current state only depends on its previous k inputs (for some constant k). Such a system is guaranteed to stabilize, post the occurrence of a fault, once it processes k inputs.

Self stabilization may also be looked upon as a special case of *non-masking* fault-tolerance [21, 22] where a system is able to recover from arbitrary transient faults. It has found applicability in different contexts where algorithms are needed that can help a system recover to a valid state after a fault.

We denote the set of all states by $\mathbf{S}$. For any predicate P, we define $\mathbf{S}_P$ as the set of states in $\mathbf{S}$ where predicate P holds.

Definition 1. *We define three predicates which serve as the basis for all subsequent development:*

1. $L : \mathbf{S} \to \{T, F\}$ *is a* legitimacy predicate*, such that* $L(x) = T$ *if* x *is a legitimate state in* $\mathbf{S}$*. The set of all legitimate states is denoted by* $\mathbf{S}_L$*.*
2. $Z : \mathbf{S} \to \{T, F\}$ *is a* safety predicate*, such that* $Z(x) = T$ *if* x *is a safe state in* $\mathbf{S}$*. Safe states* $\mathbf{S}_Z$ *denotes the set of all states where* Z *holds;* $\mathbf{S}_Z \subseteq \mathbf{S}_L$*.*
3. $Z' : \mathbf{S} \to \{T, F\}$ *is a* non-safety predicate*, such that* $Z' = \bar{Z} \wedge L$*. Unsafe states* $\mathbf{S}_{Z'}$ *are those legitimate states where* Z *does not hold.*

Safe states are those where the robot is supposed to operate, and unsafe states are the ones from where the robot can recover to safe states. Legitimate states refer to those states that are either safe or unsafe, and illegitimate states are the absolute worst case where recovery is also not possible.

The distance between two states s_l and s_k is denoted by $d(s_l, s_k)$. The exact distance metric can vary based on the environment. The distance to

safety, denoted by $d_Z(x)$ is a measure of how close a given state x is to safety; it is the minimum distance between x and any safe state in $\mathbf{S}_Z$.

$$d_Z(x) = \begin{cases} 0 & x \in \mathbf{S}_Z \\ \min_{\forall i \in \mathbf{S}_Z} d(x, i) & x \notin \mathbf{S}_Z \end{cases} \tag{1}$$

Self stabilization is the process of a robot independently stabilizing itself starting from an unsafe state to a safe state through a finite sequence of actions, provided there are no further perturbations.

When the robot is at an unsafe state x, a properly chosen action will move the robot to a state x', such that $d_Z(x') < d_Z(x)$. Such an action is termed a *stabilizing nudge*. In simpler words, a stabilizing nudge moves the robot towards safety. Alternatively, stabilization can be seen as a finite sequence of stabilizing nudges until the robot enters a safe state.

The robot has a set of permissible actions $\mathbf{A}$, whose cardinality is denoted by $n_{\mathbf{A}}$. When a certain action a is taken from a state s_t, the transition function δ returns the next state and a real-valued reward. $\delta : S \times A^n \rightarrow S \times \mathbb{R}$, and is given by:

$$\delta(s_t, a) = (s_{t+1}, \mathbf{r}_t) \tag{2}$$

The environment defined by the state space $\mathbf{S}$ and the robot's actions $\mathbf{A}$, is said to be recoverable if it satisfies the recoverability condition:

$$\forall s_1 \in \mathbf{S}_{Z'}, \exists s_2 \in \mathbf{S}_Z \mid \delta(s_1, \pi_k) = s_2 \tag{3}$$

where π_k is a finitely long sequence of actions.

Remark 1. *If the recoverability condition is satisfied, then stabilization is always possible.*

The contradiction of the above is that the recoverability condition is satisfied and there exists a case where stabilization is not possible. If stabilization is not possible, then there should be at least one unsafe state that cannot be stabilized through a finite sequence of actions, which is the negation of the recovery condition in (3). Thus, the contradiction is false, and the remark is true.

3 Self Stabilization in RL

An RL robot is typically trained in a training space, and then deployed in the real world. When the RL robot has been deployed, it could enter a state outside the training space due to some external, unforeseen perturbation.

As discussed in Section 2, each robot working in some environment typically has a legitimacy predicate L. We assume that robots are trained in the safe space, and hence use the terms training space and safe space interchangeably from here. Even though the robot is trained in $\mathbf{S}_Z$, there is no guarantee that the robot has been to all safe states. So, we define one more predicate as follows.

Definition 2. $F : \mathbf{S_Z} \rightarrow \{T, F\}$ *is the* familiarity predicate*, such that* $F(x) = T$ *if the robot has been in state* x *during the training phase. The set of states encountered by the robot in the training phase is denoted by* $\mathbf{S_F}$ *and,* $\mathbf{S}_F \subseteq \mathbf{S}_Z$*.*

State change function Δ, returns the vector difference between any two given states. If the robot transitions from state s_m to s_{m+1} by performing an action a, then the state change is given by

$$\Delta(s_m, s_{m+1}) = s_{m+1} - s_m \tag{4}$$

The state change function Δ would need some transformation if the attributes representing the state are not numeric. There are many transformation functions available in the ML literature to do this. Of course, there could be scenarios where the standard transformations are not applicable, but such cases can only be dealt with on a case by case basis.

States could be continuous in some environments, and even when discrete, there could be too many possible values of state changes in some environments. So, to handle such cases, the state changes could be discretized into some finite types. The set of state changes is denoted by Σ and its cardinality is denoted by n_Σ.

There are many different algorithms within the domain of RL, but we use Q-learning [1] in all the environments here. The goal is for the robot to learn the optimum action to take under different circumstances. Central to the algorithm is the quality function, which returns the quality of a state-action pair, $Q : \mathbf{S_Z} \times \mathbf{A} \rightarrow \mathbb{R}$.

The Q-values for all state-action pairs are initialized with some arbitrary value. Then, at some time t, the robot chooses some action a from a state s_t, and observes a reward of $\mathbf{r}_t$. The new quality of the state-action pair is updated based on the Bellman equation.

This whole process of selecting an action and updating the quality function helps the robot learn which actions to choose under various circumstances. This is done repeatedly until a certain number of episodes are over, or until a certain average score is achieved. Once the training terminates,

the robot can now be deployed. Now, when the robot is at state s, the optimum action is chosen by

$$a^* = \underset{a_i \in \mathbf{A}}{\operatorname{argmax}} Q(s, a_i) \tag{5}$$

As discussed above, the Q-function maps all safe state-action pairs. So, when the robot has been perturbed and is in a state $s' \in \mathbf{S}_{Z'}$, it has no row in the Q-Table to refer to, and hence cannot decide which action is appropriate. In such a case, the robot has only two possible courses of action—to remain idle, or choose an action at random from $\mathbf{A}$. Both of these cannot guarantee the recovery of the robot, and randomly choosing an action could further aggravate the situation by landing the robot in an illegitimate state.

However, when the recoverability condition is satisfied by the environment (3), the robot can be stabilized and return to safety if it maintains some extra information during training. The robot characteristics that facilitates this, and the maze environment where this model is tested are explained in the following subsections.

3.1 Robot Characteristics

Each robot has the following four attributes:

- Current state s_t, which is typically an n-tuple and is defined specifically for each environment.
- Visited states Ψ, a set of all states visited by the robot during training.
- Action-state change table Ω, a table which holds the number of times each action induces a certain state change.
- Q-Table or a Deep Q-Network $\mathcal{Q}$, which is used to keep track of Q-values. The choice depends on the particular environment.

Visited states Ψ is the set of familiar states, *i.e.,* the states where the familiarity predicate F holds. The only purpose of maintaining this is for the robot to find the closest safe state, and consequently the distance to safety given by (1) when unstable.

The action-state change table, Ω is an $n_{\mathbf{A}} \times n_{\Sigma}$ matrix where $\Omega[i][j]$ holds the number of times action i induces a state change j. This is used to calculate conditional probabilities to decide which action is most probable

to induce the required state change, $\sigma_x \in \Sigma$. Conditional probabilities can be calculated from Ω as:

$$P(\sigma_x|a_k) = \frac{P(\sigma_x \wedge a_k)}{P(a_k)}$$
$$P(\sigma_x|a_k) = \frac{\Omega[a_k][\sigma_x]}{\Omega[a_k]}$$
$$where, \Omega[a_k] = \sum_{\forall l \in \Sigma} \Omega[a_k][l]$$

Here, the probability of state change σ_x and action a_k happening together is given by $\Omega[a_k][\sigma_x]$, and the marginal probability $P(a_k)$ as the sum of all cells in the row corresponding to a_k.

The action $a^*_{\sigma_x}$ which is most likely to bring about a state change of σ_x is given by

$$a^*_{\sigma_x} = \underset{a_i \in \mathbf{A}}{\operatorname{argmax}} P(\sigma_x|a_i) \tag{6}$$

$P(\sigma_x|a_i)$ is the probability of inducing a state change of σ_x given that an action a_i has been chosen. So, to maximize the chances of getting σ_x, the action should be chosen such that the conditional probability is maximized. Based on Ω, the robot finds the action which can mostly likely effect a state change of σ_x, as given by (6).

The Q-Table is a matrix used to store the quality of all state-action pairs, and hence it is of dimensions $|\mathbf{S}_Z| \times n_{\mathbf{A}}$. In environments with a large number of discrete states, or when states are continuous, the number of rows grows very large and cannot be efficiently handled. In such cases, Q-learning is used along with function approximation techniques. One of the popular solutions is to use an artificial neural network as the function approximator.

3.2 Working of the Robot

3.2.1 Modified Q-learning

To accommodate the robot characteristics discussed in Section 3.1, and to learn the relation between actions and state changes, the standard Q-learning algorithm is modified to additionally capture the effect of actions on state changes. Similar to standard Q-learning, the robot chooses an action a either by exploration or exploitation based on the current state s. The transition function δ given by (2) is used to find the next state s_{t+1} and the reward $\mathbf{r}_t$. The state change function Δ, given by (4), is used to determine the state change between s and s_{t+1}, and this is stored as c. The

corresponding entry in the action-state change table denoted by $\Omega[a][c]$ is updated.

3.2.2 Self Stabilization

When the robot is deployed and if it is in unsafe territory, the self stabilization method, outlined by Algorithm 1, is used to recover. The input to this stabilization method includes current state s_t, and the action-state change table Ω. The method returns the state of the robot after stabilization, and the total reward accumulated due to the actions performed. The total reward ρ is initialized as 0 in line 1. The robot then iterates through its set of visited states Ψ to find the targeted safe state η and the corresponding distance to safety ϖ, seen in lines 2 and 3. From the set of all state changes Σ, the best value σ_x is chosen such that it minimizes distance to safety, as seen in line 4. The action $a^*_{\sigma_x}$ that is mostly likely to induce the required state change σ_x is found using the `findAction` method, which is an implementation of (6). Line 7 uses the transition function δ given by (2) to find the next state and reward associated with this action, and then ρ is updated. ϖ is now updated as the distance between updated state s_t and the targeted state η by (1). If $\varpi = 0$ (line 4), the function terminates and returns the current state, which is safe by (1), and the total reward accumulated by all the actions performed. Else, the same set of steps (lines 5–9) is repeated.

3.3 Maze

The Maze environment [23] is a simple 2D grid-based environment where the robot finds its way from a start position to the goal. Each cell in the grid is represented by its row and column numbers (r, c). The robot can move in all four directions and only one step at a time. As in any typical maze, movement along certain directions could be blocked in some cells. So, if the robot attempts to execute such an impermissible action from such a state, it remains at the same state.

The state of the robot is represented by an ordered pair $\langle r, c \rangle$, which is simply the position of the robot in the grid. A 20×20 grid is considered the legitimate space ($\mathbf{S}_L$) in this scenario; going outside of this grid is irrecoverable and has a high penalty. Centered within this large grid, a smaller 10×10 grid is considered the safe space $\mathbf{S}_Z$.

Each episode starts with the robot at any random position within the safe space. The goal or destination is fixed at $(15, 15)$. An episode ends either when the robot reaches its goal, or when it transitions into an illegitimate

Algorithm 1: Self-Stabilization Method

Input : s_t,Ω
Output: s_t,ρ

$\rho \leftarrow 0$;
/* Find the closest state and the corresponding distance to safety */
$\eta \leftarrow$ closest state in Ψ;
$\varpi \leftarrow \mathtt{d}(\eta, s_t)$;
while $\varpi \neq 0$ **do**
    /* Find the best state change */
    $\sigma_x \leftarrow \text{argmin}_{j \in \Sigma}\ \mathtt{d}(s_t + j, \eta)$;
    /* Find action corresponding to σ_x */
    $a^*_{\sigma_x} \leftarrow \mathtt{findAction}(\sigma_x, \Omega)$;
    /* Update state and total reward */
    $s_t, \mathbf{r}_t \leftarrow \delta(s_t, a^*_{\sigma_x})$;
    $\rho \leftarrow \rho + \mathbf{r}_t$;
    $\varpi \leftarrow \mathtt{d}(s_t, \eta)$;
end
return s_t,ρ

state.

The entire grid which is the legitimate space is a square bound by $(1, 1)$ and $(20, 20)$, the top left and bottom right cells respectively. The safe space, which is the inner square is bound by $(6, 6)$ and $(15, 15)$.

As described earlier, the robot can move one step in all four directions. So, the action space is a four member set given by

$$\mathbf{A} = \{N, S, E, W\}$$

Based on the actions and the state space defined, it can be clearly seen that a robot starting at any unsafe state can ultimately reach a safe state through a finite sequence of actions. Thus, the recoverability condition (3) is satisfied in this environment. Since there are four possible actions, only four different state changes are possible due to any action. Since this is a maze, and sometimes certain actions are not allowed from specific cells, such an impermissible action does not induce any change and thus the state change is 0 in both dimensions.

$$\Sigma = \{(+1, 0), (-1, 0), (0, +1), (0, -1), (0, 0)\}$$

As described in Section 3.1, the robot maintains the set of visited states Ψ during the training phase. Additionally, after each transition, it also updates the action-state change table (Ω), a 4×5 matrix. Since this is a grid-based environment, the Manhattan distance is used as the distance metric here.

Each action and the corresponding consequence determines a reward, and this is used in estimation of Q-values and to learn the best action that can be performed from each state. Achieving the goal has a reward of +1000, entering an illegitimate state offers a reward of -1000, any action within the safe space has a reward of -1, and any action outside provides a reward of -5.

Action	**(1,0)**	**(-1,0)**	**(0,1)**	**(0,-1)**	**(0,0)**
N	0	6972	0	0	288
S	8332	0	0	0	212
E	0	0	9146	0	201
W	0	0	0	7718	196

Table 1: An Example of Ω Table in Maze

A sample of the action-state change table Ω is shown in Table 1. For example, for the action N moving towards North, the row number of the state decreases by 1. This can also be observed from Table 1, where $P((-1,0)|N) = 0.97$ by (6).

Consider an event where the robot is at an unsafe state, say (2,12). It now iterates through its set of visited states Ψ and finds the state (6,12) which is the closest safe state. The difference between the two states is (4,0). The state change σ_x that will move the robot towards safety is (1,0), and the action most likely to induce that is calculated from Ω, which is to move South (S) because $P((1,0)|S) = 0.97$ by (6). The same sequence of steps is repeated until it is at a safe state as also shown in Algorithm 1.

4 Performance Improvement using Self Stabilization

We use the standard Lunar Lander environment based on the well-known Atari game; it is also a part of the OpenAI Gym [24]. Here, state is represented by an 8-tuple, $\langle X, Y, V_x, V_y, \theta, \omega, l_1, l_2 \rangle$, where X, Y correspond to the x and y coordinates of the lander and V_x, V_y denote the velocity components along the x and y axes respectively. The angle of rotation,and the angular velocity are denoted by θ and ω. The last two attributes l_1, l_2 indicate if the legs of the lander are in contact with the ground. The state space is continuous here, unlike the maze environment. The robot is trained to land at its destination $(0, 0)$. The discrete action space $\mathbf{A}$ is given by:

$$\mathbf{A} = \{N, L, M, R\}$$

The first action N corresponds to no action where the lander is subject to gravity only, while the other three actions correspond to firing the left, main, and right engines respectively to navigate the lander.

States are continuous here, and hence maintaining an action-state change table with all possible values of state change is not feasible. So, we define state change as positive and negative with respect to each attribute. Therefore, Σ is a set with $2 \times 8 = 16$ possible state changes. For illustration purposes we show only the relevant part of the action-state change table Ω for this environment in Table 2. For example, by (6), it can be seen that firing the left engine mostly results in a decrease in V_x, firing the right engine most likely corresponds to an increase in V_x, and firing the main engine is most likely to bring about an increase in V_y. All these inferences based on the action-state change table are true based on the definitions of

	X		Y		V_x		V_Y	
	+	−	+	−	+	−	+	−
N	337	1103	596	844	637	803	3.0	1437.0
L	360	745	297	808	3	1102	21.0	1084.0
M	1286	3170	1827	2629	2226	2230	4066.0	390.0
R	299	824	179	944	1119	4	7.0	1116.0

Table 2: An Example of Ω in Lunar Lander

these actions, thus showing that our model can learn the action-state change relationship accurately even in complex environments.

The goal of this experiment is to determine if the self stabilization strategy helps achieve better scores using goal-directed artificial safety constraints. This environment also serves as an example of applying the self-stabilization strategy in a complex environment with continuous states. We create a simple artificial constraint based on the lander's position on the x-axis. We introduce a safety threshold λ, such that the lander is now considered safe only when $-\lambda \leq X \leq +\lambda$. As this is a complex environment where each action impacts more than one attribute, performing self-stabilizing actions each instant could cause unintended consequences on other attributes. Hence, different frequencies of self-stabilizing nudges are considered. The stabilizing nudges are applied once every ν time steps, $\nu \geq 1$.

5 Results

In the Maze environment, the trained model is deployed and played for 5000 episodes. Three different scenarios are considered—no perturbations, RL-only, and RL+SS. The case of no perturbations is considered to measure the rewards in an ideal situation where the robot is always safe, and is never perturbed. For the other two cases, perturbation is handled by a probabilistic model, and the probability of a random perturbation in any timestep is defined as the perturbation probability. It is simply to control the frequency of perturbations, and a value of 0.05 is chosen as an example; it is not critical to the simulations. Increasing the perturbation probability would exacerbate the issues for unstabilised (RL-only) robots, but for the robots using RL+SS approach, the results do not change significantly as any perturbation is always eventually stabilized. As the perturbation probability is reduced, the scores improve in both cases and as it tends to zero, it

corresponds to the case of no perturbations.

In the case of only RL, the robot has no information on the Q-table to make an informed decision about the next move. Two alternatives are available to the robot—to remain stationary or to make random moves. The consequences of remaining idle are fairly obvious, and hence we consider only the random move strategy in this scenario. The RL+SS scenario is the case of using the strategy that has been explained in Section 3.

5.1 Performances of the Three Strategies in Maze

Metric	**RL+SS**	**RL Only**	**No perturbations**
Max	1000	1000	1000
Min	424	-1977	938
Mean	946	-80	973
Std deviation	54	1035	19
Recovery [%]	100 %	36 %	N/A

Table 3: Aggregate statistics in Maze

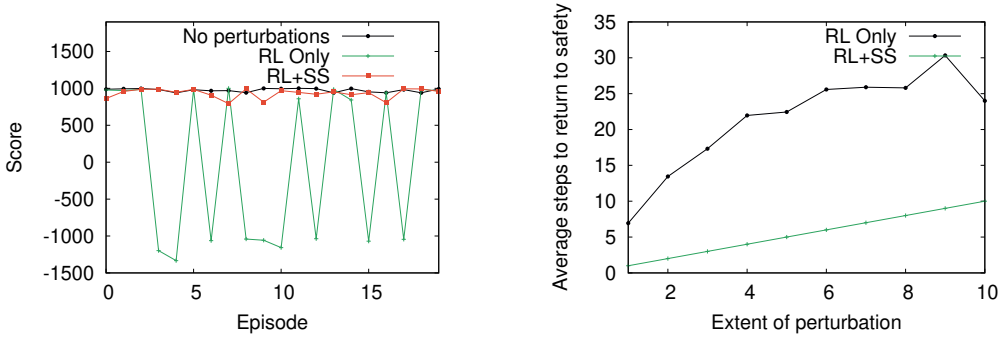
Although the robot is deployed for 5000 episodes, in Figure 1a, only a set of 20 episodes is depicted for the sake of brevity. The aggregate statistics across all 5000 episodes are shown in Table 3. The RL+SS approach guarantees recovery (100%), whereas the RL-only approach was able to recover only in 36% of the cases of perturbations. Thus, this suggests that stabilization is guaranteed by the RL+SS strategy in Maze.

From Table 3, it can be seen that the mean scores also reflect on the benefits of RL+SS (946) over the RL-only strategy (-80). This is primarily because when the RL-only strategy is used, the robot does not recover 64% of the time and end up in illegal states which have high negative payoffs. On the other hand, the RL+SS approach ensures the robot never enters an illegal state, and also achieves the goal, thus enabling higher positive payoffs.

5.2 Extent of Perturbation Versus Recovery Time in Maze

The frequency of perturbation is handled by a probabilistic model, but the extent of perturbation is random. The extent of perturbation is the minimum distance between the perturbed state and a safe state. Figure 1b depicts the relationship between the extent of perturbation and the recovery time with both strategies. In RL+SS, the number of steps taken to

recover is exactly equal to the extent of perturbation, hence we see a perfect linear relationship in this case. In the RL-only strategy, the moves are random, and hence there is no such relationship shown. However, it is clear that the curve corresponding to the RL-only strategy is always above the RL+SS line, showing that recovery, even when possible, is slower.



(a) Scores in Maze

(b) Effect of perturbation on recovery time in Maze

Figure 1: Maze scores

5.3 Performance Improvement using RL+SS in Lunar Lander

The values of the artificial safety constraint λ and stabilization period ν are varied and the results are observed. A 5% improvement is observed with a moderate value of λ and ν. It is also clear that extremely low values of stabilization period and safety thresholds result in poor rewards. Also, very high values of these two parameters give scores that are nearly same as the scores achieved by the RL-only strategy. This is expected and fairly intuitive, since the RL-only approach is basically RL+SS with infinite stabilization period and safety thresholds. However, in the moderate range, this approach offers better scores than the RL-only strategy as seen in Figure 2.

6 Conclusion

Safety of RL robots in deployment remains an unexplored idea, despite wide agreement on its importance and relevance. We show that self-stabilization, a popular paradigm in distributed computing, can be used with RL to tackle

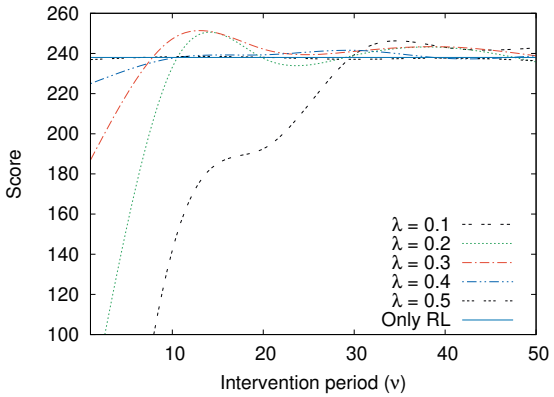


Figure 2: Performance Improvement by RL+SS in Lunar Lander

this challenge. We define a recoverability condition, which if satisfied can guarantee the stabilization of the robot.

Self-stabilization is implemented by learning the relationship between actions and state changes during training, and applying this information to stabilize when unsafe. We describe the design of an RL robot that can stabilize itself and implement it in a simulated environment. It is observed that robots with self-stabilization always recover from a perturbation, but robots trained with only RL can recover in a fraction of the cases only. Results also indicate that a linear relationship exists between the extent of perturbation and recovery time in the case of robots with self-stabilization. The recovery times are also significantly lower than those of robots trained with standard RL. The same approach was implemented in another environment with no issues of safety, to understand if there were any performance benefits. It is observed that an appropriate choice of goal-directed artificial safety constraint improves performance slightly.

This idea of self-stabilization in RL could also be extended to more complicated environments and could find applications in many RL solutions currently deployed in an industrial setting. Further, the same idea could also be expanded to support self-stabilization and safety during training as well.

References

[1] R. S. Sutton and A. G. Barto, *Introduction to Reinforcement Learning*, 1st ed. Cambridge, MA, USA: MIT Press, 1998.

[2] T. A. Mann and Y. Choe, "Scaling up reinforcement learning through targeted exploration," in *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence*, ser. AAAI'11. AAAI Press, 2011, pp. 435–440. [Online]. Available: http://dl.acm.org/citation.cfm?id=2900423.2900492

[3] S. Gu, E. Holly, T. Lillicrap, and S. Levine, "Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates," in *2017 IEEE International Conference on Robotics and Automation (ICRA)*, May 2017, pp. 3389–3396.

[4] A. E. Sallab, M. Abdou, E. Perot, and S. Yogamani, "Deep reinforcement learning framework for autonomous driving," *Electronic Imaging*, vol. 2017, no. 19, pp. 70–76, 2017.

[5] I. Arel, C. Liu, T. Urbanik, and A. G. Kohls, "Reinforcement learning-based multi-agent system for network traffic signal control," *IET Intelligent Transport Systems*, vol. 4, no. 2, pp. 128–135, June 2010.

[6] Z. Zhou, X. Li, and R. N. Zare, "Optimizing chemical reactions with deep reinforcement learning," *ACS Central Science*, vol. 3, no. 12, pp. 1337–1344, Dec 2017. [Online]. Available: https://doi.org/10.1021/acscentsci.7b00492

[7] M. El-Shamouty, X. Wu, S. Yang, M. Albus, and M. F. Huber, "Towards safe human-robot collaboration using deep reinforcement learning," in *2020 IEEE International Conference on Robotics and Automation (ICRA)*, 2020, pp. 4899–4905.

[8] J. Achiam, D. Held, A. Tamar, and P. Abbeel, "Constrained policy optimization," in *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ser. ICML'17. JMLR.org, 2017, pp. 22–31. [Online]. Available: http://dl.acm.org/citation.cfm?id=3305381.3305384

[9] G. Dalal, K. Dvijotham, M. Vecerík, T. Hester, C. Paduraru, and Y. Tassa, "Safe exploration in continuous action spaces," *CoRR*, vol. abs/1801.08757, 2018.

[10] Y. Chow, O. Nachum, E. Duenez-Guzman, and M. Ghavamzadeh, "A lyapunov-based approach to safe reinforcement learning," in *Proceedings of the 32Nd International Conference on Neural Information Processing Systems*, ser. NIPS'18. USA: Curran

Associates Inc., 2018, pp. 8103–8112. [Online]. Available: http://dl.acm.org/citation.cfm?id=3327757.3327904

[11] F. Berkenkamp, M. Turchetta, A. P. Schoellig, and A. Krause, "Safe model-based reinforcement learning with stability guarantees," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17. USA: Curran Associates Inc., 2017, pp. 908–919. [Online]. Available: http://dl.acm.org/citation.cfm?id=3294771.3294858

[12] T. Pham, G. D. Magistris, and R. Tachibana, "Optlayer - practical constrained optimization for deep reinforcement learning in the real world," *CoRR*, vol. abs/1709.07643, 2017.

[13] C. Gehring and D. Precup, "Smart exploration in reinforcement learning using absolute temporal difference errors," in *Proceedings of the 2013 International Conference on Autonomous Agents and Multi-agent Systems*, ser. AAMAS '13. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, 2013, pp. 1037–1044. [Online]. Available: http://dl.acm.org/citation.cfm?id=2484920.2485084

[14] B. Thananjeyan, A. Balakrishna, U. Rosolia, F. Li, R. McAllister, J. E. Gonzalez, S. Levine, F. Borrelli, and K. Goldberg, "Safety augmented value estimation from demonstrations (saved): Safe deep model-based rl for sparse cost robotic tasks," *IEEE Robotics and Automation Letters*, vol. 5, no. 2, pp. 3612–3619, 2020.

[15] D. R. R. Scobee, "Approaches to safety in inverse reinforcement learning," Ph.D. dissertation, U.C. Berkeley, Berkeley, CA, 2020. [Online]. Available: https://escholarship.org/uc/item/6j34r5tp

[16] J. García and F. Fernández, "A comprehensive survey on safe reinforcement learning," *J. Mach. Learn. Res.*, vol. 16, no. 1, pp. 1437–1480, Jan. 2015. [Online]. Available: http://dl.acm.org/citation.cfm?id=2789272.2886795

[17] O. Andersson and P. Doherty, "Deep RL for Autonomous Robots: Limitations and Safety Challenges," in *European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning (ESANN 2019)*, Bruges, Belgium, Apr. 2019. [Online]. Available: https://www.elen.ucl.ac.be/Proceedings/esann/esannpdf/es2019-97.pdf

[18] E. Ahn, "Towards safe reinforcement learning in the real world," Master's thesis, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, July 2019. [Online]. Available: https://www.ri.cmu.edu/wp-content/uploads/2019/08/MSR_Thesis_-Edward_Ahn_2019.pdf

[19] D. Amodei, C. Olah, J. Steinhardt, P. F. Christiano, J. Schulman, and D. Mané, "Concrete problems in AI safety," *CoRR*, vol. abs/1606.06565, 2016.

[20] E. W. Dijkstra, "Self stabilizing systems in spite of distributed control," *Comm. of the ACM*, vol. 17, no. 11, pp. 643–644, 1974.

[21] P. Jalote, *Fault Tolerance in Distributed Systems.* Prentice-Hall, 1994.

[22] A. Arora and M. G. Gouda, "Closure and convergence: a foundation of fault-tolerant computing," *IEEE Trans. Softw. Eng.*, vol. 19, pp. 1015–1027, Nov. 1993.

[23] M. Chan, "Gym maze," https://github.com/MattChanTK/gym-maze, 2016.

[24] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.