arXiv:2312.10842v1 [cs.LO] 17 Dec 2023

Compositional Inductive Invariant Based Verification of Neural Network Controlled Systems

Yuhao Zhou and Stavros Tripakis

Northeastern University, Boston, MA, USA
{zhou.yuhao, stavros}@northeastern.edu

Abstract. The integration of neural networks into safety-critical systems has shown great potential in recent years. However, the challenge of effectively verifying the safety of Neural Network Controlled Systems (NNCS) persists. This paper introduces a novel approach to NNCS safety verification, leveraging the inductive invariant method. Verifying the inductiveness of a candidate inductive invariant in the context of NNCS is hard because of the scale and nonlinearity of neural networks. Our compositional method makes this verification process manageable by decomposing the inductiveness proof obligation into smaller, more tractable subproblems. Alongside the high-level method, we present an algorithm capable of automatically verifying the inductiveness of given candidates by automatically inferring the necessary decomposition predicates. The algorithm significantly outperforms the baseline method and shows remarkable reductions in execution time in our case studies, shortening the verification time from hours (or timeout) to seconds.

Keywords: Formal Verification · Inductive Invariant · Neural Networks · Neural Network Controlled Systems

1 Introduction

Neural Network Controlled Systems (NNCSs) are closed-loop systems that consist of an *environment* controlled by a *neural network* (NN). Advancements in machine learning have made NNCSs more widespread in safety-critical applications, such as autonomous cars, industrial control, and healthcare [30]. While many methods exist for the formal verification of standard closed-loop systems [4], the scale and nonlinearity of NNs makes the direct application of these methods to NNCSs a challenging task.

In this paper, we study formal verification of safety properties of NNCSs, using the *inductive invariant method* [18]. In a nutshell, the method consists in coming up with a state predicate *IndInv* which is *inductive*, an *invariant*, and implies safety (see Section 2, Conditions (4), (5) and (6)). The most important property, and often the most difficult to check, is inductiveness. In the case of

NNCSs, checking inductiveness amounts to checking the validity of the following formula, where Next_{NNC} and Next_{ENV} are the transition relation predicates of the NN controller and of the environment, respectively, and ϕ' is the state predicate ϕ applied to the next-state (primed) variables:

$$(\mathit{IndInv} \wedge \mathit{Next}_{NNC} \wedge \mathit{Next}_{ENV}) \implies \mathit{IndInv}' \tag{1}$$

The problem that motivates our work is that checking formula (1) is infeasible in practice using state-of-the-art tools. On the one hand, generic tools such as SMT solvers [5] cannot handle formula 1 directly, typically due to the size and complexity of the NN and corresponding Next_{NNC} predicate. On the other hand, specialized NN verification tools [28, 29] also cannot handle formula (1) directly, typically due to limitations in being able to deal with closed-loop systems and the environment transition relation Next_{ENV} (which, for example, might be non-deterministic).

This paper proposes a *compositional* method to deal with this problem. The key idea is the following: instead of checking formula (1) *monolithically*, we propose to break it up into two separate conditions:

$$(\mathit{IndInv} \wedge \mathit{Next}_{NNC}) \implies \mathit{Bridge} \tag{2}$$

$$(\mathit{Bridge} \wedge \mathit{Next}_{ENV}) \implies \mathit{IndInv}' \tag{3}$$

where *Bridge* is a new state predicate that needs to be invented (we propose a technique to do this automatically). Not only are each of the formulas (2) and (3) smaller than the monolithic formula (1) they can also be handled by the corresponding specialized tool: formula (2) by a NN verifier, and formula (3) by an SMT solver.

In Section 3 we elaborate our approach. We propose a technique to construct *Bridge* predicates automatically, and show how this technique can also be used to establish the validity of Condition (2) by construction. We also present a heuristic that can falsify inductiveness compositionally in some cases when formula (1) is not valid. We incorporate these techniques into an algorithm that checks inductiveness of candidate *IndInv* predicates for NNCSs.

Our experimental results (Section 4) indicate that our algorithm consistently terminates, effectively verifying or falsifying the inductiveness of the given candidate in both deterministic and non-deterministic environment setups, for NNs of sizes up to 2×1024 neurons, in a matter of seconds. This is in contrast to the monolithic method, which times out after one hour for all our experiments with NNs larger than 2×56 neurons.

2 Preliminaries and Problem Statement

Symbolic Transition Systems: The systems considered in this paper can be modeled as *symbolic transition systems* (STSs), like the ones shown in Figure 1. An STS is defined by: (1) a set of *state variables*, each of appropriate type; (2) a predicate *Init* over the state variables, specifying the set of possible *initial*

State variable : $s \in \mathbb{Z}$
Init : $s = 0$
Next : $(s = 3 \implies s' = 0) \wedge$
$(s \neq 3 \implies s' = s + 1)$
Safe : $0 \leq s \leq 3$

(a) Mod-4 counter

State variable : $s \in \mathbb{Z}$
Init : $s = 0$
NN function : $f_{NN} : \mathbb{Z} \to \mathbb{R}$
Next_{NNC} : $a = f_{NN}(s)$
Next_{ENV} : $(a \geq 0 \implies s' = 0) \wedge$
$(a < 0 \implies s' = s + 1)$
Next : $\mathit{Next}_{NNC} \wedge \mathit{Next}_{ENV}$
Safe : $0 \leq s \leq 3$

(b) Mod-4 counter with NN controller

Fig. 1: Two STSs. The one in Fig. 1b is a NNCS: function f_{NN} (defined elsewhere) models a neural network controller.

states; and (3) a predicate *Next* over current and next (primed) state variables, specifying the *transition relation* of the system. For example, the STS shown in Figure 1a models a counter modulo-4. This system has one state variable s of type integer ($\mathbb{Z}$). Its *Init* predicate specifies that s is initially 0. *Next* specifies that at each transition, s is incremented by one until we reach $s = 3$, upon which s is reset to 0 (primed variable s' denotes the value of state variable s at the next state). The *Safe* predicate defines the set of *safe* states (used later).

A *state* is an assignment of values to all state variables. We use the notation $x \models P$ to denote that state x satisfies state predicate P, i.e., that P evaluates to true once we replace all state variables in P by their values as given by x. A *transition* is a pair of states (x, x') that satisfies the predicate *Next*, denoted $(x, x') \models \mathit{Next}$. We also use notation $x \to x'$ instead of $(x, x') \models \mathit{Next}$, when *Next* is clear from context. A *trace* is an infinite sequence of states $x_0, x_1, \cdots$ such that $x_0 \models \mathit{Init}$ and $x_i \to x_{i+1}$ for all $i \geq 0$. A state x is *reachable* if there exists a trace $x_0, x_1, \cdots$, such that $x = x_i$ for some i. We use $\mathit{Reach}(M)$ to denote the set of all reachable states of an STS M.

Invariants: In this paper we are interested in the verification of safety properties, and in particular *invariants*, which are state predicates that hold at all reachable states. Formally, for an STS M, an invariant is a state predicate *Inv* such that $\mathit{Reach}(M) \subseteq \{s \mid s \models \mathit{Inv}\}$. Predicates $s \geq 0$ and $0 \leq s \leq 3$ are both invariants of the STS of Figure 1a.

Inductive Invariants: A standard technique for checking whether a given state predicate *Safe* is an invariant of a given STS is to come up with an *inductive invariant* stronger than *Safe*, that is, with a state predicate *IndInv* which satisfies the following conditions [18]:

$$\begin{aligned} \mathit{Init} &\implies \mathit{IndInv} && (4)\\ (\mathit{IndInv} \wedge \mathit{Next}) &\implies \mathit{IndInv}' && (5)\\ \mathit{IndInv} &\implies \mathit{Safe} && (6)\end{aligned}$$

where *IndInv*$'$ denotes the predicate *IndInv* where state variables are replaced by their primed, next-state versions. Condition (4) states that *IndInv* holds at all initial states. Condition (5) states that *IndInv* is *inductive*, that is, if *IndInv* holds at a state s, then it also holds at any successor of s. We call Condition (5) the *inductiveness condition*. Condition (6) states that *IndInv* is stronger than *Safe*. Conditions (4) and (5) imply that all reachable states satisfy *IndInv*, which together with Condition (6) implies that they also satisfy *Safe*.

Neural Networks: At its core, a *neural network* (NN) is a function, receiving inputs and producing outputs. Mathematically, the output $z^{(i)}$ of each layer in a feed-forward NN with L layers can be expressed as:

$$z^{(i)} = \sigma_i(W^{(i)} z^{(i-1)} + b^{(i)}), \quad i = 1, 2, \cdots, L,$$

where $W^{(i)}$ and $b^{(i)}$ represent the weight matrix and bias vector for the i-th layer, respectively, and σ_i is the activation function for that layer. Our paper considers feed-forward NNs with ReLU activation functions, where $\mathrm{ReLU}(x) = \max(0, x)$.

Neural Network Controlled Systems: In this paper, we are interested in safety verification for *neural network controlled systems* (NNCSs). A NNCS is a STS consisting of a *neural network controller* and an *environment*, in closed-loop configuration. Formally, a NNCS is a STS whose transition relation predicate *Next* is of the form $\mathit{Next}_{NNC} \wedge \mathit{Next}_{ENV}$, where Next_{NNC} is a predicate capturing the NN controller, and Next_{ENV} is a predicate capturing the transition relation of the environment. We assume that Next_{NNC} is always of the form $\boldsymbol{a} = f_{NN}(\boldsymbol{s})$, where f_{NN} is the function modeling the NN controller, $\boldsymbol{s}$ is the vector of state variables of the environment (which are the inputs to the NN controller), and $\boldsymbol{a}$ is the vector of outputs of the NN controller (which are the inputs to the environment). Then, Next_{ENV} is a predicate on $\boldsymbol{s}$, $\boldsymbol{s'}$, and $\boldsymbol{a}$. Note that the only state variables in the NNCS are $\boldsymbol{s}$. Variables $\boldsymbol{a}$ are not state variables, but just temporary variables that can be eliminated and replaced by $f_{NN}(\boldsymbol{s})$, according to the equation $\boldsymbol{a} = f_{NN}(\boldsymbol{s})$. We assume that for any assignment of $\boldsymbol{s}$ and $\boldsymbol{a}$, there always exists an assignment of $\boldsymbol{s'}$ such that Next_{ENV} is satisfied; that is, the system is deadlock-free.

An example NNCS is shown in Figure 1b. In this simple example, the output a of the NN controller controls the environment to either reset the state variable s to 0 (if $a \geq 0$) or increment it by 1 (if $a < 0$).

Safety Verification Problem for NNCS: The problem we study in this paper is the safety verification problem for NNCS, namely: *given a NNCS M and a safety predicate Safe, check whether Safe is an invariant of M.*

3 Our Approach

To solve the safety verification problem for NNCSs, we will use the inductive invariant method described in Section 2. In particular, our approach assumes that a *candidate* inductive invariant is given, and our focus is to *check* whether this candidate is indeed a valid inductive invariant stronger than *Safe*, i.e., whether it satisfies Conditions (4), (5), and (6). Specifically, we focus on checking *inductiveness*, i.e., Condition (5), because this is the most challenging condition to check.

In the case of NNCS, Condition (5) instantiates to:

$$(\mathit{IndInv} \land \mathit{Next}_{\mathit{NNC}} \land \mathit{Next}_{\mathit{ENV}}) \implies \mathit{IndInv}'. \tag{7}$$

A naive approach is to attempt to check Condition (7) directly: we call this the *monolithic method*. Unfortunately, as we will show in Section 4, the monolithic method does not scale. This is typically because of the size of the $\mathit{Next}_{\mathit{NNC}}$ part of the formula, which encodes the NN controller, and tends to be very large. To address this challenge, we introduce a *compositional method* for checking inductiveness, described next.

3.1 Our Approach: Compositional Method

Our compositional method is centered around two key ideas: (i) automatically construct a *bridge predicate*, denoted *Bridge*; and (ii) replace the monolithic inductiveness Condition (7) by two separate conditions:

$$(\mathit{IndInv} \land \mathit{Next}_{\mathit{NNC}}) \implies \mathit{Bridge} \tag{8}$$

$$(\mathit{Bridge} \land \mathit{Next}_{\mathit{ENV}}) \implies \mathit{IndInv}' \tag{9}$$

By transitivity of logical implication, it is easy to show the following:

Theorem 1 (Soundness). *If Conditions (8) and (9) hold then Condition (7) holds.*

The completeness of our method follows from the fact that in the worst case we can set *Bridge* to be equal to $\mathit{IndInv} \land \mathit{Next}_{\mathit{NNC}}$.

Theorem 2 (Completeness). *If Condition (7) holds then we can find Bridge such that Conditions (8) and (9) hold.*

As it turns out (c.f. Section 4) checking Conditions (8) and (9) separately scales much better than checking Condition (7) monolithically. However, this relies on finding a “good” bridge precidate. Setting *Bridge* to $\mathit{IndInv} \land \mathit{Next}_{\mathit{NNC}}$ is not helpful, because then Condition (9) becomes identical to the monolithic Condition (7). Thus, a necessary step is finding a bridge predicate that satisfies Conditions (8) and (9), while remaining manageable. In Section 3.2, we present an automatic technique for doing so. But first, we examine bridge predicates in more depth.

Naive Bridge Predicates are Incomplete: A natural starting point for constructing the bridge predicate is to define it only over the output variables $\boldsymbol{a}$ of the NN controller. We call this a *naive bridge predicate*. Unfortunately, naive bridge predicates might be insufficient, as we show next. Consider the system shown in Figure 1b. Suppose we set $Next_{NNC}$ to:

$$((0 \leq s \leq 2) \wedge a = -1) \vee (s = 3 \wedge a = 1) \vee ((s < 0 \vee s > 3) \wedge (a = -1 \vee a = 1))$$

It can be checked that the set of reachable states of this system is $0 \leq s \leq 3$. Therefore, $0 \leq s \leq 3$ is also an inductive invariant of this system. A naive bridge predicate that satisfies Condition (8) could be $a = 1 \vee a = -1$. This is, in fact, the strongest naive bridge predicate that satisfies Condition (8), as it captures all the possible values of a. Therefore, if this naive bridge predicate does not satisfy Condition (9), then no naive bridge predicate that satisfies both Conditions (8) and (9) exists. Indeed, Condition (9) is violated by this naive bridge, as it becomes

$$\begin{aligned}((a = 1 \vee a = -1) \wedge (a \geq 0 \implies s' = 0) \wedge (a < 0 \implies s' = s + 1))\\ \implies (0 \leq s' \leq 3)\end{aligned}$$

which is false when $s = 3$, $a = -1$ and $s' = 4$. So, no naive bridge predicate can satisfy both Condition (8) and (9). This suggests that naive bridge predicates are insufficient. The solution is to allow bridge predicates to refer both to the inputs and outputs of the NN controller:

Generalized Bridge Predicates are Complete: A *generalized bridge predicate* is a predicate defined over both the output variables $\boldsymbol{a}$ as well as the input variables $\boldsymbol{s}$ of the NN controller (the inputs $\boldsymbol{s}$ are the same as the state variables of the environment). Continuing our example above, a generalized bridge predicate could be: $((0 \leq s \leq 2) \wedge a = -1) \vee (s = 3 \wedge a = 1)$. Then, Conditions (8) and (9) become:

$$\begin{aligned}&((0 \leq s \leq 3) \quad \wedge\\ &((0 \leq s \leq 2) \wedge a = -1) \vee (s = 3 \wedge a = 1) \vee ((s < 0 \vee s > 3) \wedge (a = -1 \vee a = 1)))\\ &\implies (((0 \leq s \leq 2) \wedge a = -1) \vee (s = 3 \wedge a = 1))\end{aligned} \tag{10}$$

$$\begin{aligned}&(((0 \leq s \leq 2) \wedge a = -1) \vee (s = 3 \wedge a = 1)) \wedge (a \geq 0 \implies s' = 0)\\ &\wedge\ (a < 0 \implies s' = s + 1))\\ &\implies (0 \leq s' \leq 3).\end{aligned} \tag{11}$$

and it can be checked that both are valid, which shows that this generalized bridge predicate is sufficient.

In general, and according to Theorem 2, a generalized bridge predicate is sufficient, since we can set *Bridge* to $IndInv \wedge Next_{NNC}$, and the latter predicate is over $\boldsymbol{s}$ and $\boldsymbol{a}$. But it is important to note that a bridge that works need not be the "worst-case scenario" predicate $\mathit{IndInv} \wedge \mathit{Next}_{\mathit{NNC}}$. Indeed, the bridge of our example is not, and neither are the bridges constructed by our tool for the case studies in Section 4.

3.2 Automatic Inference of Generalized Bridge Predicates

The key idea of our automatic bridge inference technique is to synthesize a generalized bridge predicate such that Condition (8) holds *by construction* (i.e., it does not need to be checked). To explain how this works, let us first define, given a predicate A over both $\boldsymbol{s}$ and $\boldsymbol{a}$, and predicate B over only $\boldsymbol{s}$, the *set of postconditions*, denoted $\mathrm{AllPosts}(A, B)$, to be the set of all predicates C over $\boldsymbol{a}$ such that $(A \wedge B) \implies C$ is valid.

Then, our algorithm will construct a bridge predicate the has the form:

$$\mathit{Bridge} = (p_1 \wedge \psi_1) \vee (p_2 \wedge \psi_2) \vee \cdots \vee (p_n \wedge \psi_n) \tag{12}$$

where each p_i is a predicate over $\boldsymbol{s}$, and each ψ_i is a predicate over $\boldsymbol{a}$, such that the following conditions hold:

$$(p_1 \vee p_2 \vee \cdots \vee p_n) \iff \mathit{IndInv} \tag{13}$$

$$\forall i = 1, ..., n, \;\; \psi_i \in \mathrm{AllPosts}(\mathit{Next}_{\mathit{NNC}}, p_i) \tag{14}$$

Condition (13) ensures that the set of all p_i's is a *decomposition* of the candidate inductive invariant *IndInv*, i.e., that their union, viewed as sets, "covers" *IndInv*. This decomposition need not be a partition of *IndInv*, i.e., the p_i's need not be disjoint. Condition (14) ensures that each ψ_i is a postcondition of the corresponding p_i w.r.t. the NN controller (note that ψ_i is not necessarily the strongest postcondition).

Now, from (12) and (13), Condition (8) becomes:

$$((p_1 \vee p_2 \vee \cdots \vee p_n) \wedge \mathit{Next}_{\mathit{NNC}}) \implies ((p_1 \wedge \psi_1) \vee (p_2 \wedge \psi_2) \vee \cdots \vee (p_n \wedge \psi_n))$$

which is equivalent to

$$\left(\bigvee_{i=1}^{n} (p_i \wedge \mathit{Next}_{\mathit{NNC}})\right) \implies \left(\bigvee_{j=1}^{n} (p_j \wedge \psi_j)\right) \tag{15}$$

which is equivalent to

$$\bigwedge_{i=1}^{n} \left((p_i \wedge \mathit{Next}_{\mathit{NNC}}) \implies (\bigvee_{j=1}^{n} (p_j \wedge \psi_j)) \right) \tag{16}$$

But observe that $(p_i \wedge \mathit{Next}_{\mathit{NNC}}) \implies p_i$ trivially holds, for each $i = 1, ..., n$. And note that $(p_i \wedge \mathit{Next}_{\mathit{NNC}}) \implies \psi_i$ also holds, since by construction, $\psi_i \in \mathrm{AllPosts}(\mathit{Next}_{\mathit{NNC}}, p_i)$. Therefore, $(p_i \wedge \mathit{Next}_{\mathit{NNC}}) \implies (p_i \wedge \psi_i)$ holds for each i, which implies that (16) holds by construction. Therefore, we have:

Theorem 3 (Condition (8) holds by construction). *For any bridge predicate that is in form of Condition (12), if this predicate satisfies Conditions (13) and (14), then Condition (8) holds by construction.*

Next, consider Condition (9). From (12), Condition (9) becomes:

$$\Big(\big((p_1 \wedge \psi_1) \vee (p_2 \wedge \psi_2) \vee \cdots \vee (p_n \wedge \psi_n)\big) \wedge \mathit{Next}_{ENV}\Big) \implies \mathit{IndInv}' \tag{17}$$

which is equivalent to:

$$\left(\bigvee_{i=1}^{n} (p_i \wedge \psi_i \wedge \mathit{Next}_{ENV})\right) \implies \mathit{IndInv}' \tag{18}$$

which is equivalent to:

$$\bigwedge_{i=1}^{n} \big((p_i \wedge \psi_i \wedge \mathit{Next}_{ENV}) \implies \mathit{IndInv}'\big) \tag{19}$$

which means that checking Condition (9) can be replaced by checking n smaller conditions, namely, $(p_i \wedge \psi_i \wedge \mathit{Next}_{ENV}) \implies \mathit{IndInv}'$, for $i = 1, ..., n$.

The algorithm that we present below (Algorithm 1) starts with $n = 1$ and $p_1 = \mathit{IndInv}$. It then computes some $\psi_1 \in \text{AllPosts}(\mathit{Next}_{NNC}, p_1)$ and checks whether $(p_1 \wedge \psi_1 \wedge \mathit{Next}_{ENV}) \implies \mathit{IndInv}'$ is valid. If it is, then $p_1 \wedge \psi_1$ is a valid bridge and inductiveness holds. Otherwise, p_1 is *split* into several p_i's, and the process repeats.

3.3 Heuristic for Falsifying Inductiveness

An additional feature of our algorithm is that it is often capable to *falsify* inductiveness and thereby prove that it does not hold. This allows us to avoid searching hopelessly for a bridge predicate when none exists, because the candidate invariant is not inductive.

Directly falsifying Condition (7) faces the same scalability issues as trying to prove this condition monolithically. On the other hand, falsifying Conditions (8) or (9) is not sufficient to disprove inductiveness. The failure to prove these conditions might simply mean that our chosen bridge predicate does not work.

Therefore, we propose a practical heuristic which inherits the decomposition ideas described above. This heuristic involves constructing: (i) a satisfiable *falsifying state predicate* over $\boldsymbol{s}$, denoted *FState*; and (ii) a predicate over the output of the NN controller $\boldsymbol{a}$, denoted *FPred*, such that the following conditions hold:

$$\mathit{FState} \implies \mathit{IndInv} \tag{20}$$

$$(\mathit{FState} \wedge \mathit{Next}_{NNC}) \implies \mathit{FPred} \tag{21}$$

$$(\mathit{FState} \wedge \mathit{FPred} \wedge \mathit{Next}_{ENV}) \implies \neg\mathit{IndInv}' \tag{22}$$

Intuitively, FState identifies a set of states which satisfy IndInv but violate IndInv' after a transition. FPred captures the outputs of the NN controller when its inputs satisfy FState. The conjunction of (21) and (22) implies:

$$(\mathit{FState} \land \mathit{Next}_{NNC} \land \mathit{Next}_{ENV}) \implies \neg \mathit{IndInv}' \tag{23}$$

In turn, (23) and (20) together imply that Condition (7) does not hold. This leads us to the following theorem:

Theorem 4 (Falsifying Inductiveness). *If FState is satisfiable, and if Conditions (20), (21), and (22) hold, then Condition (7) does not hold.*

Our falsification approach aligns well with the composite structure of a NNCS, and is compositional, because it allows to check separately the NN transition relation Next_{NNC} in (21) and the environment transition relation Next_{ENV} in (22).

3.4 Algorithm

The aforementioned ideas are combined in Algorithm 1, which implements our compositional inductiveness verification approach for NNCSs. Line 7 ensures that ψ is a postcondition of p w.r.t. the NN controller. In practice we use a *NN verifier* to compute the postcondition (see Section 4). Lines 15 and 16 ensure Condition (13), together with the fact that the initial p is IndInv (Line 3). Line 9 ensures that the bridge predicate is in form of Condition (12). Therefore, by Theorem 3, Algorithm 1 ensures Condition (8) by construction.

Line 8 corresponds to checking (19) compositionally, i.e., separately for each i. In practice, we use an *SMT solver* for this check (see Section 4).

For falsification, to use Theorem 4, we set $\mathit{FState} = p$ and $\mathit{FPred} = \psi$. The splitting process at Line 15 guarantees that p is satisfiable. Condition (20) then becomes $p \implies \mathit{IndInv}$, which holds by construction because of Condition (13). Condition (21) becomes $p \land \mathit{Next}_{NNC} \implies \psi$, which holds because $\psi \in \mathrm{AllPosts}(\mathit{Next}_{NNC}, p)$. Condition (22) becomes $(p \land \psi \land \mathit{Next}_{ENV}) \implies \neg \mathit{IndInv}'$, which is checked in Line 11.

If the algorithm can neither prove $(p \land \psi \land \mathit{Next}_{ENV}) \implies \mathit{IndInv}'$, nor falsify inductiveness, then it splits p into a disjunction of satisfiable state predicates, ensuring Condition (12). Our implementation utilizes various splitting strategies. The specific splitting strategies employed in our case studies are elaborated in Section 4. After splitting, all resulting state predicates are added into the queue Q. Consequently, the queue becomes empty if and only if the validity of every conjunct in Condition (19) is proved. Therefore, an empty queue indicates that inductiveness holds.

Termination: the algorithm terminates either upon successfully proving or upon falsifying inductiveness. However, termination is not guaranteed in all cases. In infinite state spaces, the algorithm may keep splitting predicates ad infinitum. In practice, as observed in our case studies, the algorithm consistently terminated (see Section 4).

Algorithm 1: Compositional Inductiveness Checking for NNCS

```
Input: Transition relations $Next_{NNC}$, $Next_{ENV}$; Candidate Inductive Invariant $IndInv$
Output: Verification result: (True with bridge predicate $Bridge$) or (False with falsifying state predicate)
Function CheckInductiveness($Next_{NNC}$, $Next_{ENV}$, $IndInv$):
    $Bridge := False$ ;
    $Q := \{IndInv\}$ ;
    while $Q \neq \emptyset$ do
        Let $p \in Q$ ;
        $Q := Q \setminus \{p\}$ ;
        Let $\psi \in \text{AllPosts}(Next_{NNC}, p)$ ;
        if $(p \wedge \psi \wedge Next_{ENV}) \implies IndInv'$ holds then
            $Bridge := Bridge \vee (p \wedge \psi)$ ;
        end
        else if $(p \wedge \psi \wedge Next_{ENV}) \implies \neg IndInv'$ holds then
            return (False, p) ;
        end
        else
            Split $p$ into $p_1, p_2, \cdots, p_k$ such that $p \iff (p_1 \vee p_2 \vee \cdots \vee p_k)$;
            $Q := Q \cup \{p_1, p_2, \cdots, p_k\}$ ;
        end
    end
    return (True, Bridge) ;
```

4 Evaluation

4.1 Implementation and Experimental Setup

We implemented Algorithm 1 in a prototype tool – the source code and models needed to reproduce our experiments are available at https://github.com/YUH-Z/comp-indinv-verification-nncs. We use the SMT solver Z3 [5] for the validity checks of Lines 8 and 11 of Algorithm 1. To compute the postcondition ψ (Line 7 of Algorithm 1), we use AutoLIRPA [29], which is the core engine of the NN verifier α-β-CROWN [28]. AutoLIRPA computes a postcondition $\psi \in \text{AllPosts}(Next_{NNC}, p)$, i.e., guarantees that $(p \wedge Next_{NNC}) \implies \psi$, but it does not generally guarantee that ψ is the strongest possible postcondition.

In the experiments reported below, we evaluate our compositional method by comparing it against the monolithic method which uses Z3 to check Condition 7 directly. We remark that we also attempted to use NN verifiers to check inductiveness monolithically, but this proved infeasible. Specifically, we tried two ways of encoding checking inductiveness as an NN verification problem: (i) Encoding $Next_{ENV}$ and Condition (7) into the NN's input and output constraints. This approach was impractical for our case studies since $Next_{ENV}$ involves arithmetic applied simultaneously to both the input and output of the NN controller, e.g., $(x' = x + 0.1a)$. Such constraints are beyond the capability of existing NN verifiers such as [2,16,17,28]. (ii) Encoding both the NN controller and the environment into a single NN. Our trials revealed that current NN verifiers support only a limited range of operators for defining a NN, restricting this approach as well. Notably, the operators we required, such as adding the NN's input to its output, are not typically supported. Although these limitations might diminish as NN verifiers evolve, an additional complication arises from the fact that $Next_{ENV}$ might be non-deterministic (as in our second set of case studies). Encoding non-deterministic transition relations into a NN remains a significant challenge because NNs are typically deterministic functions.

In our case studies reported below, each candidate inductive invariant is a union of hyperrectangles, which aligns with the constraint types supported by mainstream NN verifiers such as [2,16,17,28].

Our splitting strategy (Line 15 of Algorithm 1) follows a binary scheme, dividing hyperrectangles at their midpoints along each dimension. For example, the interval $[1, 2]$ would be split into $[1, 3/2]$ and $[3/2, 2]$. This straightforward strategy turns out to be effective in our case studies.

Our experiments were conducted on a machine equipped with a 3.3 GHz 8-core AMD CPU, 16 GB memory, and an Nvidia 3060 GPU. Z3 ran on the CPU and AutoLIRPA on the GPU with CUDA enabled, both using default configurations.

4.2 Case Studies and Experimental Results

We ran our experiments on two sets of examples, a *deterministic 2D maze*, and a *non-deterministic 2D maze*, as described below.

Deterministic 2D maze: In this example, the environment has two state variables x and y of type real ($\mathbb{R}$), representing an object's 2D position. The NN controller $f_{NN} : \mathbb{R}^2 \to \mathbb{R}^2$ outputs $(a, b) = f_{NN}(x, y)$, guiding the object's horizontal and vertical movement. The initial state is set within $0.3 \leq x \leq 0.4$ and $0.6 \leq y \leq 0.7$. The transition relation $Next_{ENV}$ of the environment is deterministic and defined by $x' = x + 0.1a$ and $y' = y + 0.1b$. The controller's goal is to navigate the object to ($0.8 \leq x \leq 0.9$ and $0.8 \leq y \leq 0.9$), while keeping it within a safe region of ($0.22 \leq x \leq 0.98$ and $0.54 \leq y \leq 0.98$), which is the safety property we want to prove.

As NN controllers, we used two-layer feed-forward NNs with ReLU activations, trained via PyTorch [20] and Stable-Baselines3 [21]. We trained NN controllers of various sizes, ranging from 2×32 to 2×1024 neurons. To standardize the comparison among the systems containing different NN controllers, for each system, we check the same candidate inductive invariant, defined as $0.25 \leq x \leq 0.95$ and $0.55 \leq y \leq 0.95$. For this candidate and for the *Init* and *Safe* predicates defined above, it is easy to see that Conditions (4) and (6) hold. So our experiments only check the inductiveness Condition (7).

The results are reported in Table 1: for this case study, the relevant columns are those marked as *Det*. The compositional method successfully terminated in all cases, and proved inductiveness for all NN configurations, except for 2×128, for which inductiveness does not hold, and in which case the compositional

Table 1: Experimental results: *Det* and *NDet* indicate the results for the deterministic and non-deterministic 2D maze case studies, respectively. All execution times are in seconds. T.O. represents an one-hour timeout. The *Verified?* column shows whether the compositional method successfully terminated, either by proving (T) or by disproving (F) inductiveness. *#Splits* reports the total number of splits performed. *#SMT* and *#NNV queries* report the total number of calls made to Z3 and AutoLIRPA, respectively.

NN size	Verified?		Monolithic execution time		Compositional execution time		#Splits		#SMT queries		#NNV queries	
	Det	NDet	Det	NDet	Det	NDet	Det	NDet	Det	NDet	Det	NDet
2×32	T	T	51.59	39.73	0.73	0.83	12	14	61	71	49	57
2×40	T	T	113.69	296.61	0.70	0.66	13	13	66	66	53	53
2×48	T	T	410.14	3002.20	0.52	0.42	10	8	51	41	41	33
2×56	T	T	1203.76	T.O.	0.42	0.46	8	9	41	46	33	37
2×64	T	T	T.O.	T.O.	0.76	0.61	15	12	76	61	61	49
2×128	F	T	T.O.	T.O.	2.21	1.43	64	28	225	141	160	113
2×256	T	T	T.O.	T.O.	1.68	1.18	27	23	136	116	109	93
2×512	T	T	T.O.	T.O.	3.04	2.30	60	45	301	226	241	181
2×1024	T	T	T.O.	T.O.	1.94	5.23	38	102	191	511	153	409

method managed to falsify it. In terms of performance, the monolithic method requires around one minute even for the smallest NN configurations, and times out after one hour for the larger configurations; while the compositional method terminates in all cases in a couple of seconds. A key metric of the compositional method is the number of splits, which is determined by the number of times Line 15 of Algorithm 1 is executed. This metric is crucial as it impacts the number of queries made to the SMT solver and to the NN verifier, typically the most time-consuming steps in the algorithm. Additionally, the number of splits indicates the size of the bridge predicate, where fewer splits suggest smaller bridge predicates.

Non-deterministic 2D maze: This case study is similar to the deterministic 2D maze, with the difference that the environment's transition relation is non-deterministic, specifically, defined as $x' = x + 0.1c \cdot a$ and $y' = y + 0.1c \cdot b$, where c is a constant of type real, non-deterministically ranging between 0.5 and 1.0. This non-determinism simulates the noise within the system.

The NN controllers in this set of experiments, while maintaining the same architecture and size as in the deterministic case studies, were retrained from scratch to adapt to the new transition relation. The candidate inductive invariant is the same as in the deterministic case study, ensuring consistency in our comparative analysis.

The results are shown in Table 1, columns marked by *NDet*. As demonstrated by the results, our compositional method successfully verified inductiveness in all configurations, in a matter of seconds.

5 Related Work

A large body of research exists on NN verification, including methods that verify NN input-output relations, using SMT solvers [13,16,17] or MILP solvers [25], as well as methods that employ abstract-interpretation techniques [9, 24], symbolic interval propagation [27], dual optimization [7], linear relaxation [29], and bound propagation [28].

These and other techniques and the corresponding tools focus on NN verification at the *component level*, that is, they verify a NN in isolation. In contrast, we focus on *system-level* verification, that is, verification of a closed-loop system consisting of a NN controller and an environment.

System-level verification approaches have also been proposed in the literature. A number of methods capture NNCSs as *hybrid systems*: in [15] the NN is transformed into a hybrid system and the existing tool Flow* [3] is used to verify the resulting hybrid system. In [12] the NN is approximated using Bernstein polynomials, while [14] employs Taylor models, and uses various techniques to reduce error. In [8] NNCSs are modeled as transition systems, and existing NN verifiers like Marabou [17] are used for bounded model checking. These methods are adept at proving safety properties within a bounded time horizon. In contrast, our approach is designed to verify safety properties over an infinite time horizon. In addition to verification, various methods for testing NNCSs have been explored. Simulation-based approaches for NNCS analysis, including falsification, fuzz testing, and counterexample analysis, were introduced by [6, 26]. [11] proposed a method to explore the state space of hybrid systems containing neural networks.

Our work is also related to research on automatic inductive invariant discovery, which is a hard, generally undecidable, problem [19]. Recently, several studies have proposed techniques for automatic inductive invariant discovery for distributed protocols [10, 23], while [22] applies deep learning techniques to infer loop invariants for programs. [1] proposes a heuristic for inferring simple inductive invariants in NNCSs, within the context of communication networking systems. Such systems possess unique traits not commonly found in general NNCSs. These specific traits enable using a NN verifier to check a weaker condition instead of directly verifying the inductiveness condition. In contrast, our method does not depend on specific properties of specialized NNCSs.

6 Conclusions

We present a compositional, inductive-invariant based, method for verifying safety in NNCSs. The key idea is to decompose the monolithic inductiveness check (which is typically not supported by state-of-the-art NN verifiers, and does not scale with state-of-the-art SMT solvers) into several manageable subproblems, which can be each individually handled by the corresponding tool. Our case studies show encouraging results where the verification time is reduced from hours (or timeout) to seconds.

Future work includes augmenting our method's capabilities to suit a broader range of NNCS applications. We also plan to explore the automatic generation of candidate inductive invariants (in addition to bridges) for NNCSs.

References

1. Amir, G., Schapira, M., Katz, G.: Towards scalable verification of deep reinforcement learning. In: Formal Methods in Computer Aided Design (FMCAD). pp. 193–203 (2021)
2. Bak, S.: nnenum: Verification of relu neural networks with optimized abstraction refinement. In: NASA Formal Methods Symposium. pp. 19–36 (2021)
3. Chen, X., Ábrahám, E., Sankaranarayanan, S.: Flow*: An analyzer for non-linear hybrid systems. In: Computer Aided Verification. pp. 258–263 (2013)
4. Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.): Handbook of Model Checking. Springer (2018)
5. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: International conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340 (2008)
6. Dreossi, T., Fremont, D.J., Ghosh, S., Kim, E., Ravanbakhsh, H., Vazquez-Chanlatte, M., Seshia, S.A.: VerifAI: A toolkit for the formal design and analysis of artificial intelligence-based systems. In: Computer Aided Verification (CAV) (Jul 2019)
7. Dvijotham, K., Stanforth, R., Gowal, S., Mann, T.A., Kohli, P.: A dual approach to scalable verification of deep networks. In: UAI. vol. 1, p. 3 (2018)
8. Eliyahu, T., Kazak, Y., Katz, G., Schapira, M.: Verifying learning-augmented systems. In: Proceedings of the 2021 ACM SIGCOMM 2021 Conference. pp. 305–318 (2021)
9. Gehr, T., Mirman, M., Drachsler-Cohen, D., Tsankov, P., Chaudhuri, S., Vechev, M.: Ai2: Safety and robustness certification of neural networks with abstract interpretation. In: 2018 IEEE symposium on security and privacy (SP). pp. 3–18 (2018)
10. Goel, A., Sakallah, K.: On Symmetry and Quantification: A New Approach to Verify Distributed Protocols. In: NASA Formal Methods Symposium. pp. 131–150 (2021)
11. Goyal, M., Duggirala, P.S.: Neuralexplorer: state space exploration of closed loop control systems using neural networks. In: Automated Technology for Verification and Analysis: 18th International Symposium, ATVA 2020, Hanoi, Vietnam, October 19–23, 2020, Proceedings 18. pp. 75–91. Springer (2020)
12. Huang, C., Fan, J., Li, W., Chen, X., Zhu, Q.: Reachnn: Reachability analysis of neural-network controlled systems. ACM Transactions on Embedded Computing Systems (TECS) **18**(5s), 1–22 (2019)
13. Huang, X., Kwiatkowska, M., Wang, S., Wu, M.: Safety Verification of Deep Neural Networks. In: Computer Aided Verification, vol. 10426, pp. 3–29. Springer (2017)
14. Ivanov, R., Carpenter, T., Weimer, J., Alur, R., Pappas, G., Lee, I.: Verisig 2.0: Verification of neural network controllers using taylor model preconditioning. In: Computer Aided Verification. pp. 249–262 (2021)
15. Ivanov, R., Weimer, J., Alur, R., Pappas, G.J., Lee, I.: Verisig: verifying safety properties of hybrid systems with neural network controllers. In: Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control. pp. 169–178 (2019)

16. Katz, G., Barrett, C., Dill, D.L., Julian, K., Kochenderfer, M.J.: Reluplex: An efficient smt solver for verifying deep neural networks. In: Computer Aided Verification. pp. 97–117 (2017)
17. Katz, G., Huang, D.A., Ibeling, D., Julian, K., Lazarus, C., Lim, R., Shah, P., Thakoor, S., Wu, H., Zeljić, A., et al.: The marabou framework for verification and analysis of deep neural networks. In: Computer Aided Verification: 31st International Conference, CAV 2019. pp. 443–452 (2019)
18. Manna, Z., Pnueli, A.: Temporal Verification of Reactive Systems: Safety. Springer-Verlag, New York (1995)
19. Padon, O., Immerman, N., Shoham, S., Karbyshev, A., Sagiv, M.: Decidability of inferring inductive invariants. ACM SIGPLAN Notices **51**(1), 217–231 (2016)
20. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al.: Pytorch: An imperative style, high-performance deep learning library. Advances in neural information processing systems **32** (2019)
21. Raffin, A., Hill, A., Gleave, A., Kanervisto, A., Ernestus, M., Dormann, N.: Stable-baselines3: Reliable reinforcement learning implementations. Journal of Machine Learning Research **22**(268), 1–8 (2021)
22. Ryan, G., Wong, J., Yao, J., Gu, R., Jana, S.: Cln2inv: learning loop invariants with continuous logic networks. arXiv preprint arXiv:1909.11542 (2019)
23. Schultz, W., Dardik, I., Tripakis, S.: Plain and simple inductive invariant inference for distributed protocols in tla+. In: Formal Methods in Computer-Aided Design (FMCAD). pp. 273–283 (2022)
24. Singh, G., Gehr, T., Mirman, M., Püschel, M., Vechev, M.: Fast and effective robustness certification. Advances in neural information processing systems **31** (2018)
25. Tjeng, V., Xiao, K.Y., Tedrake, R.: Evaluating robustness of neural networks with mixed integer programming. In: ICLR (2019)
26. Viswanadha, K., Kim, E., Indaheng, F., Fremont, D.J., Seshia, S.A.: Parallel and multi-objective falsification with scenic and verifai. In: Runtime Verification: 21st International Conference, RV 2021, Virtual Event, October 11–14, 2021, Proceedings 21. pp. 265–276 (2021)
27. Wang, S., Pei, K., Whitehouse, J., Yang, J., Jana, S.: Formal security analysis of neural networks using symbolic intervals. In: 27th USENIX Security Symposium (USENIX Security 18). pp. 1599–1614. USENIX Association, Baltimore, MD (Aug 2018)
28. Wang, S., Zhang, H., Xu, K., Lin, X., Jana, S., Hsieh, C.J., Kolter, J.Z.: Beta-crown: Efficient bound propagation with per-neuron split constraints for neural network robustness verification. Advances in Neural Information Processing Systems **34**, 29909–29921 (2021)
29. Xu, K., Shi, Z., Zhang, H., Wang, Y., Chang, K.W., Huang, M., Kailkhura, B., Lin, X., Hsieh, C.J.: Automatic perturbation analysis for scalable certified robustness and beyond. Advances in Neural Information Processing Systems **33** (2020)
30. Zhang, J., Li, J.: Testing and verification of neural-network-based safety-critical control software: A systematic literature review. Information and Software Technology **123**, 106296 (2020)