Neural Simplex Architecture

Dung T. Phan[1] *, Radu Grosu[2], Nils Jansen[3], Nicola Paoletti[4], Scott A. Smolka[1], and Scott D. Stoller[1]

[1] Department of Computer Science, Stony Brook University, USA
[2] Department of Computer Engineering, Technische Universität Wien, Austria
[3] Department of Software Science, Radboud University, Nijmegen, The Netherlands
[4] Department of Computer Science, Royal Holloway, University of London, UK

Abstract. We present the *Neural Simplex Architecture* (NSA), a new approach to runtime assurance that provides safety guarantees for neural controllers (obtained e.g. using reinforcement learning) of autonomous and other complex systems without unduly sacrificing performance. NSA is inspired by the Simplex control architecture of Sha et al., but with some significant differences. In the traditional approach, the advanced controller (AC) is treated as a black box; when the decision module switches control to the baseline controller (BC), the BC remains in control forever. There is relatively little work on switching control back to the AC, and there are no techniques for correcting the AC's behavior after it generates a potentially unsafe control input that causes a failover to the BC. Our NSA addresses both of these limitations. NSA not only provides safety assurances in the presence of a possibly unsafe neural controller, but can also improve the safety of such a controller in an online setting via retraining, without overly degrading its performance. To demonstrate NSA's benefits, we have conducted several significant case studies in the continuous control domain. These include a target-seeking ground rover navigating an obstacle field, and a neural controller for an artificial pancreas system.

Keywords: Runtime assurance · Simplex architecture · Online retraining · Reverse switching · Safe reinforcement learning.

1 Introduction

Deep neural networks (DNNs) in combination with *reinforcement learning* (RL) are increasingly being used to train powerful *AI agents*. Such agents have achieved unprecedented success in strategy games, including defeating the world champion in Go [30] and surpassing state-of-the-art chess and shogi engines [29]. For these agents, safety is not an issue: when a game-playing agent makes a mistake, the worst-case scenario is losing a game. The same cannot be said for AI agents that control autonomous and other complex systems. A mistake by an AI controller may cause physical damage to the controlled system and its environment, including humans.

[*] Corresponding author. Email: dphan@cs.stonybrook.edu

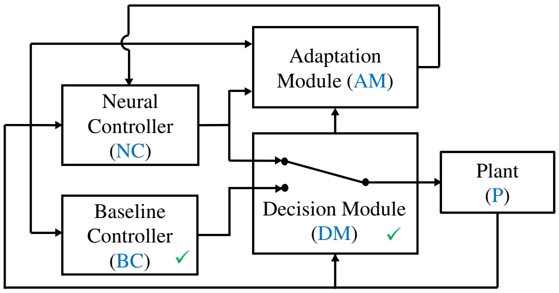

Fig. 1. The Neural Simplex Architecture. The green check marks indicate pre-certified components.

In this paper, we present the *Neural Simplex Architecture* (NSA), a new approach to runtime assurance that provides safety guarantees for AI controllers, including neural controllers, of autonomous and other complex systems without unduly sacrificing performance. NSA is inspired by Sha et al.'s Simplex control architecture [28, 26], where a pre-certified *decision module* (DM) switches control from a high-performance but unverified (hence potentially unsafe) *advanced controller* (AC) to a verified-safe *baseline controller* (BC) if the AC produces an *unrecoverable action*; i.e., an action that would lead the system within one time step to a state from which the BC is not guaranteed to preserve safety.

In the traditional Simplex approach, the AC is treated as a black box, and after the DM switches control to the BC, the BC remains in control forever. There is, however, relatively little work on switching control back to the AC [18, 10, 34], and there are no techniques to correct the AC after it generates an unrecoverable control input.

NSA, illustrated in Fig. 1, addresses both of these limitations. The high-performance *Neural Controller* (NC) is a deep neural network (DNN) that given a plant state (or raw sensor readings), produces a control input for the plant. NSA's use of an NC, as opposed to the black-box AC found in traditional Simplex, allows for online retraining of the NC's DNN. Such retraining is performed by NSA's *Adaptation Module* (AM) using RL techniques. For systems with large state spaces, it may be difficult to achieve thorough coverage during initial training of the NC. Online retraining has the advantage of focusing the learning on areas of the state space that are relevant to the actual system behavior, i.e., regions of the state space the system actually visits.

The AM seeks to eliminate unrecoverable actions from the NC's behavior, without unduly degrading its performance, and in some cases actually improving its performance. While the BC is in control of the plant, the NC runs in shadow mode and is actively retrained by the AM. The DM can subsequently switch control back to the NC with high confidence that it will not repeat the same mistakes, permitting the mission to continue under the auspices of the high-performance NC. Note that because NSA preserves the basic principles of Simplex architecture, it guarantees that the safety of the plant is never violated.

NSA addresses the problem of *safe reinforcement learning* (SRL) [15, 38]. In particular, when the learning agent (the NC) produces an unrecoverable action, the AM uses that action as a training sample (but does not execute it), with a large negative reward. A comparison with related approaches to SRL is provided in Section 6.

We conducted an extensive evaluation of NSA on several significant example systems, including a target-seeking rover navigating through an obstacle field, and a neural controller for an artificial pancreas. Our results on these case studies conclusively demonstrate NSA's benefits.

In summary, the main contributions of this paper are the following:

- We introduce the Neural Simplex Architecture, a new approach to runtime assurance that provides safety guarantees for neural controllers.
- We address two limitations of the traditional Simplex approach, namely lack of established guidelines for switching control back to the AC so that mission completion can be attained; and lack of techniques for correcting the AC's behavior after a failover to the BC, so that reverse switching makes sense in the first place.
- We provide a key insight into safe reinforcement learning (by demonstrating the utility of potentially unsafe training samples, when appropriately and significantly penalized), along with a thorough evaluation of the NSA approach on two significant case studies.

2 Background

Simplex Architecture. The main components of the Simplex architecture (AC, BC, DM) were introduced above. The BC is certified to guarantee the safety of the plant only if it takes over control while the plant's state is within a *recoverable region* $\mathcal{R}_{BC}$. For example, consider the BC for a ground rover that simply applies maximum deceleration a_{max}. The braking distance to stop the rover from a velocity v is therefore $d_{br}(v) = v^2/(2{\cdot}a_{max})$. The BC can be certified to prevent the rover from colliding with an obstacle if it takes over control in a state where $d_{br}(v)$ is less than the minimum distance d_{min} to any obstacle. The set of such states is the recoverable region of this BC.

A control input is called *recoverable* if it keeps the plant inside $\mathcal{R}_{BC}$ within the next time step. Otherwise, the control input is called *unrecoverable*. The DM switches control to the BC when the AC produces an unrecoverable control input. The DM's *switching condition* determines whether a control input is unrecoverable. We also refer to it as the *forward switching condition* (FSC) to distinguish it from the condition for *reverse switching*, a new feature of NSA.

Techniques to determine the FSC include: (i) shrink $\mathcal{R}_{BC}$ by an amount equal to a time step times the maximum gradient of the state with respect to the control input; then classify any control input as unrecoverable if the current state is outside this smaller region; (ii) simulate a model of the plant for one time step if the model is deterministic and check whether the plant strays from $\mathcal{R}_{BC}$; (iii) compute a set of states reachable within one time step and determine whether the reachable set contains states outside $\mathcal{R}_{BC}$.

Reinforcement Learning. Reinforcement learning [32] deals with the problem of how an *agent* learns which sequence of *actions* to take in a given *environment* such that a cumulative *reward* is maximized. At each time step t, the agent receives observation s_t (the environment state) and reward r_t from the environment and takes action a_t. The environment receives action a_t and emits observation s_{t+1} and reward r_{t+1} in response. In the control of autonomous systems, the agent represents the controller, the environment represents the plant, and the state and action spaces are typically continuous.

The goal of RL is to learn a *policy* $\pi(a \mid s)$, i.e., a way of choosing an action a having observed s, that maximizes the expected *return* from the initial state, where the return at time t is defined as the discounted sum of future rewards from t (following policy π): $R_t = \sum_{k=t}^{\infty} \gamma^{k-t} r_{k+1}$; here $\gamma \in [0, 1]$ is a discount factor. For this purpose, RL algorithms typically involve estimating the action-value function $Q^\pi(s, a) = \mathbb{E}[R_t \mid s_t = s, a]$, i.e., the expected return for selecting action a in state s and then always following policy π; and the state-value function $V^\pi(s) = \mathbb{E}[R_t \mid s_t = s]$, i.e., the expected return starting from s and following π.

While early RL algorithms were designed for discrete state and action spaces, recent *deep RL* algorithms, such as TRPO [25], DDPG [19], A3C [21], and ACER [36], have emerged as promising solutions for RL-based control problems in continuous domains. These algorithms leverage the expressiveness of deep neural networks (DNNs) to represent policies and value functions.

3 Neural Simplex Architecture

In this section, we discuss the main components of NSA, namely the neural controller (NC), the adaptation module (AM), and the reverse switching logic. These components in particular are not found in the Simplex control architecture.

The dynamics of the plant, i.e., the system under control, is given by $s_{t+1} = f(s_t, a_t)$, where $s_t \in \mathcal{S}$ is the state of the plant at time t, $\mathcal{S} \subseteq \mathbb{R}^n$ is the real-valued state space, f is a possibly nonlinear function, and $a_t \in \mathcal{A}$ is the control input to the plant at time t, with $\mathcal{A} \subseteq \mathbb{R}^m$ the action space. This equation specifies a deterministic dynamics, even though our approach equally supports nondeterministic ($s_{t+1} \in f_{nd}(s_t, a_t)$) and stochastic ($s_{t+1} \sim f_{st}(s \mid s_t, a_t)$) plant dynamics. We assume full observability, i.e., that the BC and NC have access to the full state of the system s_t.[5]

We denote with $\mathrm{DM}_t \in \{\mathrm{NC}, \mathrm{BC}\}$ the state of the decision module at time t: $\mathrm{DM}_t = \mathrm{NC}$ ($\mathrm{DM}_t = \mathrm{BC}$) indicates that the neural (baseline) controller is in control. Let a_t^{NC} and a_t^{BC} denote the action computed by the NC and the BC, respectively. The final action a_t performed by the NSA agent depends on the DM state: $a_t = a_t^{\mathrm{NC}}$ if $\mathrm{DM}_t = \mathrm{NC}$, $a_t = a_t^{\mathrm{BC}}$ if $\mathrm{DM}_t = \mathrm{BC}$.

Let β be the BC's control law, i.e., $a_t^{\mathrm{BC}} = \beta(s_t)$. For a set of unsafe states $\mathcal{U} \subseteq \mathcal{S}$, the *recoverable region* is the largest set $\mathcal{R}_{BC}$ such that $s \in \mathcal{R}_{BC} \Rightarrow$

[5] In case of partial observability, the full state can typically be reconstructed from sequences of past states and actions, but this process is error-prone.

$f(s, \beta(s)) \in \mathcal{R}_{BC}$ and $\mathcal{R}_{BC} \cap \mathcal{U} = \emptyset$. For $s \in \mathcal{S}$, $a \in \mathcal{A}$, the forward switching condition must satisfy $f(s, a) \notin \mathcal{R}_{BC} \Rightarrow \text{FSC}(s, a)$.

The Neural Controller. The NC is represented by a DNN-based policy π_{θ_t}, where θ_t are the current DNN parameters. The policy maps the current state into a proposed action $a_t^{\text{NC}} = \pi_{\theta_t}(s_t)$. We stress the time dependency of the parameters because adaptation and retraining of the policy is a key feature of NSA. As for the dynamics f, our approach supports stochastic policies ($a_t^{\text{NC}} \sim \pi(a \mid s_t, \theta_t)$).

The NC can be obtained using any RL algorithm. We used DDPG with the safe learning strategy of penalizing unrecoverable actions, as discussed in Section 4. DDPG is attractive as it works with deterministic policies, and allows uncorrelated samples to be added to the pool of samples for training or retraining. The latter property is important because it allows us to collect disconnected samples of what the NC would do while the plant is under the BC's control, and to use these samples for online retraining of the NC.

Adaptation and Retraining. The AM is used to retrain the NC in an online manner while the BC is in control of the plant (due to NC-to-BC failover). The main purpose of this retraining is to make the NC less likely to trigger the FSC, thereby allowing it to remain in control for longer periods of time, thereby improving overall system performance.

Techniques that we consider for online retraining of the NC include supervised learning and reinforcement learning. In supervised learning, state-action pairs of the form (s, a) are required for training purposes. The training algorithm uses these examples to teach the NC safe behavior. The control inputs produced by the BC can be used as training samples, although this will train the NC to imitate BC's behavior, which may lead to a loss in performance.

We therefore prefer SRL for online retraining, with a reward function that penalizes unrecoverable actions and rewards recoverable, high-performing ones. The reward function for retraining can be designed as follows.

$$r(s, a, s') = \begin{cases} r_{unrecov}, & \text{if FSC}(s, a) \\ r_{perf}(s, a, s'), & \text{otherwise} \end{cases} \tag{1}$$

where $r_{perf}(s, a, s')$ is a performance-related reward function, and $r_{unrecov}$ is a negative number used to penalize unrecoverable actions. The benefits of this approach to SRL are discussed in Section 4.

The AM retrains the NC at each time step the BC is in control by maintaining a pool of retraining samples of the form $(s_t, a_t^{\text{NC}}, s', r')$, where a_t^{NC} is the NC-proposed action, $s' = f(s_t, a_t^{\text{NC}})$ is the state that the system would evolve to if the NC was in control, and $r' = r(s, a_t^{\text{NC}}, s')$ is the corresponding reward. I.e., samples are obtained by running the NC in shadow mode: when BC is in control, the AM obtains a retraining sample by running a simulation of the system for one time step and applying a_t^{NC}, while the actual system evolves according to the BC action a_t^{BC}.

The AM updates the NC's parameters θ_t as follows:

$$\theta_t = \begin{cases} \mathrm{RL}(\theta_{t-1}, (s_t, a_t^{\mathrm{NC}}, s', r')), & \text{if } \mathrm{DM}_t = \mathrm{BC} \\ \theta_{t-1}, & \text{otherwise} \end{cases}$$

where RL is the deep RL algorithm chosen for NC adaptation. Note that as soon as the DM switches control to the BC after the NC has produced an unrecoverable action (see also the Switching logic paragraph below), a corresponding retraining sample for the NC's action is added to the pool.

We evaluated a number of variants of this procedure by making different choices along the following dimensions.

1. Start retraining with an empty pool of samples or with the pool created during the initial training of the NC.
2. Add (or do not add) exploration noise to NC's action when collecting a sample. With exploration noise, the resulting action is $a_t^{\mathrm{NC}} + \nu_t$, where ν_t is a random noise term. Note that we consider noise only when NC is running in shadow mode (BC in control), as directly using noisy actions to control the plant would degrade performance.
3. Collect retraining samples only while BC is in control or at every time step. In both cases, the action in each training sample is the action output by NC (or a noisy version of it); we never use BC's action in a training sample. Also, in both cases, the retraining algorithm for updating the NC is run only while the BC is in control.

We found that reusing the pool of training samples (DDPG's so-called experience replay buffer) from initial training of the NC helps evolve the policy in a more stable way, as retraining samples gradually replace initial training samples in the sample pool. Another benefit of reusing the initial training pool is that the NC can be immediately retrained without having to wait for enough samples to be collected online. We found that adding exploration noise to NC's actions in retraining samples, and collecting retraining samples at every time step, both increase the benefit of retraining. This is because these two strategies provide more diverse samples and thereby help achieve more thorough exploration of the state-action space.

Switching logic. NSA includes *reverse switching* from the BC to the retrained NC. An additional benefit of well-designed reverse switching is that it lessens the burden on the BC to achieve performance objectives, leading to a simpler BC design that focuses mainly on safety. Control of the plant is returned to the NC when the reverse switching condition (RSC) is true in the current state. We can summarize NSA's switching logic by describing the evolution of the DM state DM_t. NSA begins with the NC in control, i.e., $\mathrm{DM}_t = \mathrm{NC}$ for $t \leq 0$. For $t > 0$, the DM state is given by:

$$\mathrm{DM}_t = \begin{cases} \mathrm{BC}, & \text{if } \mathrm{DM}_{t-1} = \mathrm{NC} \text{ and } \mathrm{FSC}(s_t, a_t^{\mathrm{NC}}) \\ \mathrm{NC}, & \text{if } \mathrm{DM}_{t-1} = \mathrm{BC} \text{ and } \mathrm{RSC}(s_t) \\ \mathrm{DM}_{t-1}, & \text{otherwise} \end{cases}$$

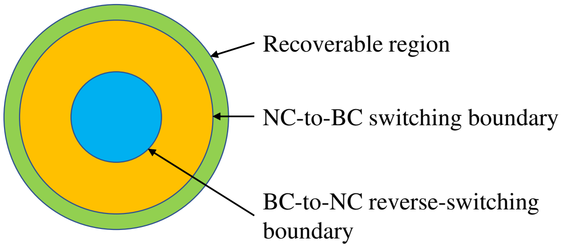
Recoverable region
NC-to-BC switching boundary
BC-to-NC reverse-switching boundary

Fig. 2. Switching boundaries. The blue region is a subset of the orange area, which in turn is a subset of the green region.

To ensure safety when returning control to the NC, the FSC must not hold if the RSC is satisfied, i.e., $\mathrm{RSC}(s) \Rightarrow \neg\mathrm{FSC}(s,a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}$.

We seek to develop reverse switching logic that returns control to NC when it is safe to do so and which avoids frequent back-and-forth switching between the BC and NC. We propose two such approaches. One is to reverse-switch if a forward switch will not occur in the near future. This can be checked by simulating the composition of the NC and plant for T time steps, and reverse-switching if the FSC does not hold within this time horizon.[6] Formally, $\mathrm{RSC}(s_t) = \bigwedge_{t'=t}^{t+T} \neg\mathrm{FSC}(s'_{t'}, \pi_{\theta_t}(s'_{t'}))$, where $s'_t = s_t$ and $s'_{t'+1} = f(s'_{t'}, \pi_{\theta_t}(s'_{t'}))$. This approach, used in our inverted pendulum and artificial pancreas case studies, prevents frequent switching.

A simpler approach is to reverse-switch if the current plant state is sufficiently far from the NC-to-BC switching boundary; see Fig. 2. Formally, $\mathrm{RSC}(s_t) = \sup\{d(s_t, s') \mid s' \in \mathcal{S},\ \mathrm{FSC}(s', \pi_{\theta_t}(s'))\} > \epsilon$, where d is a metric on $\mathbb{R}^n$ and $\epsilon \in \mathbb{R}^+$ is the desired distance. This approach is used in our rover case study.

We emphasize that the choice of RSC does not affect safety and is application-dependent. Note that both of our approaches construct an RSC that is stricter than a straight complement of the FSC. This helps avoid excessive switching. In our experiments, we empirically observed that the system behavior was not very sensitive to the exact value of T or ϵ; so choosing acceptable values for them is not difficult.

4 Safe Reinforcement Learning with Penalized Unrecoverable Continuous Actions

We evaluate the use of two policy-gradient algorithms for safe reinforcement learning in NSA. The first approach filters the learning agent's unrecoverable actions before they reach the plant. For example, when the learning agent, i.e., the NC, produces an unrecoverable action, a runtime monitor [13] or a preemptive shield [3] replaces it with a recoverable one to continue the trajectory. The

[6] For nondeterministic (stochastic) systems, a (probabilistic) model checker can be used instead of a simulator, but this approach may be computationally expensive.

recoverable action is also passed to the RL algorithm to update the agent and training continues with the rest of the trajectory.

In the second approach, when the learning agent produces an unrecoverable action, we assign a penalty (negative reward) to the action, use it as a training sample, and then use recoverable actions to safely terminate the trajectory (but not to train the agent). Safely terminating the trajectory is important in cases where for example the live system is used for training. We call this approach *safe reinforcement learning with penalized unrecoverable continuous actions* (SRL-PUA). By "continuous" here we mean real-valued action spaces, as in [9]. Other SRL approaches such as [2] use discrete actions.

To compare the two approaches, we used the DDPG and TRPO algorithms to train neural controllers for an inverted pendulum (IP) control system. Details about our IP case study, including the reward function and the BC used to generate recoverable actions, can be found in [24].

We used the implementations of DDPG and TRPO in rllab [11]. For TRPO, we trained two DNNs, one for the mean and the other for the standard deviation of a Gaussian policy. Both DNNs have two fully connected hidden layers of 32 neurons each and one output layer. The hidden layers use the `tanh` activation function, and the output layer is linear. For DDPG, we trained a DNN that computes the action directly from the state. The DNN has two fully connected hidden layers of 32 neurons each and one output layer. The hidden layers use the `ReLU` activation function, and the output layer uses `tanh`. We followed the choice of activation functions in the examples accompanying rllab.

For each algorithm, we ran two training experiments. In one experiment, we reproduce the filtering approach; i.e., we replace an unrecoverable action produced by the learning agent with the BC's recoverable action, use the latter as the training sample, and continue the trajectory. We call this training method SRL-BC. In the other experiment, we evaluate the SRL-PUA approach. Note that both algorithms explore different trajectories by resetting the system to a random initial state whenever the current trajectory is terminated. We set the maximum trajectory length to 500 time steps, meaning that a trajectory is terminated when it exceeds 500 time steps.

We trained the DDPG and TRPO agents on a total of one million time steps. After training, we evaluated all trained policies on the same set of 1,000 random initial states. During evaluation, if an agent produces an unrecoverable action, the trajectory is terminated. The results are shown in Table 1. For both algorithms, the policies trained with recoverable actions (the SRL-BC approach) produce unrecoverable actions in all test trajectories, while the SRL-PUA approach, where the policies are trained with penalties for unrecoverable actions, does not produce any such actions. As such, the latter policies achieve superior returns and trajectory lengths (they are able to safely control the system the entire time).

In the above experiments, we replaced unrecoverable actions with actions generated by a deterministic BC, whereas the monitoring [13] and preemptive shielding [2] approaches allow unrecoverable actions to be replaced with random recoverable ones, an approach we refer to as SRL-RND. To show that our conclusions are independent of this difference, we ran one more experiment with each learning algorithm, in which we replaced each unrecoverable action with an action selected by randomly generating actions until a recoverable one is found. The results, shown in Table 2, once again demonstrate that training with only recoverable actions is ineffective. Compared to filtering-based approaches (SRL-BC in Table 1 and SRL-RND in Table 2), the SRL-PUA approach yields a 25- to 775-fold improvement in the average return.

	TRPO		**DDPG**	
	SRL-BC	**SRL-PUA**	**SRL-BC**	**SRL-PUA**
Unrec Trajs	1,000	0	1,000	0
Comp Trajs	0	1,000	0	1,000
Avg. Return	112.53	4,603.97	61.52	4,596.04
Avg. Length	15.15	500	14.56	500

Table 1. Policy performance comparison. **SRL-BC**: policy trained with BC's actions replacing unrecoverable ones. **SRL-PUA**: policy trained with penalized unsafe actions. **Unrec Trajs**: # trajectories terminated due to an unrecoverable action. **Comp Trajs**: # trajectories that reach the limit of 500 time steps. **Avg. Return** and **Avg. Length**: average return and trajectory length over 1,000 trajectories.

	TRPO		**DDPG**	
	SRL-RND	**SRL-PUA**	**SRL-RND**	**SRL-PUA**
Unrec Trajs	1,000	0	1,000	0
Comp Trajs	0	1,000	0	1,000
Avg. Return	183.36	4,603.97	5.93	4,596.04
Avg. Length	1.93	500	14	500

Table 2. Policy performance comparison. **SRL-RND**: policy trained with random recoverable actions replacing unrecoverable ones.

5 Case Studies

An additional case study, the Inverted Pendulum, along with further details about the case studies presented in this section can be found in [24].

5.1 Rover Navigation

We consider the problem of navigating a rover to a predetermined target location while avoiding collisions with static obstacles. The rover is a circular disk of radius r. It has a maximum speed v_{max} and a maximum acceleration a_{max}. The maximum braking time is therefore $t_{br\_max} = v_{max}/a_{max}$, and the maximum braking distance is $d_{br\_max} = v_{max}^2/(2 \cdot a_{max})$. The control inputs are the accelerations a_x and a_y in the x and y directions, respectively. The system uses discrete-time control with a time step of dt.

The rover has n distance sensors whose detection range is l_{max}. The sensors are placed evenly around the perimeter of the rover; i.e., the center lines of sight of two adjacent sensors form an angle of $2\pi/n$. The rover can only move forwards, so its orientation is the same as its heading angle. The state vector for the rover is $[x, y, \theta, v, l_1, l_2, ..., l_n]$, where (x, y) is the position, θ is the heading angle, v is the velocity, and the l_i's are the sensor readings.

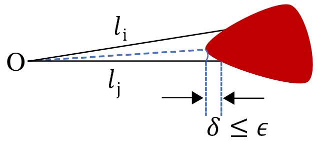

Fig. 3. Illustration of assumptions about obstacle shapes.

We assume the sensors have a small angular field-of-view so that each sensor reading reflects the distance from the rover to an obstacle along the sensor's center line of sight. If a sensor does not detect an obstacle, its reading is l_{max}.

We also assume that when the sensor readings of two adjacent sensors s_i and s_j are l_i and l_j, respectively, then the (conservative) minimum distance to any obstacle point located in the cone formed by the center lines of sight of s_i and s_j is $\min\{l_i, l_j\} - \epsilon$. Here, ϵ is a constant that limits by how much an obstacle can protrude into the blind spot between s_i and s_j's lines of sight; see Fig. 3.

A state s of the rover is *recoverable* if starting from s, the baseline controller (BC) can brake to a stop at least distance d_{safe} from any obstacle. Let the braking distance in state s be $d_{br}(s) = v^2/(2 \cdot a_{max})$, where v is the rover's speed in s. Then s is recoverable if the minimum sensor reading l_{min} in state s is at least $d_{safe} + d_{br}(s) + \epsilon$.

The FSC holds when the control input u_{NC} proposed by the NC will put the rover in an unrecoverable state in the next time step. We check this condition by simulating the rover for one time step with u_{NC} as the control input, and by then determining if $l_{min} < d_{safe} + d_{br}(s) + \epsilon$.

The RSC is $l_{min} \geq m \cdot v_{max} \cdot dt + d_{safe} + d_{br\_max} + \epsilon$, ensuring that the FSC does not hold for the next $m - 1$ time steps. Parameter m can be chosen to reduce excessive back-and-forth switching between the NC and BC.

The BC performs the following steps: 1) Apply the maximum braking power a_{max} until the rover stops. 2) Randomly pick a safe heading angle θ based on the current position and sensor readings. 3) Rotate the rover until its heading angle is θ. 4) Move with heading angle θ until either the FSC becomes true (this is checked after each time step by the BC itself), in which case the BC is re-started at Step 1, or the RSC becomes true (this is checked by the DM), in which case the NC takes over.

Experimental Results. Parameter values used: $r = 0.1\,\text{m}$, $v_{max} = 0.8\,\text{m/s}$, $a_{max} = 1.6\,\text{m/s}^2$, $l_{max} = 2\,\text{m}$, $n = 32$, $d_{safe} = 0.2\,\text{m}$, $\epsilon = 0.01\,\text{m}$, $m = 5$, $dt = 0.1\,\text{s}$. The target is a circular disk at location $(0, 0)$ with a radius of 0.1m.

The obstacle field, which is fixed during training and testing, consists of 12 circular obstacles with a minimum radius of 0.25m. Rover initial position (x_0, y_0) is randomized in the area $[-5, 5] \times [-5, 5]$. [7] We assume that the sensor field-of-view is at least 7.25°, thereby satisfying the assumption that an obstacle does not protrude more than ϵ into the blind spot between adjacent sensors. See also Fig. 3. The NC is a DNN with two `ReLU` hidden layers, each of size 64, and a `tanh` output layer. We used the DDPG algorithm for both initial training and online retraining of the NC. For initial training, we ran DDPG for 5 million time steps. The reward function for initial training and online retraining is:

$$r(s, a, s') = \begin{cases} -20,000, & \text{if FSC}(s, a) \\ 10,000, & \text{if DT}(s) \leq 0.2 \\ -1 - 20 \cdot \text{DT}(s), & \text{otherwise} \end{cases} \tag{2}$$

where $\text{FSC}(s, a)$ is the forward switching condition and $\text{DT}(s)$ is the center-to-center distance from the rover to the target in state s. The rover is considered to have reached the target if $\text{DT}(s) \leq 0.2$, as, recall, the target is a circular disk with radius of 0.1m and the radius r of the rover is 0.1m. If the action a triggers the forward switching logic, it is penalized by assigning it a negative reward of -20,000. If a causes the rover to reach the target, it receives a positive reward of 10,000. All other actions are penalized by an amount proportional to the distance to the target, encouraging the agent to reach the target quickly.

Our experiments with online retraining use the same DDPG settings as in initial training, except that we initialize the AM's pool of retraining samples with the pool created by initial training, instead of an empty pool. The pool created by initial training contains one million samples; this is the maximum pool size, which is a parameter of the algorithm. When creating retraining samples, the AM adds Gaussian noise to the NC's actions. The NC's actions are collected (added to the pool) at every time step, regardless of which controller is in control; thus, the AM also collects samples of what the NC would do while the BC is in control.

We ran the NSA instance starting from 10,000 random initial states. Out of 10,000 trajectories, forward switching occurred in 456 of them. Of these 456 trajectories, the BC was in control for a total of 70,974 time steps. This means there were 70,974 ($\sim$71K) retraining updates to the NC. To evaluate the benefits of online retraining, we compared the performance of the NC after initial training and after 20K, 50K, and 71K online updates. We evaluated the performance of each of these controllers (by itself, without NSA) by running it from the same set of 1,000 random initial states.

The results in Table 3 show that after 71K retraining updates, the NC outperforms the initially trained version on every metric. Table 3 also shows that the NC's performance increases with the number of retraining updates, thus demonstrating that NSA's online retraining not only improves the safety of the NC, but also its performance.

[7] Although the obstacles are fixed, the NC still generalizes well (but not perfectly) to random obstacle fields not seen during training, as shown in this video https://youtu.be/ICT8D1uniIw.

	IT	**20K RT**	**50K RT**	**71K RT**
FSCs	100	79	43	8
Timeouts	35	49	50	22
Targets	865	872	907	970
Avg. Ret.	-9,137.3	-9,968.8	-5,314.6	-684.0
Avg. Len.	138.67	142.29	156.13	146.56

Table 3. Benefits of online retraining (∼71K NC updates in total) for ground rover navigation. **IT**: results for initially trained NC. **20K RT**, **50K RT**, **71K RT**: results for NC after 20K, 50K, 71K retraining updates. All controllers evaluated on same set of 1,000 random initial states. **FSCs**: # trajectories in which FSC becomes true. **Timeouts**: # trajectories that reach the limit of 500 time steps without reaching target or having FSC become true. **Targets**: # trajectories that reach the target. **Avg. Ret.** and **Avg. Len.**: average return and average trajectory length over all 1,000 trajectories.

We resumed initial training to see if this would produce similar improvements. Specifically, we continued the initial training for an additional 71K, 1M, and 3M samples. The results, included in [24], show that extending the initial training slowly improves both the safety and performance of the NC but requires substantially more updates. 71K retraining updates provide significantly more benefits than even 3M additional samples of initial training.

5.2 Artificial Pancreas

The artificial pancreas (AP) is used to control blood glucose (BG) levels in Type 1 diabetes patients through automated delivery of insulin. We use the linear plant model of [6] to describe the physiological state of the patient. The main state variable of interest is G, which is the difference between the reference BG (7.8 mmol/L) and the patient's BG. The control action, i.e., the insulin input, is denoted by u. Further details of this model, including its ODE dynamics, can be found in [24].

The AP should maintain BG levels within the safe range of 4 to 11 mmol/L. In particular, it should avoid hypoglycemia (i.e., BG levels below the safe range), which can lead to severe health consequences. Hypoglycemia occurs when the controller overshoots the insulin dose. Insulin control is uniquely challenging because the controller cannot take a corrective action to counteract an excessive dose; its most extreme safety measure is to turn off the insulin pump. Hence, the baseline controller for the AP sets $u = 0$.

We intentionally under-train the initial NC so that it exhibits low performance and produces unrecoverable actions. Low-performing AP controllers may arise in practice for several reasons, e.g., when the training-time model parameters do not match the current real-life patient parameters.

The reward function r is designed to penalize deviations from the reference BG level, as captured by state variable G. We assign a positive reward when G is close to zero (within ± 1), and we penalize larger deviations with a $5\times$ factor for mild hyperglycemia ($1 < G \leq 3.2$), a $7\times$ factor for mild hypoglycemia ($-3.8 \leq G < -1$), $9\times$ for strong hyperglycemia ($G > 3.2$), and $20\times$ for strong hypoglycemia ($G < -3.8$). The other constants are chosen to avoid jump discontinuities in the reward function.

$$r(s,u,s') = \begin{cases} 10 - |G'|, & \text{if } |G'| \leq 1 \\ 14 - 5 \cdot |G'|, & \text{if } 1 < G' \leq 3.2 \\ 26.8 - 9 \cdot |G'|, & \text{if } G' > 3.2 \\ 16 - 7 \cdot |G'|, & \text{if } -3.8 \leq G' < -1 \\ 65.4 - 20 \cdot |G'| & \text{otherwise} \end{cases}$$

where G' is the value of G in state s'.

An AP plant state s is *recoverable* if under the control of the BC, a state where $G' < -3.8$ cannot be reached starting from s. This condition can be checked by simulation. The FSC holds when the NC's action leads to an unrecoverable state in the next time step. For reverse switching, we return control to the NC if the FSC does not hold within time $T = 10$ from the current state.

Experimental Results. To produce an under-trained NC, we used 107,000 time steps of initial training. We ran NSA on the under-trained controller on 10,000 trajectories, each starting from a random initial state. Among the first 400 trajectories, 250 led to forward switching and hence retraining. The retraining was very effective, as forward switching did not occur after the first 400 trajectories.

As in the other case studies we conducted, we then evaluated the benefits of retraining by comparing the performance of the initially trained NC and the retrained NC on trajectories starting from the same set of 1,000 random initial states. The results are given in Table 4. Retraining greatly improves the safety of the NC: the initially trained controller reaches an unrecoverable state in all 1,000 of these trajectories, while the retrained controller never does. The retrained controller's performance is also significantly enhanced, with an average return 2.9 times that of the initial controller.

	Initially Trained	**Retrained**
Unrecov Trajs	1,000	0
Complete Trajs	0	1,000
Avg. Return	824	2,402
Avg. Length	217	500

Table 4. Benefits of retraining for the AP case study. There were 61 updates to the NC. Row labels are as per Table 1.

6 Related Work

The original Simplex architecture did not consider reverse switching. In [27, 26], when the AC produces an unrecoverable action, it is disabled until it is manually re-enabled. It is briefly mentioned in [18] that reverse switching should be performed only when the FSC is false, and that a stricter RSC might be needed to prevent frequent switching, but the paper does not pursue this idea further. A more general approach to reverse switching, which uses reachability analysis to determine if the plant is safe in the next two time steps irrespective of the controller, is presented in [10]. This approach results in more conservative reverse switching conditions, as it does not take the behavior of the AC into account, unlike one of the approaches we propose. The idea of reverse switching when the AC's outputs are stabilized is briefly mentioned in [34].

Regarding approaches to safe reinforcement learning (SRL), we refer the reader to two recent comprehensive literature reviews [15, 38]. Bootstrapping of policies that are known to be safe in certain environments is employed in [31], while [16] restricts exploration to a portion of the state space close to an optimal, pre-computed policy.

In [3], the authors synthesize a *shield* (a.k.a. *post-posed shield*) from a temporal-logic safety specification based on knowledge of the system dynamics. The shield monitors and corrects an agent's actions to ensure safety. This approach targets systems with finite state and action spaces. Suitable finite-state abstractions are needed for infinite-state systems. In [5], the shield-based approach is extended to stochastic systems. In contrast, NSA's policy-gradient-based approach is directly applicable to systems with infinite state spaces and continuous action spaces.

In [13], the authors use formally verified runtime monitors in the RL training phase to constrain the actions taken by the learning agent to a set of safe actions. The idea of using the learned policy together with a known-safe fallback policy in the deployed system is mentioned, but further details are not provided. In contrast, we discuss in detail how the NSA approach guarantees runtime safety and how SRL is is used for online retraining of the NC. In [14], a verification-preserving procedure is proposed for learning updates to the environment model when SRL is used and the exact model is not initially known. The approach to SRL is mainly taken from [13], so again the learned policy is not guaranteed safe. Note that the SRL approach of [13, 14] allows the training algorithm to speculate when the plant model is deviating from reality.

Other approaches to SRL incorporate formal methods to constrain the SRL exploration process. These include the use of (probabilistic) temporal logic [37, 17, 20], ergodicity-based notions of safety [22], and providing probably approximately correct (PAC) guarantees [12]. All of these techniques work on finite state spaces.

In [8], the authors use Lyapunov functions in the framework of constrained Markov decision processes to guarantee policy safety during training. They focus on policy-iteration and Q-learning for discrete state and action problems. Their approach is currently not applicable to policy-gradient algorithms, such as the DDPG algorithm used in our experiments, nor continuous state/action problems. Lyapuanov functions are also used in [4] for SRL, but it likewise cannot be used for policy-gradient algorithms.

In [33], the authors propose Reward Constrained Policy Optimization (RCPO), where a per-state weighted penalty term is added to the reward function. Such weights are updated during training. RCPO is shown to almost surely converge to a solution, but does not address the problem of guaranteeing safety during training. In contrast, we penalize unrecoverable actions and safely terminate the current trajectory to ensure plant safety.

In [1], the authors present the Constrained Policy Optimization algorithm for constrained MDPs, which guarantees safe exploration during training. CPO only ensures approximate satisfaction of constraints and provides an upper bound on the cost associated with constraint violations. In [23], the authors use control barrier functions (CBFs) for SRL. Whenever the learning agent produces an unsafe action, it is minimally perturbed to preserve safety. In contrast, in NSA, when the NC proposes an unsafe action, the BC takes over and the NC is retrained by the AM. CBFs are also used in [7].

Similar to the shield-based method, a safety layer is inserted between the policy and the plant in [9]. Like the CBF approach, the safety layer uses quadratic programming to minimally perturb the action to ensure safety. There are, however, no formal guarantees of safety because of the data-driven linearization of the constraint function.

7 Conclusions

We have presented the Neural Simplex Architecture for assuring the runtime safety of systems with neural controllers. NSA features an adaptation module that retrains the NC in an online fashion, seeking to eliminate its faulty behavior without unduly sacrificing performance. NSA's reverse switching capability allows control of the plant to be returned to the NC after a failover to BC, thereby allowing NC's performance benefits to come back into play. We have demonstrated the utility of NSA on three significant case studies in the continuous control domain.

As future work, we plan to investigate methods for establishing statistical bounds on the degree of improvement that online retraining yields in terms of safety and performance of the NC. We also plan to incorporate techniques from the L1Simplex architecture [35] to deal with deviations of the plant model's behavior from the actual behavior.

Acknowledgments. We thank the anonymous reviewers for their helpful comments. This material is based upon work supported in part by NSF grants CCF-191822, CPS-1446832, IIS-1447549, CNS-1445770, and CCF-1414078, FWF-NFN RiSE Award, and ONR grant N00014-15-1-2208. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of these organizations.

References

1. Achiam, J., Held, D., Tamar, A., Abbeel, P.: Constrained policy optimization. In: International Conference on Machine Learning. pp. 22–31 (2017)

2. Alshiekh, M., Bloem, R., Ehlers, R., Könighofer, B., Niekum, S., Topcu, U.: Safe reinforcement learning via shielding. arXiv preprint arXiv:1708.08611 (2017)
3. Alshiekh, M., Bloem, R., Ehlers, R., Könighofer, B., Niekum, S., Topcu, U.: Safe reinforcement learning via shielding. In: AAAI (2018), https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/17211
4. Berkenkamp, F., Turchetta, M., Schoellig, A., Krause, A.: Safe model-based reinforcement learning with stability guarantees. In: Advances in neural information processing systems. pp. 908–918 (2017)
5. Bouton, M., Karlsson, J., Nakhaei, A., Fujimura, K., Kochenderfer, M.J., Tumova, J.: Reinforcement learning with probabilistic guarantees for autonomous driving. CoRR **abs/1904.07189** (2019)
6. Chen, H., Paoletti, N., Smolka, S.A., Lin, S.: Committed moving horizon estimation for meal detection and estimation in type 1 diabetes. In: American Control Conference (ACC 2019). pp. 4765–4772 (2019)
7. Cheng, R., Orosz, G., Murray, R.M., Burdick, J.W.: End-to-end safe reinforcement learning through barrier functions for safety-critical continuous control tasks. AAAI (2019)
8. Chow, Y., Nachum, O., Duenez-Guzman, E., Ghavamzadeh, M.: A Lyapunov-based approach to safe reinforcement learning. In: Advances in Neural Information Processing Systems. pp. 8103–8112 (2018)
9. Dalal, G., Dvijotham, K., Vecerik, M., Hester, T., Paduraru, C., Tassa, Y.: Safe exploration in continuous action spaces. ArXiv e-prints (Jan 2018)
10. Desai, A., Ghosh, S., Seshia, S.A., Shankar, N., Tiwari, A.: A runtime assurance framework for programming safe robotics systems. In: IEEE/IFIP International Conference on Dependable Systems and Networks (DSN) (Jun 2019)
11. Duan, Y., Chen, X., Houthooft, R., Schulman, J., Abbeel, P.: Benchmarking deep reinforcement learning for continuous control. In: Proceedings of the 33rd International Conference on Machine Learning - Volume 48. pp. 1329–1338. ICML'16 (2016), http://dl.acm.org/citation.cfm?id=3045390.3045531
12. Fu, J., Topcu, U.: Probably approximately correct MDP learning and control with temporal logic constraints. In: 2014 Robotics: Science and Systems Conference (2014)
13. Fulton, N., Platzer, A.: Safe reinforcement learning via formal methods. In: AAAI'18 (2018)
14. Fulton, N., Platzer, A.: Verifiably safe off-model reinforcement learning. In: Vojnar, T., Zhang, L. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 413–430. Springer International Publishing, Cham (2019)
15. García, J., Fernández, F.: A comprehensive survey on safe reinforcement learning. J. Mach. Learn. Res. **16**(1), 1437–1480 (Jan 2015), http://dl.acm.org/citation.cfm?id=2789272.2886795
16. García, J., Fernández, F.: Probabilistic policy reuse for safe reinforcement learning. ACM Transactions on Autonomous and Adaptive Systems (TAAS) **13**(3), 14 (2019)
17. Hasanbeig, M., Abate, A., Kroening, D.: Logically-correct reinforcement learning. CoRR **abs/1801.08099** (2018)
18. Johnson, T., Bak, S., Caccamo, M., Sha, L.: Real-time reachability for verified Simplex design. ACM Trans. Embed. Comput. Syst. **15**(2), 26:1–26:27 (Feb 2016). https://doi.org/10.1145/2723871, http://doi.acm.org/10.1145/2723871
19. Lillicrap, T., Hunt, J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., Wierstra, D.: Continuous control with deep reinforcement learning. arXiv preprint arXiv:1509.02971 (2015)

20. Mason, G., Calinescu, R., Kudenko, D., Banks, A.: Assured reinforcement learning with formally verified abstract policies. In: ICAART (2). pp. 105–117. SciTePress (2017)
21. Mnih, V., Badia, A., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., Kavukcuoglu, K.: Asynchronous methods for deep reinforcement learning. In: ICML. pp. 1928–1937 (2016)
22. Moldovan, T.M., Abbeel, P.: Safe exploration in Markov decision processes. In: ICML. icml.cc / Omnipress (2012)
23. Ohnishi, M., Wang, L., Notomista, G., Egerstedt, M.: Barrier-certified adaptive reinforcement learning with applications to brushbot navigation. IEEE Transactions on Robotics pp. 1–20 (2019). https://doi.org/10.1109/TRO.2019.2920206
24. Phan, D., Paoletti, N., Grosu, R., Jansen, N., Smolka, S.A., Stoller, S.D.: Neural Simplex Architecture. arXiv preprint arXiv:1908.00528 (2019)
25. Schulman, J., Levine, S., Abbeel, P., Jordan, M., Moritz, P.: Trust region policy optimization. In: ICML. pp. 1889–1897 (2015)
26. Seto, D., Krogh, B., Sha, L., Chutinan, A.: The Simplex architecture for safe online control system upgrades. In: Proc. 1998 American Control Conference. vol. 6, pp. 3504–3508 (1998). https://doi.org/10.1109/ACC.1998.703255
27. Seto, D., Sha, L., Compton, N.: A case study on analytical analysis of the inverted pendulum real-time control system (1999)
28. Sha, L.: Using simplicity to control complexity. IEEE Software **18**(4), 20–28 (2001). https://doi.org/10.1109/MS.2001.936213
29. Silver, D., Hubert, T., Schrittwieser, J., et al.: Mastering chess and shogi by self-play with a general reinforcement learning algorithm. arXiv preprint arXiv:1712.01815 (2017)
30. Silver, D., Schrittwieser, J., Simonyan, K., et al.: Mastering the game of Go without human knowledge. Nature **550**(7676), 354 (2017)
31. Simão, T.D., Spaan, M.T.J.: Safe policy improvement with baseline bootstrapping in factored environments. In: AAAI. pp. 4967–4974. AAAI Press (2019)
32. Sutton, R., Barto, A.: Reinforcement Learning: An Introduction. MIT Press, Cambridge (1998)
33. Tessler, C., Mankowitz, D.J., Mannor, S.: Reward constrained policy optimization. ArXiv e-prints (May 2018)
34. Vivekanandan, P., Garcia, G., Yun, H., Keshmiri, S.: A Simplex architecture for intelligent and safe unmanned aerial vehicles. In: 2016 IEEE 22nd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA). pp. 69–75 (Aug 2016). https://doi.org/10.1109/RTCSA.2016.17
35. Wang, X., Hovakimyan, N., Sha, L.: L1Simplex: Fault-tolerant control of cyber-physical systems. In: 2013 ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS). pp. 41–50 (April 2013)
36. Wang, Z., Bapst, V., Heess, N., Mnih, V., Munos, R., Kavukcuoglu, K., de Freitas, N.: Sample efficient actor-critic with experience replay. arXiv preprint arXiv:1611.01224 (2016)
37. Wen, M., Ehlers, R., Topcu, U.: Correct-by-synthesis reinforcement learning with temporal logic constraints. In: IROS. pp. 4983–4990. IEEE Computer Society Press (2015)
38. Xiang, W., Musau, P., Wild, A.A., Manzanas Lopez, D., Hamilton, N., Yang, X., Rosenfeld, J., Johnson, T.T.: Verification for machine learning, autonomy, and neural networks survey. ArXiv e-prints (Oct 2018)