Neural State Classification for Hybrid Systems*

Dung Phan[1], Nicola Paoletti[1], Timothy Zhang[1], Radu Grosu[2],
Scott A. Smolka[1], and Scott D. Stoller[1]

[1] Department of Computer Science, Stony Brook University, Stony Brook, NY, USA
[2] Department of Computer Engineering, Technische Universitat Wien, Vienna, Austria

Abstract. We introduce the *State Classification Problem* (SCP) for hybrid systems, and present *Neural State Classification* (NSC) as an efficient solution technique. SCP generalizes the model checking problem as it entails classifying each state s of a hybrid automaton as either positive or negative, depending on whether or not s satisfies a given time-bounded reachability specification. This is an interesting problem in its own right, which NSC solves using machine-learning techniques, Deep Neural Networks in particular. State classifiers produced by NSC tend to be very efficient (run in constant time and space), but may be subject to classification errors. To quantify and mitigate such errors, our approach comprises: i) techniques for certifying, with statistical guarantees, that an NSC classifier meets given accuracy levels; ii) tuning techniques, including a novel technique based on *adversarial sampling*, that can virtually eliminate false negatives (positive states classified as negative), thereby making the classifier more conservative. We have applied NSC to six nonlinear hybrid system benchmarks, achieving an accuracy of 99.25% to 99.98%, and a false-negative rate of 0.0033 to 0, which we further reduced to 0.0015 to 0 after tuning the classifier. We believe that this level of accuracy is acceptable in many practical applications, and that these results demonstrate the promise of the NSC approach.

1 Introduction

Model checking of hybrid systems is usually expressed in terms of the following reachability problem for hybrid automata (HA): given an HA $\mathcal{M}$, a set of initial states I, and a set of unsafe states U, determine whether there exists a trajectory of $\mathcal{M}$ starting in an initial state and ending in an unsafe state. The time-bounded version of this problem considers trajectories that are within a given time bound T. It has been shown that reachability problems and time-bounded reachability problems for HA are undecidable [16], except for some fairly restrictive classes of HA [7,16]. HA model checkers cope with this undecidability by providing approximate answers to reachability [13].

This paper introduces the *State Classification Problem* (SCP), a generalization of the model checking problem for hybrid systems. Let $\mathbb{B} = \{0, 1\}$ be the set of Boolean values. Given an HA $\mathcal{M}$ with state space S, time bound T, and set of unsafe states $U \subset S$, the SCP problem is to find a function $F^* : S \to \mathbb{B}$ such that for all $s \in S$, $F^*(s) = 1$ if it is possible for $\mathcal{M}$, starting in s, to reach a state in U within time T;

* This material is based on work supported in part by AFOSR Grant FA9550-14-1-0261, NSF Grants CPS-1446832, IIS-1447549, CNS-1445770, CNS-1421893, and CCF-1414078, FWF-NFN RiSE Award, and ONR Grant N00014-15-1-2208.

$F^*(s) = 0$ otherwise. A state $s \in S$ is called *positive* if $F^*(s) = 1$. Otherwise, s is *negative*. We call such a function a *state classifier*.

SCP generalizes the model checking problem. Model checking, in the context of SCP, is simply the problem of determining whether there exists a positive state in the set of initial states. Its intent is not to classify all states in S.

Classifying the states of a complex system is an interesting problem in its own right. State classification is also useful in at least two other contexts. First, due to random disturbances, a hybrid system may restart in a random state outside the initial region, and we may wish to check the system's safety from that state. Secondly, a classifier can be used for *online model checking* [27], where in the process of monitoring a system's behavior, one would like to determine, in real-time, the fate of the system going forward from the current (non-initial) state.

This paper shows how deep neural networks (DNNs) can be used for state classification, an approach we refer to as *Neural State Classification* (NSC). An NSC classifier is subject to *false positives* (FPs) – a state s is deemed positive when it is actually negative, and, more importantly, *false negatives* (FNs) – s is deemed negative when it is actually positive.

A well-trained NSC classifier offers high accuracy, runs in constant time (approximately 1 millisecond, in our experiments), and takes constant space (e.g., a DNN with l hidden layers and n neurons only requires functions of dimension $l \cdot n$ for its encoding). This makes NSC classifiers very appealing for applications such as online model checking, a type of analysis subject to strict time and space constraints. NSC classifiers can also be used in runtime verification applications where a low probability of FNs is acceptable, e.g., performance-related system monitoring.

Our approach can also classify states of *parametric* HA by simply encoding each parameter as an additional input to the classifier. This makes NSC more powerful than state-of-the-art hybrid system reachability tools that have little or no support for parametric analysis [12,13]. In particular, we can train a classifier that classifies states of any instance of the parameterized HA, even instances with parameter values not seen during training.

NSC-based classification can be lifted from states to (convex) sets of states by applying output-range estimation [30]. Such techniques can be used to compute safe bounds for the given state region.

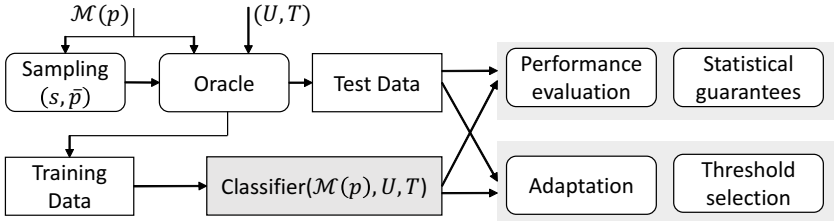


Fig. 1: Overview of the NSC approach.

The NSC method is summarized in Fig. 1. We train the state classifier using supervised learning, where the training examples are derived by sampling the state and

parameter spaces according to some distribution. Reachability values for the examples are computed by invoking an oracle, i.e., an hybrid system model checker [13] or a simulator when the system is deterministic.

We consider three sampling strategies: *uniform*, where every state is equi-probable, *balanced*, which seeks to improve accuracy by drawing a balanced number of positive and negative states, and *dynamics-aware*, which assigns to each state the estimated probability that the state is visited in any time-bounded evolution of the system. The choice of sampling strategy depends on the intended application of NSC. For example, in the case of online model checking, dynamics-aware sampling may be the most appropriate. For balanced sampling, we introduce a method to generate arbitrarily large sets of positive samples based on constructing and simulating reverse HAs.

NSC is not limited to DNN-based classifiers. We demonstrate that other machine-learning models for classification, such as support vector machines (SVMs) and binary decision trees (BDTs), also provide powerful solution techniques.

Given the impossibility of training machine-learning models with guaranteed accuracy w.r.t. the true input distribution, we evaluate a trained state classifier by estimating its accuracy, false-positive rate, and false-negative rate (together with their confidence intervals) on a test dataset of fresh samples. This allows us to quantify how well the classifier extrapolates to unseen states, i.e., the probability that it correctly predicts reachability for any state.

Inspired by statistical model checking [24], we also provide statistical guarantees through sequential hypothesis testing to certify (up to some confidence level) that the classifier meets prescribed accuracy levels on unseen data. Note that the systems we consider are *nonprobabilistic*. The statistical guarantees we provide are for the probability that the classifier makes the correct prediction. In contrast, the aim of *probabilistic model checking* [22] and *statistical model checking* [24] is to compute the probability that a probabilistic system satisfies a given correctness property. Relatedly, the focus of neural network (NN) verification [18,11,21] is on proving properties of an NN's output rather than the NN's accuracy.

We also consider two tuning methods that can reduce and virtually eliminate false negatives: a new method called *falsification-guided adaptation* that iteratively re-trains the classifier with false negatives found through adversarial sampling; and *threshold selection*, which adjusts the NN's classification threshold to favor FPs over FNs.

Our experimental results demonstrate the feasibility and promise of our approach, evaluated on six nonlinear hybrid system benchmarks. We consider shallow (1 hidden layer) and deep (3 hidden layers) NNs with sigmoid and ReLU activation functions. Our techniques achieve a prediction accuracy of 99.25% to 99.98% and a false-negative rate of 0.0033 to 0, taking into account the best classifier for each of the six models, with DNNs yielding superior accuracy than shallow NNs, SVMs, and BDTs. We believe that such a range for the FN rate is acceptable in many practical applications, and we show how this can be further improved through tuning of the classifiers.

In summary, our main contributions are the following:

- We introduce the State Classification Problem for hybrid systems.

– We develop the Neural State Classification method for solving the SCP, including techniques for sampling, establishing statistical guarantees on a classifier's accuracy, and reducing its FN rate.
– We introduce a new technique for constructing the *reverse HA* of a given HA, for a general class of HAs, and use reverse HAs to generate balanced training datasets.
– We introduce a *falsification-guided adaptation algorithm* for eliminating FNs, thereby producing conservative state classifiers.
– We provide an extensive evaluation on six nonlinear hybrid system models.

2 Problem Formulation

We introduce the problem of learning a state classifier for a hybrid automaton and a bounded reachability property. First, we define these terms.

Definition 1 (Hybrid automaton). *A* hybrid automaton *(HA) is a tuple* $\mathcal{M} = (Loc, Var, Init, Flow, Trans, Inv)$*, where* Loc *is a finite set of discrete* locations *(or modes);* $Var = \{x_1, \ldots, x_n\}$ *is a set of continuous* variables*, evaluated over a* continuous domain $X \subseteq \mathbb{R}^n$*;* $Init \subseteq S(\mathcal{M})$ *is the set of* initial states*, where* $S(\mathcal{M}) = Loc \times X$ *is the* state space *of* $\mathcal{M}$*;* $Flow : Loc \to (X \to X)$ *is the* flow *function, defining the continuous dynamics at each location;* $Trans$ *is the* transition relation*, consisting of tuples of the form* (l, g, v, l')*, where* $l, l' \in Loc$ *are* source and target locations*, respectively,* $g \subseteq X$ *is the* guard*, and* $v : X \to X$ *is the* reset*;* $Inv : Loc \to 2^X$ *is the* invariant *at each location.*

We also consider *parameterized* HA in which the flow, guard, reset and invariant may have parameters whose values are constant throughout an execution. We treat parameters as continuous variables with flow equal to zero and identity reset map.

The behavior of an HA $\mathcal{M}$ can be described in terms of its trajectories. A trajectory may start from any state; it does not need to start from an initial state. For time bound $T \in \mathbb{R}^{\geq 0}$, let $\mathbb{T} = [0, T] \subseteq \mathbb{R}^{\geq 0}$ be the time domain.

Definition 2 (Trajectory [3]). *For HA* $\mathcal{M} = (Loc, Var, Init, Flow, Trans, Inv)$*, time domain* $\mathbb{T} = [0, T]$*, let* $\rho : \mathbb{T} \to S(\mathcal{M})$ *be a function mapping time instants into states of* $\mathcal{M}$*. For* $t \in \mathbb{T}$*, let* $\rho(t) = (l(t), \mathbf{x}(t))$ *be the state at time* t*, with* $l(t)$ *being the location and* $\mathbf{x}(t)$ *the vector of continuous variables. Let* $(\xi_i)_{i=0,\ldots,k} \in \mathbb{T}^{k+1}$ *be the ordered sequence of time points where mode jumps happen, i.e., such that* $\xi_0 = 0$*,* $\xi_k = T$*, and for all* $i = 0, \ldots, k-1$ *and for all* $t \in [\xi_i, \xi_{i+1})$*,* $l(t) = l(\xi_i)$*. Then,* ρ *is a* trajectory *of* $\mathcal{M}$ *if it is consistent with the invariants:* $\forall t \in \mathbb{T}.\ \mathbf{x}(t) \in Inv(l(t))$*; flows:* $\forall t \in \mathbb{T}.\ \dot{\mathbf{x}}(t) = Flow(l(t))(\mathbf{x}(t))$*; and transition relation:* $\forall i < k.\ \exists (l(\xi_i), g, v, l(\xi_{i+1})) \in Trans.\ \mathbf{x}(\xi_{i+1}^-) \in g \wedge \mathbf{x}(\xi_{i+1}) = v(\mathbf{x}(\xi_{i+1}^-))$.

Definition 3 (Time-bounded reachability). *Given an HA* $\mathcal{M}$*, set of states* $U \subseteq S(\mathcal{M})$*, state* $s \in S(\mathcal{M})$*, and time bound* T*, decide whether there exists a trajectory* ρ *of* $\mathcal{M}$ *starting from* s *and* $t \in [0, T]$ *such that* $\rho(t) \in U$*, denoted* $\mathcal{M} \models \mathsf{Reach}(U, s, T)$.

Definition 4 (Positive and negative states). *Given an HA* $\mathcal{M}$*, set of states* $U \subseteq S(\mathcal{M})$*, called* unsafe states*, and time bound* T*, a state* $s \in S(\mathcal{M})$ *is called* positive *if* $\mathcal{M} \models \mathsf{Reach}(U, s, T)$*, i.e., an unsafe state is reachable from* s *within time* T*. Otherwise,* s *is called* negative*.*

We will use the term *positive (negative) region* for $\mathcal{M}$'s set of positive (negative) states.

Definition 5 (State classification problem). *Given an HA $\mathcal{M}$, set of states $U \subseteq S(\mathcal{M})$, and time bound T, find a function $F^* : S(\mathcal{M}) \to \mathbb{B}$ such that $F^*(s) = \mathcal{M} \models \mathsf{Reach}(U, s, T)$ for all $s \in S(\mathcal{M})$.*

It is easy to see that the model checking problem for hybrid systems can be expressed as an SCP in which the domain of F^* is the set of initial states $Init$, instead of the whole state space. SCP is therefore a generalization of the model checking problem.

Sample sets are used by NSC to learn state classifiers and to evaluate their performance. Unsafe states are trivially positive (for any T), so we exclude them from the sampling domain. Each sample consists of a state s and Boolean b which is the answer to the reachability problem starting from state s. We call $(s, 1)$ a *positive sample* and $(s, 0)$ a *negative sample*. Both kinds of samples are generally needed for adequately learning a classifier.

Definition 6 (Sample set). *For model $\mathcal{M}$, set of states $U \subseteq S(\mathcal{M})$, and time bound T, a* sample set *is any finite set $\{(s, b) \in (S(\mathcal{M}) \setminus U) \times \mathbb{B} \mid b = (\mathcal{M} \models \mathsf{Reach}(U, s, T))\}$*

The derivation of an NSC classifier reduces to a *supervised learning problem*, specifically, a binary classification problem. Given a sample set D called the *training set*, NSC approximates the exact state classifier F^* in Definition 5 by learning a total function $F : (S(\mathcal{M}) \setminus U) \to \mathbb{B}$ from D.

Learning F typically corresponds to finding values of F's parameters that minimize some measure of discrepancy between F and the training set D. We do not require that the learned function agree with the D on every state that appears in D, because this can lead to over-fitting to D and hence poor generalization to other states.

To evaluate the performance of F, we use three standard metrics: *accuracy* P_{A}, i.e, the probability that F produces the correct prediction; the probability of *false positives*, P_{FP}; and the probability of *false negatives*, P_{FN}. In safety-critical applications, achieving a low FN rate is typically more important than achieving a low FP rate. Precisely computing these probabilities is, in general, infeasible. We therefore compute an empirical accuracy measure, false-positive rate, and false-negative rate over a *test set* $\mathcal{D}'$ containing n fresh samples not appearing in the training set as follows:

$$\hat{P}_{\mathsf{A}} = \frac{1}{n} \sum_{(s,b)\in D'} \mathbf{1}_{F(s)=b} \ , \ \hat{P}_{\mathsf{FP}} = \frac{1}{n} \sum_{(s,b)\in D'} \mathbf{1}_{F(s)\wedge\neg b} \ , \ \hat{P}_{\mathsf{FN}} = \frac{1}{n} \sum_{(s,b)\in D'} \mathbf{1}_{\neg F(s)\wedge b} \quad (1)$$

where $\mathbf{1}$ is the indicator function. We obtain statistically sound bounds for these probabilities through the Clopper-Pearson method for deriving precise confidence intervals.

3 Neural State Classification

This section introduces the main components of the NSC approach.

3.1 Neural Networks for Classification

NSC uses *feedforward* neural networks, a type of neural network with one-way connections from input to output layers [23]. NSC uses both *shallow* NNs, with one hidden layer and one output layer, and *deep* NNs, with multiple hidden layers. Additional background on NNs is provided in an extended version of this paper [26].

An NN defines a real-valued function $F(\mathrm{x})$. When using an NN for classification, a *classification threshold* θ is specified, and an input vector x is classified as positive if $F(\mathrm{x}) \geq \theta$, and as negative otherwise.

The theoretical justification for using NNs to solve the SCP is the following. In [17], it is shown that shallow feedforward NNs are universal approximators; i.e., with appropriate parameters, they can approximate any Borel-measurable function arbitrarily well with a finite number of neurons (and just one hidden layer). Under mild assumptions, this also applies to the true state classifier F^* of the SCP (Definition 5). A proof of this claim is given in [26] . Arbitrarily high precision might not be achievable in practice, as it would require significantly large training sets and numbers of neurons, and a precise learning algorithm. Nevertheless, NNs are extremely powerful.

3.2 Oracles

Given a state (sample) s of an HA M, an NSC *oracle* is a procedure for labeling s; i.e., for deciding whether $\mathcal{M} \models \mathsf{Reach}(U, s, T)$. NSC utilizes the following oracles.

Reachability checker. For nonlinear HA, NSC uses dReal [13], an SMT solver that supports bounded model checking of such HA. dReal provides sound unsatisfiability proofs, but satisfiability is approximated up to a user-defined precision (δ-satisfiability). The oracle first attempts to verify that s is negative by checking $\mathcal{M} \models \mathsf{Reach}(U, s, T)$ for unsatisfiability. If this instance is instead δ-sat, the oracle attempts to prove the unsatisfiability of $\mathcal{M} \models \neg\mathsf{Reach}(U, s, T)$, which would imply that s is positive. The latter instance can also be δ-sat, meaning that this oracle cannot make a decision about s. This situation never occurred in our evaluation and can be made less likely by choosing a small δ. If it did occur, our plan would be to conservatively mark the state as positive. The oracle requires an upper bound on the number of discrete jumps to be considered. It supports HAs with Lipschitz continuous dynamics and hyperrectangular continuous domains (i.e., defined as the product of closed intervals), and allows trigonometric and other non-polynomial functions in the initial conditions, guards, invariants, and resets.

Simulator. For deterministic systems, we implemented a simulator based on MATLAB's `ode45` variable-step ODE solver. To check reachability, we employ the solver's event-detection method to catch relevant zero-crossing events (i.e., reaching U).

Backwards simulator. The backwards simulator is not an oracle per se, but, as described in Section 3.3, is central to one of our sampling methods. We first construct the reverse HA according to Definition 7, which is more general than the one for rectangular HAs given in [16]. We use dot-notation to indicate members of a tuple, and lift resets v to sets of states; i.e., $v(X') = \{v(\mathrm{x}) \mid \mathrm{x} \in X'\}$.

Definition 7 (Reverse HA). *Given an HA $\mathcal{M}$, its* reverse HA $\overleftarrow{\mathcal{M}}$ *is an HA such that the modes, continuous variables, and invariants are the same as for $\mathcal{M}$, the flows are*

reversed, i.e., $\forall(l,\mathrm{x}) \in S(\mathcal{M}),\ \overleftarrow{\mathcal{M}}.Flow(l)(\mathrm{x}) = -\mathcal{M}.Flow(l)(\mathrm{x})$*, and for each transition* $(l,g,v,l') \in \mathcal{M}.Trans$*, the corresponding transition* $(l',\overleftarrow{g},\overleftarrow{v},l) \in \overleftarrow{\mathcal{M}}.Trans$ *must be such that* $\overleftarrow{g} = v(g)$ *and* $\overleftarrow{v}$ *is the inverse of* v *if* v *is injective; otherwise,* $\overleftarrow{v}$ *updates the continuous state* x *to any value in the set* $\overleftarrow{v}(\mathrm{x}) = \{\mathrm{x}' \mid \mathrm{x}' \in g \wedge v(\mathrm{x}') = \mathrm{x}\}$.[3]

Although every HA admits a reverse counterpart according to Definition 7, it is clearly impractical to find a reverse reset function $\overleftarrow{v}(\mathrm{x})$ if v is a one-way function. For an example of reversible HA with non-injective reset functions, see the HA and reverse HA given in Appendix D.4 in [26].

Note that a deterministic HA may admit a nondeterministic reverse HA. Since we classify all states in the state space, we assume that $\mathcal{M}$ and $\overleftarrow{\mathcal{M}}$ can be initialized to any state. We next define the notion of a reverse trajectory $\overleftarrow{\rho}$, which intuitively is obtained by running ρ backwards, starting from ρ's last state and ending with its first state.

Definition 8 (Reverse trajectory). *For HA* $\mathcal{M}$*, time domain* $\mathbb{T} = [0,T]$*, trajectory* ρ *with its corresponding sequence of switching time points* $(\xi_i)_{i=0,\ldots,k} \in \mathbb{T}^{k+1}$*, the* reverse trajectory $\overleftarrow{\rho} = (l(t),\mathbf{x}(t))$ *of* ρ *and its corresponding sequence of switching time points* $\left(\overleftarrow{\xi}_i\right)_{i=0,\ldots,k} \in \mathbb{T}^{k+1}$ *are such that for* $i = 0,\ldots,k$, $\overleftarrow{\xi}_i = T - \xi_{k-i}$*, and* $\forall i < k$, $\overleftarrow{\rho}.l(\overleftarrow{\xi}_i) = \rho.l(\xi_{k-i-1}) \wedge \forall t \in [\overleftarrow{\xi}_i, \overleftarrow{\xi}_{i+1})$, $\overleftarrow{\rho}.l(t) = \overleftarrow{\rho}.l(\overleftarrow{\xi}_i) \wedge \overleftarrow{\rho}.\mathbf{x}(t) = \rho.\mathbf{x}(T-t)$.

Theorem 1. *For an HA* $\mathcal{M}$ *that admits a reverse HA* $\overleftarrow{\mathcal{M}}$*, every trajectory* ρ *of* $\mathcal{M}$ *is reversible, i.e., the reverse trajectory* $\overleftarrow{\rho}$ *of* ρ *is a trajectory of* $\overleftarrow{\mathcal{M}}$*, and every trajectory* $\overleftarrow{\rho}$ *of* $\overleftarrow{\mathcal{M}}$ *is forward-feasible, i.e., the reverse trajectory* ρ *of* $\overleftarrow{\rho}$ *is a trajectory of* $\mathcal{M}$.

Proof. See Appendix A.2 in [26].

Given an unsafe state $u \in U$ of an HA $\mathcal{M}$ that admits a reverse HA $\overleftarrow{\mathcal{M}}$, Theorem 1 allows one to find a positive state $s \in S(\mathcal{M}) \setminus U$ from which u can be reached within time T. The method works by simulating multiple trajectories of $\overleftarrow{\mathcal{M}}$ starting in u and up to time T. In particular, we explore the reverse trajectories from u through an isotropic random walk, i.e., by choosing uniformly at random, at each step of the simulation, the next transition from those available.

3.3 Generation of Training Data and Test Data

We present three sampling methods for generation of training data and test data. Let $\bar{X}$ denote the continuous component of $S(\mathcal{M}) \setminus U$, i.e., without the automaton's location. Recall that model parameters, when present, are expressed as (constant) continuous state variables. They can be sampled independently from the other state variables using appropriate distributions, possibly different from those described below.

Uniform Sampling. When the union of mode invariants covers $\bar{X}$, the algorithm first uniformly samples a continuous state x from $\bar{X}$ and then samples a mode m whose invariant is consistent with x (i.e, $x \in Inv(m)$). When the union of mode invariants

[3] Technically, for v non-injective, $\overleftarrow{v}$ is in general a nondeterministic reset: $\overleftarrow{v} : X \to 2^X$.

does not cover $\bar{X}$, we first uniformly sample the mode m and then a continuous state $x \in Inv(m)$. For simplicity, we restrict attention to cases where the region to be sampled is rectangular, although we could use algorithms for uniform sampling of convex polytopes [20]. We use the reachability checker or the simulator (for deterministic systems) to label the sampled states.

Balanced Sampling. In systems where the unsafe states U are a small part of the overall state space, a uniform sampling strategy produces imbalanced datasets with insufficient positive samples, causing the learned classifier to have relatively low accuracy. For such systems, we generate balanced datasets with equal numbers of negative and positive samples as follows. Negative samples are obtained by uniformly sampling states from $S(\mathcal{M}) \setminus U$ and invoking the reachability checker on those states. In this case, the oracle only needs to verify that the sampled state is negative, i.e., to check that $\mathcal{M} \models \mathsf{Reach}(U, s, T)$ is unsatisfiable. For deterministic systems, the simulator is used instead. Positive samples are obtained by uniformly sampling unsafe states u from U and invoking the backwards simulator from u.

Dynamics-Aware Sampling. This technique generates datasets according to a state distribution expected in a deployed system. It does this by estimating the probability that a state is visited in a trajectory starting from the initial region Init within time T', where $T' > T$. This is accomplished by uniformly sampling states from Init and performing a random exploration of the trajectories from those states up to time T'. The resulting distribution, called *dynamics-aware state distribution*, is estimated from the multiset of states encountered in those trajectories. In our experiments, we estimate a discrete distribution, but other kinds of distributions (e.g., smooth kernel or piecewise-linear) are also supported. The reachability checker or simulator is used to label states sampled from the resulting distribution. This method typically yields highly unbalanced datasets, and thus should not be applied on its own to generate training data.

3.4 Statistical Guarantees with Sequential Hypothesis Testing

Given the infeasibility of training machine-learning models with guaranteed accuracy on unseen data[4], we provide statistical guarantees *a posteriori*, i.e., after training. Inspired by statistical approaches to model checking [24], we employ hypothesis testing to certify that our classifiers meet prescribed levels of accuracy, and FN/FP rates.

We provide guarantees of the form $P_{\mathsf{A}} \geq \theta_{\mathsf{A}}$ (i.e., the true accuracy value is above θ_{A}), $P_{\mathsf{FN}} \leq \theta_{\mathsf{FN}}$ and $P_{\mathsf{FP}} \leq \theta_{\mathsf{FP}}$ (i.e., the true rate of FNs and FPs are below θ_{FN} and θ_{FP}, respectively). Being based on hypothesis testing, such guarantees are precise up to arbitrary error bounds $\alpha, \beta \in (0, 1)$, such that the probability of Type-I errors (i.e., of accepting $P_x < \theta_x$ when $P_x \geq \theta_x$, where $x \in \{\mathsf{A}, \mathsf{FN}, \mathsf{FP}\}$) is bounded by α, and the probability of Type-II errors (i.e., of accepting $P_x \geq \theta_x$ when $P_x < \theta_x$) is bounded by β. The pair (α, β) is known as the *strength* of the test.

To ensure both error bounds simultaneously, the original test $P_x \geq \theta_x$ vs $P_x < \theta_x$ is relaxed by introducing a small indifference region, i.e., we test the hypothesis H_0 :

[4] Statistical learning theory [29] provides statistical bounds on the generalization error of learn models, but these bounds are very conservative and thus of little use in practice. We use these bounds, however, in the proof of Theorem 2.

$P_x \geq \theta_x + \delta$ against $H_1 : P_x \leq \theta_x - \delta$ for some $\delta > 0$. We use Wald's sequential probability ratio test (SPRT) to provide the above guarantees. SPRT has the important advantage that it does not require a prescribed number of samples to accept one of the two hypotheses, but the decision is made if the available samples provide sufficient evidence. Details of the SPRT can be found in Appendix B in [26].

Note that in statistical model checking, SPRT is used to verify that a probabilistic system satisfies a given property with probability above/below a given threshold. In contrast, in NSC, SPRT is used to verify that the probability of the classifier producing the correct prediction meets a given threshold.

3.5 Reducing the False Negative Rate

We discuss strategies to reduce the rate of FNs, the most serious errors from a safety-critical perspective. *Threshold selection* is a simple, yet effective method, which is based on tuning the classification threshold θ of the NN classifier (see Section 3.1). Decreasing θ reduces the number of FNs but may increase the number of FPs and thereby reduce overall accuracy. We evaluate the trade-off between accuracy and FNs in Section 4.2.

Another way to reduce the FN rate is to re-train the classifier with unseen FN samples found in the test stage. For this purpose, we devised a whitebox *falsification-guided adaptation algorithm* that, at each iteration, systematically searches for FNs using *adversarial sampling*; i.e., by solving an optimization problem that seeks to maximize the disagreement between predicted and true reachability values. The optimization problem exploits the knowledge it possesses of the function computed by the NN classifier (whitebox approach). FNs found in this way are used to retrain the classifier. The algorithm iterates until the falsifier cannot find any more FNs.

This approach can be viewed as the dual of counterexample-guided abstraction refinement [9]. CEGAR starts from an abstract model that represents an over-approximation of the system dynamics, and uses counterexamples (FPs) to refine the model, thereby reducing the FP rate. Our approach starts from an under-approximation of the positive region (i.e., the set of states leading to a violation) and uses counterexamples (FNs) to make this region more conservative, reducing the FN rate.

We show that under some assumptions about the performance of the classifier and the falsifier, our algorithm converges to an empty set of FNs. Although it may be difficult in practice to guarantee that these assumptions are satisfied, we also show in Section 4.2 that our algorithm performs reasonably well in practice.

For a state s, let $F(s) \in [0, 1]$ and $b(s) \in \{0, 1\}$ be the NN prediction and true reachability value, respectively. Let FN_k denote the true set of false negatives (i.e., all states s such that $b(s) = 1$ and $F(s) < \theta$) at the k-th iteration of the adaptation algorithm, and let $\hat{FN}_k$ denote the finite subset of FN_k found by the falsifier. The cumulative set of training samples at the k-th iteration of the algorithm is denoted $D_k = D \cup \bigcup_{i=1}^{k} \hat{FN}_k$, where D is the set of samples for the initial training of the classifier.

Assumption 1 *At each iteration k, the classifier correctly predicts positive training samples, i.e., $\forall s \in D_k.\ b(s) = 1 \implies F(s) \geq \theta$, and is such that the FP rate w.r.t. training samples is no larger than the FP rate w.r.t. unseen samples.*

Assumption 2 *At each iteration k, the falsifier can always find an FN when it exists, i.e., $FN_k \neq \emptyset \iff \hat{FN}_k \neq \emptyset$.*

Theorem 2. *Under Assumptions 1–2, the adaptation algorithm converges to an empty set of FNs with high probability, i.e., for all $\eta \in (0, 1)$, $\Pr(\lim_{k\to\infty} FN_k = \emptyset) \geq 1-\eta$.*

Proof. See Appendix A.3 in [26].

We developed a falsifier that uses a genetic algorithm (GA) [25], a nonlinear optimization method for finding multiple global (sub-)optima. In our case, we indeed have multiple solutions because FN samples are found at the decision boundaries of the classifier, separating the predicted positive and negative regions. Due to the real-valued state space, each set FN_k is either empty or infinite.

FN states have $F(s) - b(s) < -\theta$, while FPs are such that $F(s) - b(s) \geq \theta$. By maximizing the absolute discrepancy $|F(s) - b(s)|$, we can identify both FNs and FPs, where only the former are kept for retraining. Specifically, the GA minimizes the objective function $o(s) = 1/(8 \cdot (F(s) - b(s))^2)$ which, for default threshold $\theta = 0.5$, gives a proportionally higher penalty to correctly predicted states ($0.5 \leq o(s) \leq \infty$) than wrong predictions ($0.125 \leq o(s) \leq 0.5$). We retrain the network with all FN candidates found by the GA, not just the optima.

4 Experimental Evaluation

We evaluated our NSC approach on six hybrid-system case studies: a model of the spiking neuron action potential [8], the classic inverted pendulum on a cart, a quadcopter system [15], a cruise controller [8], a powertrain model [19], and a helicopter model [2]. These case studies represent a broad spectrum of hybrid systems and varying degrees of complexity (deterministic, nondeterministic, nonlinear dynamics including trig functions, 2–29 variables, 1–6 modes, 1–11 transitions). Detailed descriptions of the case studies are given in Appendix D in [26].

For all case studies, NSC neural networks were learned using MATLAB's `train` function, with the Levenberg-Marquardt backpropagation algorithm optimizing the mean square error loss function, and the Nguyen-Widrow initialization method for the NN layers. With this setup, we achieved better performance than more standard approaches such as minimizing binary cross entropy using stochastic gradient methods. Training is very fast, taking 2 to 19 seconds for a training dataset with 20,000 samples.

We evaluated the following types of classifiers: sigmoid DNNs (**DNN-S**) with 3 hidden layers of 10 neurons each, with the Tan-Sigmoid activation function for the hidden layers and the Log-Sigmoid activation function for the output layer; shallow NNs (**SNN**), with the same activation functions as **DNN-S** but with one hidden layer of 20 neurons; ReLU DNNs (**DNN-R**), with 3 hidden layers of 10 neurons each, the rectified linear unit (ReLU) activation function for the hidden layers, and the softmax function for the output layer; support vector machines with radial kernel (**SVM**); binary decision trees (**BDT**); and a simple classifier that returns the label of the nearest neighbor in the training set (**NBOR**). We also obtained results for DNN ensembles that combine the predictions of multiple DNNs through majority voting. As expected, ensembles outperformed all of the other classifiers. Due to space limitations, these results are omitted.

We learned the classifiers from relatively small datasets, using training sets of 20K samples and test sets of 10K samples, except where noted otherwise. Larger training sets significantly improved classifier performance for only two of the case studies; see Figure 2. Unless otherwise specified, training and test sets are drawn from the same distribution. The NN architecture (numbers of layers and neurons) was chosen empirically. To avoid overfitting, we did not tune the architecture to optimize the performance for our data. We systematically evaluated other architectures (see Appendix E in [26]), but found no alternatives with consistently better performance than our default configuration of 3 layers and 10 neurons. We also experimented with 1D Convolutional Neural Networks (CNNs), but they performed worse than the DNN architectures.

In the following, when clear from the context, we omit the modifier "empirical" when referring to accuracy, FN, and FP rates over a test dataset (as opposed to the true accuracy over the state distribution).

4.1 Performance Evaluation

Table 1 shows empirical accuracy and FN rate for all classifiers and case studies, using uniform and balanced sampling. We obtain very high classification accuracy for neuron, pendulum, quadcopter and cruise. For these case studies, DNN-based classifiers registered the best performance, with accuracy values ranging between 99.48 % and 99.98 % and FN rates between 0.24% and 0%. Only a minor performance degradation is observed for the shallow neural network **SNN**, with accuracy in the range 98.89-99.85%.

In contrast, the accuracy for the helicopter and powertrain models is poor if we use only 20K training samples. These models are indeed particularly challenging, owing to their high dimensionality (helicopter) and highly nonlinear dynamics (powertrain). Larger training sets provide considerable improvement in accuracy and FN rate, as shown in Figure 2. For helicopter, accuracy jumps from 98.49% (20K samples) to 99.92% (1M samples), and the FN rate decreases from 0.84% (20K) to 0.04% (1M). For powertrain, accuracy increases from 96.68% (20K) to 99.25% (1M), and the FN rate decreases from 1.28% (20K) to 0.33% (1M).

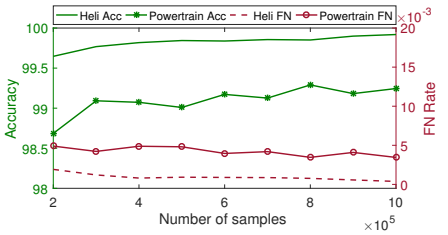


Fig. 2: Performance of DNN-S classifier on helicopter and powertrain models with varying numbers of training samples (uniform sampling).

In general, we found that the NN-based classifiers have superior accuracy compared to support vector machines and binary decision trees. As expected, the nearest-neighbor method demonstrated poor prediction capabilities. No single sampling method provides a clear advantage over the others in terms of accuracy, most likely because training and test sets are drawn from the same distribution.

Dynamics-aware state distribution. To evaluate the behavior of the classifiers with the dynamics-aware state distribution (introduced in Section 3.3), we generate training data with a combination of dynamics-aware sampling and either uniform or balanced sampling, because dynamics-aware sampling alone yields unbalanced datasets unsuitable for training. Test data consists exclusively of dynamics-aware samples.

	Neuron		Pendulum		Quadcopter		Cruise		Powertrain		Helicopter		
	Acc	FN	Acc	FN	Acc	FN	Acc	FN	Acc	FN	Acc	FN	
DNN-S	**99.81**	**0.1**	**99.98**	**0**	99.83	0.1	99.95	0.01	**96.68**	1.28	**98.49**	**0.84**	
DNN-R	99.52	0.29	99.93	0.04	**99.89**	**0.06**	**99.98**	**0**	96.21	**1.08**	98	0.96	**Uniform**
SNN	99.17	0.43	99.81	**0**	99.85	0.08	99.84	0.15	96.02	1.37	97.69	1.25	
SVM	98.73	0.75	99.84	**0**	97.33	0.69	99.88	0.1	92.26	3.48	95.58	2.42	
BDT	99.3	0.37	99.6	0.17	99.52	0.2	99.84	0.08	95.59	2.19	80.07	9.8	
NBOR	97.03	1.22	99.69	0.14	99.53	0.25	99.49	0.33	71.44	14.51	67.39	16.98	
	Acc	FN	Acc	FN	Acc	FN	Acc	FN	Acc	FN	Acc	FN	
DNN-S	**99.83**	**0.12**	**99.89**	**0**	**99.82**	0.04	99.94	**0**	**97.2**	**0.86**	**98.24**	**0.79**	
DNN-R	99.48	0.24	99.63	0.01	99.67	0.09	**99.95**	**0**	96.07	1.24	97.91	1.2	**Balanced**
SNN	98.89	0.69	99.2	**0**	99.49	**0.01**	99.6	**0**	95.21	1.79	97.58	1.16	
SVM	98.63	0.78	99.37	**0**	96.93	0.2	99.61	**0**	91.84	3.3	95.36	1.85	
BDT	99.07	0.45	99.46	0.05	99.36	0.22	99.9	0.03	95.86	2.4	79.03	10.26	
NBOR	96.95	1.62	99.51	0.04	99.11	0.56	99.47	0.11	71.33	13.99	65.18	17.48	

Table 1: Empirical accuracy (Acc) and FN rate of the state classifiers for each case study, classifier type, and sampling method. Values are in percentages. For each measure and sampling method, the best result is highlighted in bold. False positives and confidence intervals are reported in Tables 5 and 6 of the Appendix provided in [26].

Table 2 shows that the classifiers yield accuracy values comparable to those of Table 1 (compiled with balanced and uniform distributions) for all case studies. We see that the powertrain model attains 100% accuracy, indicating that its dynamics-aware distribution favors states that are easy enough for the DNN to classify correctly.

	Neuron	Pendulum	Quadcopter	Cruise	Helicopter	Powertrain
Unif+Dyn-aware	99.91 (+0.1)	99.93 (-0.05)	99.84 (+0.01)	99.14 (-0.81)	98.77 (+0.28)	100 (+3.32)
Bal+Dyn-aware	99.8 (-0.03)	99.88 (-0.01)	99.79 (-0.03)	99.35 (-0.59)	98.46 (+0.22)	100 (+2.8)

Table 2: Empirical accuracy of DNN-S classifiers tested on 10K dynamics-aware samples and trained with 20K samples. Each row corresponds to a different training distribution. **Unif+Dyn-aware** and **Bal+Dyn-aware** were obtained by combining 10K uniform/balanced samples with 10K dynamics-aware samples. In parenthesis is the accuracy difference with the corresponding classifier from Table 1.

Parametric analysis. We show that NSC works effectively for parametric systems, being able to classify states in models with parameter values not seen during training. We derive parametric versions of the neuron model by turning constants a, b, c, d, I (see Appendix D.1 in [26]) into parameters uniformly distributed in the $\pm$ 50% interval around their default value.

	Num. of parameters				
	1	**2**	**3**	**4**	**5**
$\hat{P}_A$	99.8	99.7	97.9	98.1	97.8
$\hat{P}_{FN}$	0.2	0.2	1.6	1.3	1.5

Table 3: Empirical accuracy ($\hat{P}_A$) and FN rate ($\hat{P}_{FN}$) for DNN-S classifier for neuron model with increasing number of parameters.

Table 3 shows the accuracy and FN rates for DNN-S, trained with 110K samples for models with increasing numbers of parameters, which are increasingly long prefixes of the sequence a, b, c, d, I. We achieve very high accuracy ($\geq$ 99.7%) for up to two parameters. For three to five parameters, the accuracy decreases but stays relatively high (around 98%), suggesting that larger training sets are required for these cases.

Indeed the input space grows exponentially in the number of parameters, while we kept the size of the training set constant.

Statistical guarantees. We use SPRT (Section 3.4) to provide statistical guarantees for four case studies, each trained with 20K balanced samples. See Table 4. We assess two properties certifying that the *true* (not empirical) accuracy and FNs meet given performance levels: $P_{\mathsf{A}} \geq 99.7\%$, and $P_{\mathsf{FN}} \leq 0.2\%$. We omit the helicopter and powertrain models from this assessment, because performance results for these models are clearly outside the desired levels when only 20K samples are used for training.

The only classifier that guarantees these performance levels for all case studies is the sigmoid DNN. We also observe that a small number of samples suffices to obtain statistical guarantees with the given strength: only 3 out of 48 tests needed more than 10K samples to reach a decision.

	Neuron		**Pendulum**		**Quadcopter**		**Cruise**	
	$P_{\mathsf{A}} \geq \theta_{\mathsf{A}}$	$P_{\mathsf{FN}} \leq \theta_{\mathsf{FN}}$	$P_{\mathsf{A}} \geq \theta_{\mathsf{A}}$	$P_{\mathsf{FN}} \leq \theta_{\mathsf{FN}}$	$P_{\mathsf{A}} \geq \theta_{\mathsf{A}}$	$P_{\mathsf{FN}} \leq \theta_{\mathsf{FN}}$	$P_{\mathsf{A}} \geq \theta_{\mathsf{A}}$	$P_{\mathsf{FN}} \leq \theta_{\mathsf{FN}}$
DNN-S	✓ (5800)	✓ (2900)	✓ (2300)	✓ (2300)	✓ (4400)	✓ (2300)	✓ (3000)	✓ (2300)
DNN-R	✗ (3600)	✗ (8600)	✓ (15500)	✓ (4000)	✗ (1400)	✓ (7300)	✓ (3000)	✓ (2300)
SNN	✗ (700)	✗ (1000)	✗ (2900)	✓ (2300)	✗ (1500)	✓ (3400)	✗ (3600)	✓ (2300)
SVM	✗ (400)	✗ (600)	✗ (6600)	✓ (2300)	✗ (200)	✗ (5300)	✗ (3400)	✓ (2300)
BDT	✗ (1700)	✗ (3300)	✗ (6300)	✓ (15000)	✗ (800)	✗ (1100)	✓ (2700)	✓ (2900)
NBOR	✗ (300)	✗ (300)	✗ (28500)	✓ (2900)	✗ (1000)	✗ (1300)	✗ (3400)	✗ (2300)

Table 4: Statistical guarantees based on the SPRT. Samples were generated using balanced sampling. In parenthesis are the number of samples required to reach the decision. Parameters of the test are $\alpha = \beta = 0.01$ and $\delta = 0.001$. Thresholds are $\theta_{\mathsf{A}} = 99.7\%$ and $\theta_{\mathsf{FN}} = 0.2\%$.

4.2 Reducing the False Negative Rate

Falsification-guided adaptation. We evaluate the benefits of adaptation by incrementally adapting the trained NNs with false negative samples (see Section 3.5). At each iteration, we run our GA-based falsifier to find FN samples, which are then used to adapt the DNN. The adaptation loop terminates when the falsifier cannot find a FN.

We employ MATLAB's `adapt` function with gradient descent learning algorithm and learning rates of 0.0005 for neuron and 0.003 for quadcopter, helicopter, and powertrain. For neuron and quadcopter, we use DNN-S classifiers trained with 20K balanced samples. We use DNN-S trained with 1M balanced samples for helicopter, and DNN-S trained with 1M uniform samples for powertrain, because these classifiers have the best accuracy before adaptation. To measure adaptation performances, we test the DNNs on 10K samples after each iteration of adaptation. Fig. 3 shows how accuracy, FNs and FPs of the classifier evolve at each adaptation step. For the neuron, quadcopter, and helicopter case studies, our falsification-guided adaptation algorithm works well to eliminate the FN rate at the cost of a slight increase in the FP rate after only 5-10 iterations. In these case studies, the number of FNs found by the falsifier decreases quickly from hundreds or thousands to zero. For powertrain, the number of FNs found by the falsifier stays almost constant at about 70 on average at each iteration. After 150 iterations, FN rate of the powertrain DNN decreases slowly from 0.33% to 0.15%.

Figure 4 visualizes the effects of adaptation on the **DNN-S** classifier for the neuron case study. Fig. 4 (a) shows the prediction of the DNN after training with 20K samples. Fig. 4 (b) shows the prediction of the DNN after adaptation. We see that adaptation expands the predicted positive region to enclose all previous FN samples, i.e., they are correctly re-classified as positive. The enlarged positive region also means the adapted DNN is more conservative, producing more FPs as shown in Fig. 4 (b).

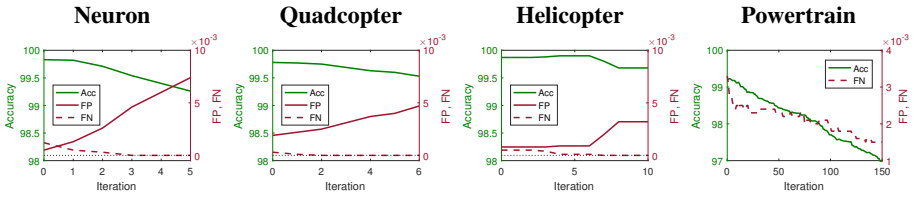
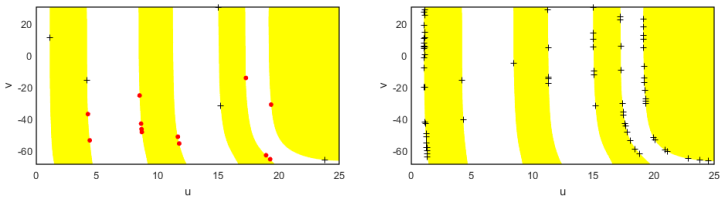


Fig. 3: Impact of incremental adaptation on empirical accuracy, FN and FP rates. FP-rate curve for powertrain is omitted to allow using a scale that shows the decreasing trend of the FN rate. The FP rate for powertrain increases from 0.48% to 2.89%.



(a) Before adaptation (b) After adaptation

Fig. 4: Effects of adaptation on the DNN-S for the neuron case study. The white region is the predicted negative region. The yellow region is the predicted positive region. Red dots are FN samples. Crosses are FP samples.

Threshold selection. We show that threshold selection can considerably reduce the FN rate. Figure 5 shows the effect of threshold selection on accuracy, FN rate, and FP rate for classifier **DNN-S** trained with uniform sampling (20K samples for neuron and quadcopter, 1M samples for helicoper and powertrain). Pendulum and cruise control case studies are excluded as they have low FN rate ($\leq 0.01\%$) prior to threshold selection.

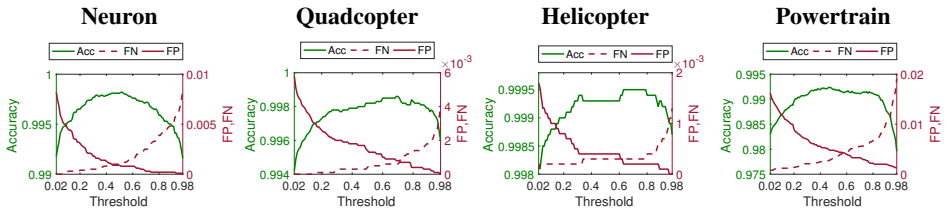


Fig. 5: Impact of classification threshold on empirical accuracy, FN rate, and FP rate.

For the neuron case study, selecting $\theta = 0.32$ reduces the FN rate from 10^{-3} to $5 \cdot 10^{-4}$, with an accuracy loss of only 0.02%. With $\theta = 0.06$, we obtain a zero FN rate

and a minor accuracy loss of 0.37%. For quadcopter, selecting $\theta = 0.28$ decreases the FN rate from $4 \cdot 10^{-4}$ to 10^{-4}, with an accuracy loss of just 0.02%. Selecting $\theta = 0.16$ yields zero FN rate and accuracy loss of just 0.12%. For helicopter, selecting $\theta = 0.33$ reduces the FN rate from $3 \cdot 10^{-4}$ to $2 \cdot 10^{-4}$, with an *accuracy gain* of 0.01%. For powertrain, $\theta = 0.34$ yields a good trade-off between FN rate reduction (from $3.3 \cdot 10^{-3}$ to $2.1 \cdot 10^{-3}$) and accuracy loss (0.1%).

5 Related Work

Related work includes techniques for *simulation-based verification*, which enables rigorous system analysis from finitely many executions. Statistical model checking [24] for the verification of probabilistic systems with statistical guarantees is an example of this form of verification. Simulation is also used for falsification and reachability analysis of hybrid systems [1,2]. Our NSC approach also simulates system executions (when the system is deterministic), but for the purpose of learning a state classifier.

Other applications of *machine learning in verification* include parameter synthesis of stochastic systems [5], techniques for inferring temporal logic specifications from examples [4], synthesis of invariants for program verification [28,14], and reachability checking of Markov decision processes [6].

For safety-critical applications, *verification of NNs* has become a very active area, with a focus on the derivation of adversarial inputs (i.e., those that induce incorrect predictions). Most such approaches rely on SMT-based techniques [18,21,11], while sampling-based methods are used in [10] for the analysis of NN components "in the loop" with cyber-physical system models. Similarly, our adaptation method systematically searches for adversarial inputs (FNs) to render the classifier more conservative. A related problem is that of range estimation [30], i.e., computing safe and tight enclosures for the predictions of an NN over a (convex) input region. Such methods could be used to extend NSC classification to sets of states.

6 Conclusions

We have introduced the state classification problem for hybrid systems and offered a highly efficient solution based on neural state classification. NSC features high accuracy and low false-negative rates, while including techniques for virtually eliminating such errors and for certifying an NSC classifier's performance with statistical guarantees. Plans for future work include considering more expressive temporal properties and extending our approach to stochastic hybrid systems.

References

1. Annpureddy, Y., et al.: S-TaLiRo: A tool for temporal logic falsification for hybrid systems. In: TACAS. vol. 6605, pp. 254–257. Springer (2011)
2. Bak, S., Duggirala, P.S.: Rigorous simulation-based analysis of linear hybrid systems. In: TACAS. pp. 555–572. Springer (2017)
3. Bak, S., et al.: Hybrid automata: From verification to implementation. International Journal on Software Tools for Technology Transfer pp. 1–18 (2017)

4. Bartocci, E., Bortolussi, L., Sanguinetti, G.: Data-driven statistical learning of temporal logic properties. In: FORMATS. pp. 23–37. Springer (2014)
5. Bortolussi, L., Silvetti, S.: Bayesian statistical parameter synthesis for linear temporal properties of stochastic models. In: TACAS. pp. 396–413 (2018)
6. Brázdil, T., et al.: Verification of Markov decision processes using learning algorithms. In: ATVA. pp. 98–114. Springer (2014)
7. Brihaye, T., et al.: On reachability for hybrid automata over bounded time. In: ICALP. pp. 416–427. Springer (2011)
8. Chen, X., et al.: A benchmark suite for hybrid systems reachability analysis. In: NFM. pp. 408–414. Springer (2015)
9. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: CAV. pp. 154–169. Springer (2000)
10. Dreossi, T., Donzé, A., Seshia, S.A.: Compositional falsification of cyber-physical systems with machine learning components. In: NFM. pp. 357–372. Springer (2017)
11. Ehlers, R.: Formal verification of piece-wise linear feed-forward neural networks. In: ATVA. pp. 269–286 (2017)
12. Frehse, G., et al.: SpaceEx: Scalable verification of hybrid systems. In: CAV. pp. 379–395. Springer (2011)
13. Gao, S., Kong, S., Clarke, E.M.: dReal: An SMT solver for nonlinear theories over the reals. In: International Conference on Automated Deduction. pp. 208–214. Springer (2013)
14. Garg, P., et al.: Learning invariants using decision trees and implication counterexamples. ACM Sigplan Notices 51(1), 499–512 (2016)
15. Gibiansky, A.: Quadcopter dynamics and simulation (2012), `http://andrew.gibiansky.com/blog/physics/quadcopter-dynamics/`
16. Henzinger, T.A., et al.: What's decidable about hybrid automata? In: STOC. pp. 373–382 (1995)
17. Hornik, K., Stinchcombe, M., White, H.: Multilayer feedforward networks are universal approximators. Neural networks 2(5), 359–366 (1989)
18. Huang, X., et al.: Safety verification of deep neural networks. In: CAV. pp. 3–29 (2017)
19. Jin, X., et al.: Powertrain control verification benchmark. In: HSCC. pp. 253–262 (2014)
20. Kannan, R., Lovász, L., Simonovits, M.: Random walks and an $o^*(n^5)$ volume algorithm for convex bodies. Random structures and algorithms 11(1), 1–50 (1997)
21. Katz, G., et al.: Reluplex: An efficient SMT solver for verifying deep neural networks. In: CAV. pp. 97–117 (2017)
22. Kwiatkowska, M., Norman, G., Parker, D.: Stochastic model checking. In: SFM. pp. 220–270. Springer (2007)
23. LeCun, Y., Bengio, Y., Hinton, G.: Deep learning. Nature 521(7553), 436 (2015)
24. Legay, A., Delahaye, B., Bensalem, S.: Statistical model checking: An overview. In: RV. pp. 122–135. Springer (2010)
25. Mitchell, M.: An introduction to genetic algorithms. MIT press (1998)
26. Phan, D., Paoletti, N., Zhang, T., Grosu, R., Smolka, S.A., Stoller, S.D.: Neural state classification for hybrid systems. ArXiv e-prints (Jul 2018)
27. Sen, K., Rosu, G., Agha, G.: Online efficient predictive safety analysis of multithreaded programs. In: TACAS. vol. 2988, pp. 123–138. Springer (2004)
28. Sharma, R., et al.: A data driven approach for algebraic loop invariants. In: ESOP. pp. 574–592. Springer (2013)
29. Vapnik, V.: The nature of statistical learning theory. Springer (2013)
30. Xiang, W., Tran, H.D., Johnson, T.T.: Output reachable set estimation and verification for multi-layer neural networks. arXiv preprint arXiv:1708.03322 (2017)