Safe Neurosymbolic Learning with Differentiable Symbolic Execution

Anonymous Author(s)
Affiliation
Address
email

Abstract

We study the problem of learning worst-case-safe parameters for programs that use neural networks as well as symbolic, human-written code. Such *neurosymbolic* programs arise in many safety-critical domains. However, because they need not be continuous, let alone differentiable, they cannot be learned using existing gradient-based approaches to safe learning. Our method, *Differentiable Symbolic Execution* (DSE), learns such programs by sampling code paths using symbolic execution, constructing gradients of a worst-case "safety loss" along these paths, and then backpropagating these gradients through program operations using a generalization of the REINFORCE estimator. We evaluate the method on real-world benchmarks. Our experiments show that DSE significantly outperforms the state-of-the-art DIFFAI method on these tasks.

1 Introduction

Safety on worst-case inputs has recently emerged as a key challenge in deep learning research. Formal verification of neural networks [2] is an established response to this challenge. In particular, an exciting body of recent work [12, 10, 6, 14] has sought to incorporate formal verification into the training of neural networks. DIFFAI, among the most prominent of such approaches, uses a neural network verifier to construct a differentiable, worst-case safety loss for the learner. This loss is used to regularize a standard data-driven loss, biasing the learner towards parameters that are both performant and safe.

A weakness of these methods is that they only consider functional properties (such as adversarial robustness) of isolated neural networks. By contrast, in real-world applications, neural networks are often embedded within human-written symbolic code, and correctness requirements apply to the entire neurosymbolic composition. For example, consider a car directed by a neural controller [13]. Safety properties for the car are functions of its trajectories, and these trajectories depend not just on the controller but also the symbolic equations that define the environment. While recent work [5] has studied the verification of such neurosymbolic programs, there is no prior work on integrating verification and learning for such systems.

In this paper, we present the first steps towards such an integration. The fundamental technical challenge here is that while a neural network is differentiable over its parameters, the code surrounding may be non-differentiable or even discontinuous. This puts our problem beyond the scope of existing methods for integrated learning and verification.

We overcome this difficulty using a new method, *Differentiable Symbolic Execution* (DSE), for estimating gradients of worst-case safety losses of nondifferentiable neurosymbolic programs. DSE is based on a generalization of the classic REINFORCE estimator, which backpropagates gradients through non-differentiable operations by approximating integrals with sampling. In our problem, the integral is the aggregate of the safety losses along symbolic control flow paths in the program. To apply REINFORCE-like ideas here, we need to represent and sample paths in a symbolic way. We do so through an adaptation of the classic method of *symbolic execution*.

Submitted to 35th Conference on Neural Information Processing Systems (aiplans 2021). Do not distribute.

We evaluate DSE through several case studies in the embedded control and navigation domains. Our baselines include an extended version of DIFFAI, the current state of the art, and an ablation that does not use an explicit safety loss. Our experiments show that DSE significantly outperforms the baselines in finding safe and high-performance model parameters.

2 Problem Formulation

Programs. We define programs with embedded neural networks as *symbolic transition systems* (STS) [11]. Formally, a program F_θ is a tuple $(Loc, X, l_0, Init, Safe, Trans_\theta)$. Here, Loc is a finite set of *(control) locations*, $X = \{x_1, \ldots, x_m\}$ is a set of real-valued variables, and $l_0 \in Loc$ is the *initial location*. $Init$, a boolean formula over X, is an initial condition for the program. $Safe$ is a map from locations to constraints over X; intuitively, $Safe(l)$ is a safety requirement asserted at location l. Finally, $Trans_\theta$ is a *transition relation* consisting of *transitions* (l, G, U_θ, l') such that: (i) l is the *source location* and l' is the *destination location*; (ii) the *guard* G is a constraint over X; and (iii) the *update* U_θ is a vector $\langle U_{1,\theta}, \ldots, U_{m,\theta}\rangle$, where each $U_{i,\theta}$ is a real-valued expression over X constructed using standard symbolic operators and neural networks with parameters θ. We assume that each $U_{i,\theta}$ is differentiable in θ. Also, we assume that the programs are *deterministic*. That is, if G and G' are guards for two distinct transitions from the same source state, then $G \wedge G'$ is unsatisfiable.

Safety Semantics. Let a *state* of F_θ be a pair $s = (l, v)$, where l is a location and $v \in \mathbb{R}^m$ is an *assignment* of values to the variables (i.e., $v(i)$ is the value of x_i). Such a state is said to be *at* location l. A state (l_0, v), where v satisfies $Init$, is an *initial state*. A state (l, v) is *safe* if v satisfies $Safe(l)$.

Let $v \in \mathbb{R}^m$ be an assignment to the variables. For a real-valued expression E over X, let $E(v)$ be the value of E when x_i is substituted by $v(i)$. For an update $U = \langle U_1, \ldots, U_n\rangle$, we define $U(v)$ as the assignment $\langle U_1(v), \ldots, U_n(v)\rangle$. A length-$n$ *trajectory* of F_θ is a sequence $\tau = \langle s_0, \ldots, s_n\rangle$, with $s_i = (l_i, v_i)$, such that: (i) s_0 is an initial state; and (ii) for each i, there is a transition (l_i, G, U, l_{i+1}) such that v_i satisfies G and $v_{i+1} = U(v_i)$. We denote this trajectory by $\tau(s_0)$.

Let us assume a real-valued loss $Unsafe(s)$ that quantifies the *unsafeness* of each state s. We require $Unsafe(s) = 0$ if s is safe and $Unsafe(s) > 0$ otherwise. We lift this measure to trajectories τ by letting $Unsafe(\tau) = \sum_{s \text{ appears in } \tau} Unsafe(s)$. The *safety loss* $C(\theta)$ for F_θ is now defined as:

$$C(\theta) = \max_{s \text{ is an initial state}} Unsafe(\tau(s)). \tag{1}$$

Thus, $C(\theta) = 0$ if and only if all program trajectories are safe.

Problem Statement. Our learning problem formalizes a setting in which we have training data for neural networks inside a program F_θ. While training the networks with respect to this data, we must ensure that the overall program satisfies its safety requirements. To ensure that the parameters of the different neural networks in F_θ are not incorrectly entangled, we assume that only one of these networks, $\mathbf{NN}_\theta$, has trainable parameters.

We expect as input a training set of i.i.d. samples from an unknown distribution over the inputs and outputs of $\mathbf{NN}_\theta$, and a differentiable *data loss* $Q(\theta)$ that quantifies the network's fidelity to this training set. Our learning goal is to solve the following constrained optimization problem:

$$\min_\theta Q(\theta) \qquad \text{s.t. } C(\theta) \leq 0. \tag{2}$$

3 Approach

Learning Framework. Our learning approach is based on two ideas. First, we directly apply a recently-developed equivalence between constrained and regularized learning [1, 9] to reduce Equation (2) to a series of unconstrained optimization problems. (For more details, see Appendix A.1.) Second, we use the novel technique of *Differentiable Symbolic Execution* (DSE) to solve these unconstrained problems.

A key feature of our high-level algorithm is that it repeatedly solves the optimization problem

$$\min_\theta Q(\theta) + \lambda C(\theta) \tag{3}$$

for fixed values of λ. This problem is challenging because while $Q(\theta)$ is differentiable in θ, $C(\theta)$ depends on the entirety of F_θ and may not even be continuous. As we demonstrate in Section 4, this makes it difficult to apply state-of-the-art gradient-based approaches to worst-case safe learning. DSE, our main contribution, addresses this challenge by estimating gradients $\nabla_\theta C^\#(\theta)$ of a differentiable approximation $C^\#(\theta)$ of $C(\theta)$.

Background on Symbolic Execution. DSE is a refinement of *symbolic execution* [3], a classic technique for systematic formal analysis of programs. A symbolic executor systematically searches the set of *symbolic trajectories* of programs, which we now define. Consider a program $F_\theta = (Loc, X = \{x_1, \ldots, x_m\}, l_0, Init, Safe, Trans_\theta)$ as in Section 2. First, we fix a syntactically restricted class $\mathcal{V}_\theta$ of boolean constraints over the variables X and parameters θ. $\mathcal{V}_\theta$ is required to be closed under conjunction. Next, we define an *abstraction function* $\alpha : 2^{\mathbb{R}^m} \to \mathcal{V}_\theta$ that maps sets S of assignments to X to overapproximations drawn from $\mathcal{V}_\theta$.

We use α to construct an *overapproximate update* $U_\theta^\# : \mathcal{V}_\theta \to \mathcal{V}_\theta$ for each update U_θ in F_θ. For all $V \in \mathcal{V}_\theta$, we have $U_\theta^\#(V) = \alpha(\{U_\theta(v) : v \text{ satisfies } V\})$. A *symbolic state* of F_θ is now defined as a pair $\sigma_\theta = (l, V_\theta)$, where l is a location and $V \in \mathcal{V}_\theta$. Intuitively, σ_θ represents the set of states of F_θ that are at location l and satisfy the property V_θ.

Let us call a transition $t = (l, G, U, l')$ *enabled* at a symbolic state (l, V) if $(G \wedge V)$ is satisfiable. A *symbolic trajectory* of F_θ is a sequence $\tau_\theta^\# = \langle \sigma_0, \ldots, \sigma_n \rangle$ with the following properties: (i). $\sigma_0 = (l_0, V_0)$, with $V_0 \supseteq Init$; (ii). Let $\sigma_i = (l_i, V_i)$. Then there exists a transition $t = (l_i, G_i, U_i, l_{i+1})$ such that: (i) t is enabled at σ_i, and (ii) $V_{i+1} = U^\#(G_i \wedge V_i)$. In this case, we write $\sigma_{i+1} = t(\sigma_i)$. Intuitively, $\tau_\theta^\#$ represents an overapproximation of the set of concrete trajectories of F_θ that pass through the "control path" $\langle l_0, \ldots, l_n \rangle$.

The above symbolic trajectory is *safe* if for all i, $V_i \Rightarrow Safe(l_i)$. Intuitively, in this case, all concrete trajectories that the symbolic trajectory represents follow the program's safety requirements.

Probabilistic Symbolic Execution. A key difficulty with using symbolic execution to estimate the safety loss $C(\theta)$ is that our programs need not be differentiable, or even continuous. On the one hand, a discontinuous conditional statement can assign very different values to the variable in different branches. On the other hand, a slight change in θ can sometimes cause the guard G of the conditional to go from being $\mathbf{True}$ on *some* values of the variables used in conditional statement x, to being $\mathbf{False}$ on *all* values of x. If the symbolic trajectory $\tau^\#$ in which G is $\mathbf{True}$ serves as an edge case to judge whether a program is worst-case-safe, learning safe parameters would be difficult, as there would be no gradient guiding the optimizer back to the case in which G is $\mathbf{True}$.

DSE overcomes these difficulties using a probabilistic approach to symbolic execution. Specifically, at a symbolic state $\sigma_i = (l_i, V_i)$, the symbolic executor in DSE *samples* its next action following a probability distribution $p_\theta(t \mid \sigma_i)$, where t ranges over program transitions or a special action $Stop$.

For a boolean expression V_θ over the program variables and parameters, let $\mathbf{Vol}(V_\theta)$ denote the *volume* of the assignments to X that satisfy V_θ. Then we define: (i). If there is a unique program transition t that is enabled at σ_i, then $p_\theta(t|\sigma_i) = 1$; (ii). If there is no transition t that is enabled at σ_i, then $p_\theta(Stop|\sigma_i) = 1$; and (iii). Otherwise, let $t_1, \ldots, t_k$ be the transitions that are enabled at σ_i, with $t_i = (l_i, G_i, U_i, l_{i+1})$. Then $p_\theta(t_j|\sigma_i) = \frac{\mathbf{Vol}(G_j \wedge V_i)}{\mathbf{Vol}(V_i)}$.

DSE uses the distribution $p_\theta(t|\sigma_i)$ to sample symbolic trajectories. Let $\tau_\theta^\# = \langle \sigma_0, \ldots, \sigma_n \rangle$, with $\sigma_{i+1} = t_i(\sigma_i)$. The probability of sampling $\tau^\#$ is $p_\theta(\tau_\theta^\#) = \prod_i p_\theta(t_i|\sigma_{i-1})$. We note that p_θ is differentiable in θ.

Approximate Safety Loss. A key decision in DSE is to approximate the overall worst-case loss $C(\theta)$ by the *expectation* of a differentiable safety loss computed per sampled symbolic trajectory. Recall the safety loss $Unsafe(s)$ for individual states. For $\sigma = (l, V)$, we define $Unsafe_\theta(\sigma)$ as a differentiable approximation of the worst-case loss $\max_{s \text{ satisfies } V} Unsafe(s)$. We lift this loss to symbolic trajectories $\tau_\theta^\# = \langle \sigma_0, \ldots, \sigma_n \rangle$ by defining $Unsafe_\theta(\tau^\#) = \sum_i Unsafe_\theta(\sigma_i)$. We observe that $Unsafe_\theta$ is differentiable in θ.

Our approximation to the safety loss is now given by:

$$C^\#(\theta) = \mathbf{E}_{\tau^\# \sim p_\theta(\tau^\#)} Unsafe_\theta(\tau^\#). \tag{4}$$

Gradient Estimation. Our ultimate goal is to compute the gradient $\nabla_\theta C^\#(\theta)$ of the approximate safety loss. At first sight, the classic REINFORCE estimator, or its lower-variance refinements, seems suitable for this task, given that they can differentiate an integral over sampled "paths". However, a subtlety in our setting is that the losses for individual paths (symbolic trajectories) is a function of θ. This calls for a generalization of traditional REINFORCE-like estimators.

More precisely, we adapt the derivation of REINFORCE to our setting as follows:

$$
\begin{aligned}
\nabla_\theta(C^\#(\theta)) &= \nabla_\theta \mathbf{E}_{\tau^\# \sim p(\tau^\#)} Unsafe(\tau^\#) \\
&= \nabla_\theta \int_{\tau^\# \sim p(\tau^\#)} p(\tau^\#) Unsafe(\tau^\#) d\tau^\# \\
&= \int_{\tau^\# \sim p(\tau^\#)} p(\tau^\#) * \nabla_\theta Unsafe(\tau^\#) + \nabla_\theta p(\tau^\#) * Unsafe(\tau^\#) d\tau^\# \\
&= \mathbf{E}_{\tau^\# \sim p(\tau^\#)}[Unsafe(\tau^\#) * \nabla_\theta(\log p(\tau^\#))] + \mathbf{E}_{\tau^\# \sim p(\tau^\#)}[\nabla_\theta Unsafe(\tau^\#)]
\end{aligned}
$$

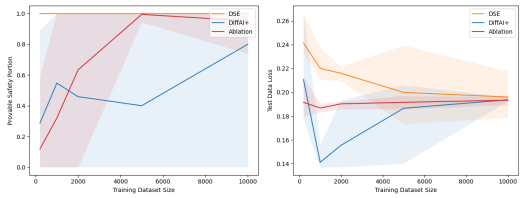
The above derivation can be further refined to incorporate the variance reduction techniques for REINFORCE that are commonly employed in generative modeling and reinforcement learning. The use of such techniques is orthogonal to our main contribution, and we ignore it in this paper.

4 Evaluation

Baselines. We use two types of baselines: (i). *Ablation*, which does not use safety constraint explicitly, but train each neural network modules in the programs for each task with its own dataset. (ii). DIFFAI +, an extended version of the original DIFFAI method [12]. DIFFAI+ extends DIFFAI by adding the meet and join operations following the abstract interpretation technique [7].

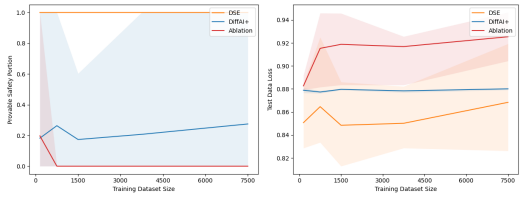
Benchmarks. We give 3 case studies as benchmarks (See A.5 for detailed programs.): (i). *Thermostat* models the controlling dynamics of a thermostat, where two neural networks modeling the controller are the target to learn; (ii). *Racetrack* is a variant of the navigation benchmark [4, 5]. Two vehicles are trained by path planners separately and the racetrack system is expected to learn two safe controllers so that vehicles do not crash into walls or with each other; and (iii). *Aircraft-Collision*(AC) aims to learn safe parameters for an airplane that performs maneuvers to avoid a collision with a second plane.

Results. From Figure 1a, 1c, 1e, we exhibit that DSE can learn programs with 0.99 provably safe portion even with 200 data points and the results keep when increasing the number of data used. Meanwhile, both DIFFAI + and ablation can not provide safe programs for AC and racetrack. For thermostat, ablation and DIFFAI + can achieve 0.8 provably safe portion only when using 5000 and 10000 data points to train.



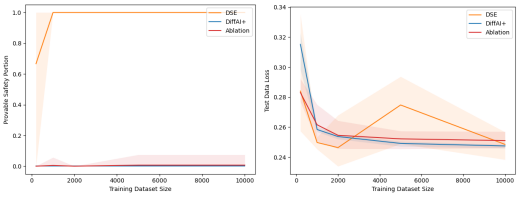
(a) Thermostat[Safety]

(b) Thermostat[Data Loss]



(c) AC[Safety]

(d) AC[Data Loss]



(e) Racetrack[Safety]

(f) Racetrack[Data Loss]

Figure 1: The provably safe portion and the test data loss of Ablation, DIFFAI + and DSE when varying the size of the data to train.

When increasing the number of data used, ablation and DIFFAI + can reach 0.95 provable safe portion with 10000 data points for the thermostat case while the results are still not stable as the minimum provable safe portion can reach 0.74 and 0.0 for ablation and DIFFAI +. In the thermostat case, DIFFAI +'s performance mainly comes from the guidance from the data loss rather than the safety loss. For the other two cases, larger data size can not help the ablation and DIFFAI + to give much safer programs. For AC, the program learnt from ablation is very close to the safe area but the ablation is still not accurate enough to give a safe program in every step. In the racetrack, increasing the data size to train vehicles' controllers can only satisfy the safety of "not crashing into walls". The property, "distance between vehicles" which depends on the interaction between neural network modules, can not be learnt to be safe when increasing data size as this property is not represented by isolated neural networks. Meanwhile, our test data loss is sometimes larger yet comparable with the ablation and DIFFAI + from Figure 1b, 1f. For AC specifically, the safety constraint can help the learner overcome some local optimal yet unsafe areas to get a safer result with more accurate behaviors.

References

[1] Alekh Agarwal, Alina Beygelzimer, Miroslav Dudík, John Langford, and Hanna Wallach. A reductions approach to fair classification. In *International Conference on Machine Learning*, pages 60–69. PMLR, 2018.

[2] Aws Albarghouthi. Introduction to neural network verification. *arXiv preprint arXiv:2109.10317*, 2021.

[3] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51(3):1–39, 2018.

[4] Andrew G Barto, Steven J Bradtke, and Satinder P Singh. Learning to act using real-time dynamic programming. *Artificial intelligence*, 72(1-2):81–138, 1995.

[5] Maria Christakis, Hasan Ferit Eniser, Holger Hermanns, Jörg Hoffmann, Yugesh Kothari, Jianlin Li, Jorge A Navas, and Valentin Wüstholz. Automated safety verification of programs invoking neural networks. In *International Conference on Computer Aided Verification*, pages 201–224. Springer, 2021.

[6] Jeremy Cohen, Elan Rosenfeld, and Zico Kolter. Certified adversarial robustness via randomized smoothing. In *International Conference on Machine Learning*, pages 1310–1320. PMLR, 2019.

[7] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, 1977.

[8] Yoav Freund and Robert E Schapire. Game theory, on-line prediction and boosting. In *Proceedings of the ninth annual conference on Computational learning theory*, pages 325–332, 1996.

[9] Hoang Le, Cameron Voloshin, and Yisong Yue. Batch policy learning under constraints. In *International Conference on Machine Learning*, pages 3703–3712. PMLR, 2019.

[10] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. *arXiv preprint arXiv:1706.06083*, 2017.

[11] Zohar Manna and Amir Pnueli. *Temporal verification of reactive systems: safety*. Springer Science & Business Media, 2012.

[12] Matthew Mirman, Timon Gehr, and Martin Vechev. Differentiable abstract interpretation for provably robust neural networks. In *International Conference on Machine Learning*, pages 3578–3586. PMLR, 2018.

[13] S Joe Qin and Thomas A Badgwell. An overview of nonlinear model predictive control applications. *Nonlinear model predictive control*, pages 369–392, 2000.

[14] Gagandeep Singh, Timon Gehr, Matthew Mirman, Markus Püschel, and Martin T Vechev. Fast and effective robustness certification. *NeurIPS*, 1(4):6, 2018.

Algorithm 1: Learning Safe, Optimal Parameter Mixtures [1, 9]

for $t = 1, ..., N$ **do**
  $\theta_t \leftarrow \mathbf{Best}_\theta(\lambda_t)$
  $\hat{\theta}_t \leftarrow \mathbf{Uniform}(\theta_1, \ldots, \theta_t),\ \hat{\lambda}_t \leftarrow \frac{1}{t}\sum \lambda_t$
  $L_{\max} \leftarrow L(\hat{\theta}, \mathbf{Best}_\lambda(\hat{\theta}))$
  $L_{\min} \leftarrow L(\mathbf{Best}_\theta(\hat{\lambda}_t), \hat{\lambda}_t)$
  if $L_{\max} - L_{\min} < \nu$ **then return** $(\hat{\theta}_t, \hat{\lambda}_t)$;
  $\lambda_{t+1} \leftarrow \boldsymbol{\lambda}\text{-}\mathbf{Update}(\theta_1, \ldots, \theta_t)$

A Appendix

A.1 Learning Framework

We use an equivalence between constrained and regularized learning that [1, 9], among others, have recently developed in other learning settings, we reduce our problem to a series of unconstrained optimization tasks.

We convexify the program set $\{[\![F_\theta]\!] : \theta \in \mathbb{R}^k\}$ by considering stochastic mixtures [9] and represent the convexified set as a probabilistic function $[\![F_{\hat{\theta}}]\!](v)$. Following DSE, we rewrite Equation 2 in terms of these mixtures:

$$\hat{\theta}^* \quad = \quad \arg\min_{\hat{\theta}} Q(\hat{\theta}) \qquad \text{s.t. } C^{\#}(\hat{\theta}) \leq 0 \tag{5}$$

We convert Equation 5 to a Lagrangian function

$$L(\hat{\theta}, \lambda) = Q(\hat{\theta}) + \lambda C^{\#}(\hat{\theta}) \tag{6}$$

$$\tag{7}$$

, where $\lambda \in \mathbb{R}^+$ is a *Lagrange multiplier*. Following the equilibrium computation technique by [8], Equation 5 can be rewritten as

$$\max_{\lambda \in \mathbb{R}^+} \min_{\hat{\theta}} L(\hat{\theta}, \lambda). \tag{8}$$

Solutions to this problem can be interpreted as equilibria of a game between a λ-player and a $\hat{\theta}$-player in which the $\hat{\theta}$-player minimizes $L(\hat{\theta}, \lambda)$ given the current λ, and the λ-player maximizes $L(\hat{\theta}, \lambda)$ given the current $\hat{\theta}$. Our overall algorithm is shown in Algorithm 1. In this pseudocode, ν is a predefined positive real. $\mathbf{Uniform}(\theta_1, \ldots, \theta_t)$ is the mixture that selects a θ_i out of $\{\theta_1, \ldots, \theta_t\}$ uniformly at random.

$\mathbf{Best}_\theta(\lambda)$ refers to the $\hat{\theta}$ player's *best response* for a given value of λ (it can be shown that this best response is a single parameter value, rather than a mixture of parameters). Computing this best response amounts to solving the unconstrained optimization problem $\min_\theta L(\theta, \lambda)$, i.e.,

$$\mathbf{Best}_\theta(\lambda) = \min_{\theta} Q(\theta) + \lambda C^{\#}(\theta). \tag{9}$$

$\mathbf{Best}_\lambda(\hat{\theta})$ is the λ-player's best response to a particular parameter mixture $\hat{\theta}$. We define this function as

$$\mathbf{Best}_\lambda(\hat{\theta}) = \begin{cases} 0 & \text{if } C^{\#}(\hat{\theta}) \leq 0 \\ S & \text{otherwise} \end{cases} \tag{10}$$

where S is the upper bound on λ. Intuitively, when $C^{\#}(\hat{\theta})$ is non-positive, the current parameter mixture is safe. As L is a linear function of λ, the minimum value of L is achieved in this case when λ is zero. For other cases, the minimum L is reached by setting λ to the maximum value S.

Finally, $\boldsymbol{\lambda}\text{-}\mathbf{Update}(\theta_t)$ computes, in constant time, a new value of λ based on the most recent best response by $\hat{\theta}$ player. We do not describe this function in detail; please see [1] for more details.

Following prior work, we can show that Algorithm 1 converges to a value $(\hat{\theta}, \hat{\lambda})$ that is within additive distance ν from a saddle point for Equation 8. To solve our original problem (Equation 2), we take

```
1  Example(x): // x ∈ [−5, 5]
2      y := NN_θ(x)
3      if y ≤ 1.0:
4          z := x + 10.0
5      else:
6          z := x − 5.0
7      assert (z <= 1)
```

$$
\begin{aligned}
Loc &= \{\ell_2, \ell_3, \ell_7\}, X = \{x, y, z\}, l_0 = \{\ell_2\} \\
Init &= (-5 \leq x \leq 5) \\
Safe(l_7) &= (z \leq 1), Safe(l_2) = \mathbf{True}, Safe(l_3) = \mathbf{True} \\
Trans_\theta &= \{(\ell_2, \mathbf{True}, \langle y := \mathbf{NN}_\theta(x)\rangle, \ell_3), \\
&\quad (\ell_3, (y \leq 1.0), \langle z := x + 10.0\rangle, \ell_7), \\
&\quad (\ell_3, (y > 1.0), \langle z := x - 5.0\rangle, \ell_7)
\end{aligned}
$$

Figure 2: (Left) An example program. $\mathbf{NN}_\theta$ is a neural network with parameters θ. DIFFAI fails to learn safe parameters for this program. (Right) The program as an STS. The location ℓ_i in the STS corresponds to line i in the program. We abbreviate the updates by only showing the variable that changes value.

the returned $\hat{\theta}$ and then return the real-valued parameter θ_i to which this mixture assigns the highest probability. It is easy to see that this value is an approximate solution to Equation 2.

A.2 An Example Program Following Our Approach

Programs in higher-level languages can be translated to the STS notation in a standard way. For example, Figure 2 (left) shows a simple high-level program. The STS for this program appears in Figure 2 (right). Remarkably, while the program is simple, the state-of-the-art DIFFAI approach to verified learning fails to learn safe parameters for it.

Assume that $\mathbf{NN}_\theta \in [-2, 1]$ when $x > 0$, and $\mathbf{NN}_\theta \in (1, 2]$ when $x \leq 0$. This program has two symbolic trajectories from $x \in [-5, 5]$. We represent the state set over x, y, z:

- $\tau_1^\# = \langle(\ell_2, x \in [-5, 5]), (\ell_3, x \in [-5, 5] \wedge y \in [-2, 2]), (\ell_7, x \in [-5, 5] \wedge y \in [-2, 1] \wedge z \in [5, 15])\rangle$
- $\tau_2^\# = \langle(\ell_2, x \in [-5, 5]), (\ell_3, x \in [-5, 5] \wedge y \in [-2, 2]), (\ell_7, x \in [-5, 5] \wedge y \in (1, 2] \wedge z \in [-10, 0])\rangle$.

We note that only $\tau_2^\#$ is safe.

We have $p(\tau_1^\#) = p(t_2|\sigma_2)$, as the other transitions in $\tau_1^\#$ have probability 1. We compute $p(t_2|\sigma_2) = \frac{\mathbf{Vol}([-2,1])}{\mathbf{Vol}([-2,2])} = 0.75$. Thus, $p(\tau_1^\#) = 0.75$ and similarly, $p(\tau_2^\#) = 0.25$.

A.3 Abstract Update for Neural Network Modules

We consider the box domain in the implementation. For a program with p variables, each component in the domain represents a p-dimensional box. Each component of the domain is a pair $b = \langle b_c, b_e\rangle$, where $b_c \in \mathbb{R}^p$ is the center of the box and $b_e \in \mathbb{R}^p_{\geq 0}$ represents the non-negative deviations. The interval concretization of the i-th dimension variable of b is given by

$$[(b_c)_i - (b_e)_i, (b_c)_i + (b_e)_i].$$

Now we give the abstract update for the box domain following [12].

Add. For a concrete function f that replaces the i-th element in the input vector $x \in \mathbb{R}^p$ by the sum of the j-th and k-th element:

$$f(x) = (x_1, \ldots, x_{i-1}, x_j + x_k, x_{i+1}, \ldots x_p)^T.$$

The abstraction function of f is given by:

$$f^\#(b) = \langle M \cdot b_c, M \cdot b_e\rangle,$$

where $M \in \mathbb{R}^{p \times p}$ can replace the i-th element of x by the sum of the j-th and k-th element by $M \cdot b_c$.

Multiplication. For a concrete function f that multiplies the i-th element in the input vector $x \in \mathbb{R}^p$ by a constant w:

$$f(x) = (x_1, \ldots, x_{i-1}, w \cdot x_i, x_{i+1}, \ldots, x_p)^T.$$

The abstraction function of f is given by:

$$f^\#(b) = \langle M_w \cdot b_c, M_{|w|} \cdot b_e\rangle,$$

where $M_w \cdot b_c$ multiplies the i-th element of b_c by w and $M_{|w|} \cdot b_e$ multiplies the i-th element of b_e with $|w|$.

Matrix Multiplication. For a concrete function f that multiplies the input $x \in \mathbb{R}^p$ by a fixed matrix $M \in \mathbb{R}^{p' \times p}$:

$$f(x) = M \cdot x.$$

The abstraction function of f is given by:

$$f^{\#}(b) = \langle M \cdot b_c, |M| \cdot b_e \rangle,$$

where M is an element-wise absolute value operation. Convolutions follow the same approach, as they are also linear operations.

ReLU. For a concrete element-wise ReLU operation over $x \in \mathbb{R}^p$:

$$\text{ReLU}(x) = (\max(x_1, 0), \ldots, \max(x_p, 0))^T,$$

the abstraction function of ReLU is given by:

$$\text{ReLU}^{\#}(b) = \langle \frac{\text{ReLU}(b_c + b_e) + \text{ReLU}(b_c - b_e)}{2}, \frac{\text{ReLU}(b_c + b_e) - \text{ReLU}(b_c - b_e)}{2} \rangle.$$

where $b_c + b_e$ and $b_c - b_e$ denotes the element-wise sum and element-wise subtraction between b_c and b_e.

Sigmoid. As Sigmoid and ReLU are both monotonic functions, the abstraction functions follow the same approach. For a concrete element-wise Sigmoid operation over $x \in \mathbb{R}^p$:

$$\text{Sigmoid}(x) = (\frac{1}{1 + \exp(-x_1)}, \ldots, \frac{1}{1 + \exp(-x_1)})^T,$$

the abstraction function of Sigmoid is given by:

$$\text{Sigmoid}^{\#}(b) = \langle \frac{\text{Sigmoid}(b_c + b_e) + \text{Sigmoid}(b_c - b_e)}{2}, \frac{\text{Sigmoid}(b_c + b_e) - \text{Sigmoid}(b_c - b_e)}{2} \rangle.$$

where $b_c + b_e$ and $b_c - b_e$ denotes the element-wise sum and element-wise subtraction between b_c and b_e. All the above abstract updates can be easily differentiable and parallelized on the GPU.

A.4 Instantiation of the $Unsafe$ Function

In general, the $Unsafe(s)$ function over individual states can be any differentiable distance function between a point and a set which satisfies the property that $Unsafe(s) = 0$ if s is in the safe set, $\mathcal{A}$, and $Unsafe(s) > 0$ if s is not in $\mathcal{A}$. We give the following instantiation as the unsafeness score over individual states:

$$Unsafe(s) = \begin{cases} \min_{x \in \mathcal{A}} \textsc{Dist}(s, x) & \text{if } s \notin \mathcal{A} \\ 0 & \text{if } s \in \mathcal{A} \end{cases}$$

where DIST denotes the euclidean distance between two points.

Similarly, the $Unsafe(\sigma)$ function over symbolic states can be any differentiable distance function between two sets which satisfies the property that $Unsafe(\sigma) = 0$ if V is in $\mathcal{A}$, and $Unsafe(\sigma) > 0$ if V is not in $\mathcal{A}$. We give the following instantiation as a differentiable approximation of the worst-case loss $\max_{s \text{ satisfies } V} Unsafe(s)$ in our implementation:

$$Unsafe(\sigma) = \begin{cases} \min_{s \text{ satisfies } V} Unsafe(s) + 1 & \text{if } V \wedge \mathcal{A} = \emptyset \\ 1 - \frac{\mathbf{Vol}(V \wedge \mathcal{A})}{\mathbf{Vol}(V)} & \text{if } V \wedge \mathcal{A} \neq \emptyset \end{cases}$$

A.5 Benchmarks for Case Studies

The detailed programs describing Aircraft-Collision, Thermostat and Racetrack are in Figure 3, 4 and 5. The π_θ of Aircraft Collision and $\pi_\theta^{\texttt{cool}}$, $\pi_\theta^{\texttt{heat}}$ in Thermostat are a 3 layer feed forward net with 64 nodes in each and a ReLU after each layer except the last one. A sigmoid layer serves as the last layer. Both the $\pi_\theta^{\texttt{agent1}}$ and $\pi_\theta^{\texttt{agent2}}$ in Racetrack are a 3 layer feed forward net with 64 nodes in each and a ReLU after each layer.

```
aircraftcollision(x):
   x1, y1 := x, -15.0
   x2, y2 := 0.0, 0.0
   steps := 15
   i = 0
   while i < steps:
      p0, p1, p2, p3, step := π_θ(x1, y1, x2, y2, step, stage)
      stage := argmax(p0, p1, p2, p3)
      if stage == CRUISE:
         x1, y1 := MOVE_CRUISE(x1, y1)
      else if stage == LEFT:
         x1, y1 := MOVE_LEFT(x1, y1)
      else if stage == STRAIGHT:
         x1, y1 := MOVE_STRAIGHT(x1, y1)
      else:
         x1, y1 := MOVE_RIGHT(x1, y1)

      x2, y2 := MOVE_2(x2, y2)
      i := i + 1
      assert (!IS_CRASH(x1, y1, x2, y2))
   return
```

Figure 3: Aircraft Collision

```
thermostat(x):
   steps = 20
   isOn = 0.0
   i = 0
   while i < steps:
      if isOn ≤ 0.5:
         isOn := π_θ^cool(x)
         x := COOLING(x)
      else:
         isOn, heat := π_θ^heat(x)
         x := WARMING(x, heat)
      i := i + 1
      assert(!EXTREME_TEMPERATURE(x))

   return
```

Figure 4: Thermostat

A.6 Experimental Details and Additional Results

A.6.1 Data Generation

For each program to be synthesized, we have a benchmark program consisting of predefined functions (in the form of classic programs without neural networks).The program-form module is replaced by neural networks with unknown parameters in the training and evaluation steps.

For each benchmark, we uniformly sample the points from a continuous input set and execute the predefined benchmark program to get the input-output example dataset for each neural network modules. Specifically, as we generate the data when treating the neurosymbolic program as a whole and each program is safe, a neurosymbolic program with neural network modules 100% fitting the datasets satisfies the safety constraint when the safety property can be measured. That said, any concrete trajectories produced by the benchmark program, which starts from the points in the input set are *safe*. Interestingly, there are also benchmark programs where the safety can not be measured. Take racetrack as an example, we consider a racetrack system safe if two vehicles do not crash into the maps and each other. The two vehicles are trained separately as we only require each of the vehicle to follow the path planner in one map and there is no specific distance requirement of them when mimicing the path planner's behaviors. Therefore, there is no safety property representation

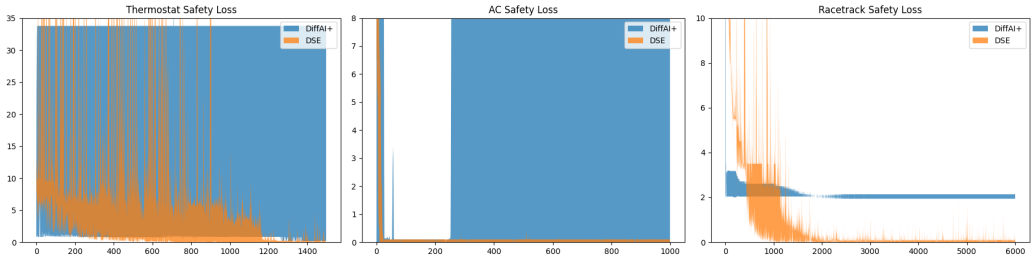
```
racetrack(x):
   x1, y1, x2, y2 := x, 0.0, x, 0.0
   steps := 20
   while i < steps:
      p10, p11, p12 := π_θ^agent1(x1, y1)
      p20, p21, p22 := π_θ^agent2(x2, y2)
      action1 := argmax(p10, p11, p12)
      action2 := argmax(p20, p21, p22)
      x1, y1 := MOVE(x1, y1, action1)
      x2, y2 := MOVE(x2, y2, action2)
      i := i + 1
      assert(!CRASH_WALL(x1, y1) && !CRASH_WALL(x2, y2)
         && !CRASH(x1, y1, x2, y2))

   return
```

Figure 5: Racetrack

over the distance between vehicles during data generation. The benchmark program satisfies other properties which exist, e.g. each vehicle not crashing into the wall.

A.6.2 Training Performance

Figure 6 illustrates the training performance of safety loss when we use 200 data points to train for DIFFAI + and DSE. We can see that DSE can converge while DIFFAI + is quite unstable for these three cases and can not converge to a small safety loss.



(a) Thermostat (b) Aircraft Collision (c) Racetrack

Figure 6: Training performance on different benchmarks. The y-axis represents the safety loss ($C^{\#}(\theta)$) and the x-axis gives the number of training epochs.

A.6.3 Provable Safety Analysis

Figure 7 displays the trajectories from the learnt programs of Ablation, DIFFAI + and DSE. The larger portion of the symbolic trajectories is provably safe from DSE than the ones from baselines which is indicated by the less overlapping between the green trajectories and the gray area. Meanwhile, we give the concrete trajectories to remove the influence from the potential over-approximation of the symbolic trajectories. More concrete trajectories of thermostat, aircraft-collision, and the distance property of racetrack from DSE fall into the safe area compared with DIFFAI + and Ablation.

A.6.4 Target Problems

For cases studies, the three cases are all neurosymbolic programs with neural modules to make decisions. Meanwhile, the interaction between neural networks is important for these benchmarks. In thermostat and aircraft-collision, the neural modules' outputs decide the actions that the neural module to take in the next steps. In racetrack, the distance between two neural modules regulates each step of the vehicles' controllers. In summary, DSE performs much better in the cases where neural modules are used in the conditional statement and the interaction between neural modules is important for the safety property.

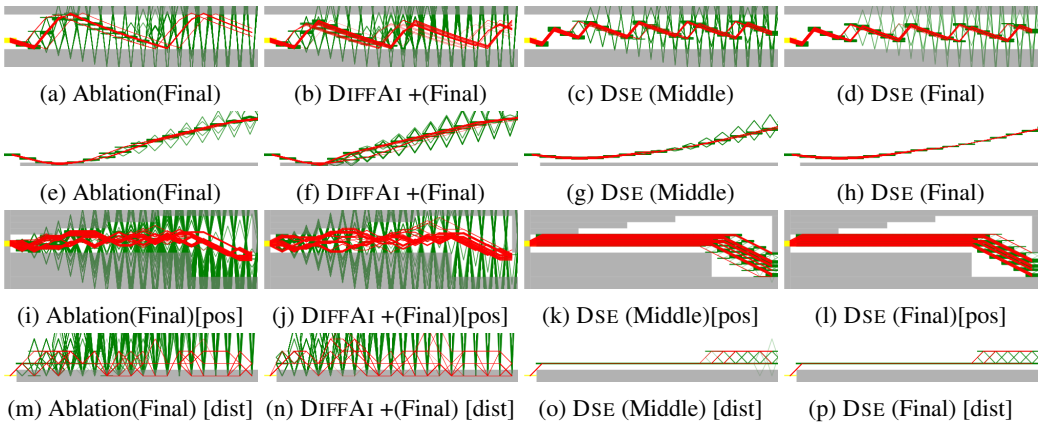


Figure 7: Trajectories during training. Each row exhibits the concrete trajectories and symbolic trajectories of one case from different methods. From top to bottom, the cases are Thermostat (Figure. 7a, 7b, 7c, 7d), Aircraft-Collision (Figure. 7e, 7f, 7g, 7h), Racetrack with position property (Figure. 7i, 7j, 7k, 7p) and distance property (Figure. 7m, 7n, 7o, 7p). Each figure shows the trajectories (concrete: red, symbolic: green) of programs learnt by the method from the different training stages, which is denoted by "Middle" and "Final". We separate the input state set into 100 components to plot the symbolic trajectories clearly. During evaluation, we separate the input set into 10000 components to get more accurate symbolic trajectories measurement.