Noname manuscript No.
(will be inserted by the editor)

NeVer: A Tool for Artificial Neural Networks Verification

Luca Pulina and Armando Tacchella

the date of receipt and acceptance should be inserted later

Abstract The adoption of Artificial Neural Networks (ANNs) in safety-related applications is often avoided because it is difficult to rule out possible misbehaviors with traditional analytical or probabilistic techniques. In this paper we present NeVer, our tool for checking safety of ANNs. NeVer encodes the problem of verifying safety of ANNs into the problem of satisfying corresponding Boolean combinations of linear arithmetic constraints. We describe the main verification algorithm and the structure of NeVer. We present also empirical results confirming the effectiveness of NeVer on realistic case studies.

1 Introduction

Artificial Neural Networks (ANNs) find application in a wide range of research and engineering domains – see, e.g., [1] for a survey. In the case of *safety-related* equipment, i.e., "hardwired or programmable systems where a failure, singly or in combination with other failures/errors, could lead to death, injury or environmental damage", ANNs are confined to systems which must comply only to the lowest safety integrity levels (SILs) [2], achievable with standard industrial best practices [2]. With a few exceptions like, e.g., [3], using ANNs for safety-related equipment is mostly avoided due to the absence of safety assurance methods. In particular, while traditional analytical and probabilistic methods can be extended to ANNs, they can be ineffective in ensuring that outputs coming from ANNs do not generate a potential hazard in safety-critical applications [4]. On the other hand, the usage of – strongly advocated – formal verification techniques has never been investigated in the case of ANNs.

Formal verification is a promising path if we consider that there are several success stories wherein techniques like, e.g., Model Checking [5,6], are routinely used to insure correctness of hardware and software in industrial contexts. Examples include debugging of integrated circuits [7], device drivers [8], security protocols [9], embedded systems [10] and many other artifacts whose correctness should be relied upon. These experiences suggest that, albeit computational tractability is not ensured in most cases,

DIST, Università di Genova, Viale Causa, 13 – 16145 Genova, Italy
`Luca.Pulina@unige.it` – `Armando.Tacchella@unige.it`

and thus scalability remains an important issue, we are currently able to automatically verify small-to-medium scale pieces of hardware and software. The advantage of formal methods is their complementarity with respect to traditional verification techniques – e.g., testing – since they are usually effective in spotting “corner bugs” which are known to be very hard to find by resorting to empirical techniques only [11].

Starting from the lessons learned in applied formal verification contexts, in this paper we present NeVer (“Ne”ural networks “Ver”ifier), a tool leveraging formal methods to verify ANNs. NeVer is designed to deal with a particular class of ANNs, namely the multilayer perceptron (MLP). We chose this kind of network because, albeit of its relatively simple topology, MLPs can in principle approximate every non-linear real-valued function of n real-valued inputs, as stated by the Universal Approximation Theorem for ANNs [12]. The core of NeVer is composed by an abstraction-refinement mechanism to verify MLPs proposed, for the first time, in [13]. The intuition behind this mechanism is that the abstract MLP provides a sound overapproximation of the concrete one, i.e. if NeVer states that the abstract MLP is safe, the same holds true for the concrete MLP. If this is not the case, the counterexample found is tested using the concrete MLP. If the solution is spurious, NeVer computes a refinement of the current abstraction. A further peculiarity of the approach is that NeVer uses counterexamples to try to “repair” the behaviour of the concrete MLP, i.e., to eliminate the causes of its misbehavior. This can be viewed as a way to synthesize a correct program given its specification, and it is close in spirit to what has been proposed, e.g., in [14, 15] for concurrent program synthesis.

In NeVer, the MLP infrastructure and related services are provided by SHARK [16], a state of the art Machine Learning (ML) library. We chose SHARK because it offers both a wide range of efficient training algorithms for MLPs, and it supports several other ML algorithms. This gives us the opportunity to extend the capabilities of NeVer to other ML algorithms without extensive tool redesign. Concerning the verification part, NeVer can leverage any off-the-shelf solver integrating Boolean reasoning and linear arithmetic constraint solving. In the current implementation, we chose HySAT [17] because it features interval propagation, and it enables NeVer to deal with large encodings in reasonable time. In order to test the effectiveness of our approach, we experiment with NeVer on two realistic case studies, one about learning the forward kinematics of an industrial manipulator, and another about modeling concrete mix [18]. Using tunable problems regarding polynomial extrapolation, we also run tests concerning the scalability of NeVer, i.e., its ability to verify MLPs with a growing number of neurons. Our results show that NeVer can handle realistic sized MLPs, as well as support the MLP designer in improving on safety in a completely automated way.

The paper is structured as follows. In section 2 we introduce some basics about MLPs and reasoning on arithmetic constraints. In Section 3 we present the main elements of our system NeVer, including a brief theoretical outline of abstraction-refinement for MLPs. In particular, we present the main algorithm, and the architecture of NeVer. In Section 4 we present the result of our experimental analyses considering three different problems: (*i*) An artificial problem about polynomial extrapolation to test the scalability of NeVer; (*ii*) the prediction of a manipulator forward kinematics; and (*iii*) the prediction of compressive strength in high-performance concrete. In Section 5 we compare our work with related literature, and we conclude the paper in Section 6 with some final remarks and our proposal for a future research agenda. Finally, Appendix A contains the BNF grammar of NeVer input format, while Ap-

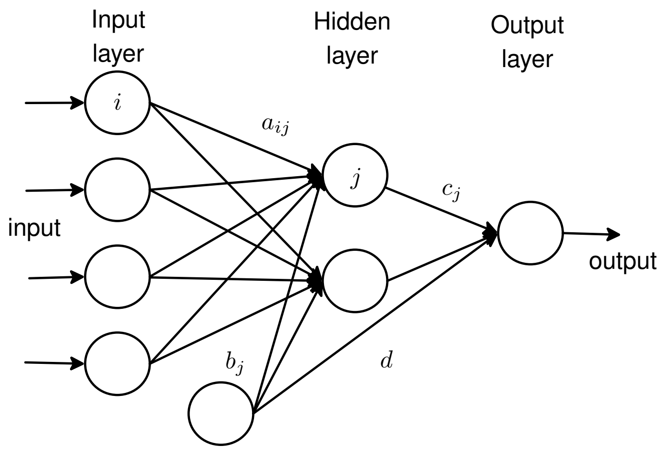

Fig. 1 Our MLP architecture of choice; neurons and connections are represented by circles and arrows, respectively.

pendix B contains an example encoding an abstract network and the related safety check problem into the language of HySAT.

2 Preliminaries

2.1 Multi-layer perceptrons

Multi-Layer Perceptrons [19] are probably the most widely studied and adopted type of ANN. An MLP is composed of a system of interconnected computing units (neurons), which are organized in layers. Figure 1 shows our MLP architecture of choice, consisting of three layers: An *input layer*, that serves to pass the input vector to the network; a *hidden layer* of computation neurons; and an *output layer* composed of at least one computation neuron. The MLPs that we consider are *fully connected*, i.e., each neuron is connected to every neuron in the previous and next layer. To see how an MLP processes information, let us consider the network ν in Figure 1. Having n neurons in the input layer ($n = 4$ in Figure 1), the i-th input value is denoted by x_i, $i = \{1, \ldots, n\}$. With m neurons in the hidden layer ($m = 2$ in Figure 1), the total input r_j received by neuron j, with $j = \{1, \ldots, m\}$, is called *induced local field* (ILF) and it is defined as

$$r_j = \sum_{i=1}^{n} a_{ji} x_i + b_j \tag{1}$$

where a_{ji} is the *weight* of the connection from the i-th neuron in the input layer to the j-th neuron in the hidden layer, and the constant b_j is the *bias* of the j-th neuron. The output of a neuron j in the hidden layer is a monotonic non-linear function of its ILF, the *activation function.* As long as such activation function is differentiable everywhere, MLPs with *only one* hidden layer can, in principle, approximate any real-valued function with n real-valued inputs [12]. A commonly used activation function [19] is the *logistic function*

$$\sigma(r) = \frac{1}{1+\exp(-r)},\ r \in \mathbb{R} \tag{2}$$

Therefore, the output of the MLP is

$$\nu(\underline{x}) = \sum_{j=1}^{m} c_j \sigma(r_j) + d \tag{3}$$

where c_j denotes the weight of the connection from the j-th neuron in the hidden layer to the output neuron, while d represents the bias of the output neuron. Equation (3) implies that the identity function is used as activation function of input- and output-layer neurons. This is a common choice when MLPs deal with *regression problems*. In regression problems, we are given a set of *patterns*, i.e., input vectors $X = \{\underline{x}_1, \ldots, \underline{x}_k\}$ with $\underline{x}_i \in \mathbb{R}^n$, and a corresponding set of *labels*, i.e., output values $Y = \{y_1, \ldots, y_k\}$ with $y_i \in \mathbb{R}$. We think of the labels as generated by some unknown function $f : \mathbb{R}^n \to \mathbb{R}$ applied to the patterns, i.e., $f(\underline{x}_i) = y_i$ for $i \in \{1, \ldots, k\}$. The task of ν is to *extrapolate* f given X and Y, i.e., construct ν from X and Y so that when we are given some $\underline{x}^* \notin X$ we should ensure that $\nu(\underline{x}^*)$ is "as close as possible" to $f(\underline{x}^*)$. In the following, we briefly describe how this can be achieved in practice.

Given a set of patterns X and a corresponding set of labels Y generated by some unknown function f, the process of tuning the weights and the biases of an MLP ν in order to extrapolate f is called *training*, and the pair (X, Y) is called the *training set*. We can see training as a way of learning a concept, i.e., the function f, from the labelled patterns in the training set. In particular, we speak of *supervised learning* because labels can be used as a reference for training, i.e., whenever $\nu(\underline{x}_i) \neq y_i$ with $\underline{x}_i \in X$ and $y_i \in Y$ an *error signal* can be computed to determine how much the weights should be adjusted to improve the quality of the response of ν. A well-established training algorithm for MLPs is *back-propagation* (BP) [19]. Informally, an *epoch* of BP-based training is the combination of two steps. In the *forward step*, for all $i \in \{1, \ldots, k\}$, $\underline{x}_i \in X$ is input to ν, and some cumulative error measure ϵ is evaluated. In the *backward step*, the weights and the biases of the network are all adjusted in order to reduce ϵ. After a number of epochs, e.g., when ϵ stabilizes under a desired threshold, BP stops and returns the weights of the neurons, i.e., ν is the *inductive model* of f.

In general, extrapolation is an ill-posed problem. Even assuming that X and Y are sufficient to learn f, it is still the case that different sets X, Y will yield different settings of the MLP parameters. Again, we cannot choose elements of X and Y to guarantee that the resulting network ν will not *underfit* f, i.e., consistently deviate from f, or *overfit* f, i.e., be very close to f only when the input vector is in X. Both underfitting and overfitting lead to poor *generalization* performances, i.e., the network largely fails to predict $f(\underline{x}^*)$ on yet-to-be-seen inputs $\underline{x}^*$. Statistical techniques can provide reasonable estimates of the generalization error – see, e.g., [19]. In our experiments, we use *leave-one-out cross-validation* (or, simply, leave-one-out) which works as follows. Given the set of patterns X and the set of labels Y, we obtain the MLP $\nu_{(i)}$ by applying BP to the set of patterns $X_{(i)} = \{x_1, \ldots, x_{i-1}, x_{i+1}, \ldots x_k\}$ and to the corresponding set of labels $Y_{(i)}$. If we repeat the process k times, then we obtain k different MLPs so that we can estimate the generalization error as

$$\hat{\epsilon} = \sqrt{\frac{1}{k} \sum_{i=1}^{k} (y_i - \nu_{(i)}(\underline{x}_i))^2} \tag{4}$$

which is the root mean squared error (RMSE) among all the predictions made by each $\nu_{(i)}$ when tested on the unseen input $\underline{x}_i$. Both leave-one-out and RMSE are a common method of estimating and summarizing the generalization error in MLP applications (see e.g. [19]).

2.2 Reasoning on arithmetic constraints

A Constraint Satisfaction Problem (CSP) [20] is defined over a *constraint network*, i.e., a finite set of *variables*, each ranging over a given domain, and a finite set of *constraints*. A constraint represents a combination of values that a certain subset of variables is allowed to take. Formally, given an assignment of values to the variables, a constraint is a function that returns a Boolean value. A constraint is *satisfied* if it is true under a given assignment, and it is *violated* otherwise. A *solution* to a CSP is an assignment of values to the variables that satisfies all the constraints. In this paper we are concerned with CSPs involving arithmetic constraints over real-valued variables. In particular, the constraint sets that we consider are built according to the following grammar:

$$\begin{array}{rcl}
\textit{set} & \rightarrow & \{\ \textit{constraint}\ \}^*\ \textit{constraint} \\
\textit{constraint} & \rightarrow & (\{\ \textit{atom}\ \}^*\ \textit{atom}) \\
\textit{atom} & \rightarrow & \textit{expr}\ \textit{relop}\ \textit{expr} \\
\textit{relop} & \rightarrow & < \mid \leq \mid \geq \mid > \\
\textit{expr} & \rightarrow & \textit{expr}\ \textit{addop}\ \textit{expr} \\
 & & \mid \textit{factor} \\
\textit{factor} & \rightarrow & \textit{var} \mid \textit{const} \mid \textit{const} \cdot \textit{var} \\
\textit{addop} & \rightarrow & + \mid \text{-}
\end{array}$$

where *const* is a real-valued constant, and *var* is a real-valued variable. As we can see, for our purposes it is sufficient to have addition and subtraction between generic expressions, whereas multiplication happens between constants and variables only, i.e., we deal with *linear* arithmetic constraints. Let X be a set of real-valued variables, and μ be an assignment of values to the variables in X, i.e., a partial function $\mu : X \rightarrow \mathbb{R}$. Assuming that, for every μ, $\textit{relop}^{\mu}$, $-^{\mu}$, $+^{\mu}$ and $*^{\mu}$ denote the standard interpretations of relational and arithmetical operators over real numbers, we interpret a constraint network over X as follows:

- A constraint $(a_1 \ldots a_n)$ is true exactly when at least one of the atoms a_i with $i \in \{1, \ldots, n\}$ is true.
- Given a variable $x \in X$ and a constant $c \in \mathbb{R}$, an atom *expr relop expr* is true exactly when $\textit{expr}^{\mu}\ \textit{relop}^{\mu}\ \textit{expr}^{\mu}$ holds.
- Finally, $\textit{expr}^{\mu} \in \mathbb{R}$ is the extension of μ to well formed expressions, and it is defined recursively as follows
  - if $\textit{expr} = x$ with $x \in X$ then $\textit{expr}^{\mu} = \mu(x)$;
  - if $\textit{expr} = c$ with $c \in \mathbb{R}$ then $\textit{expr}^{\mu} = c$;
  - if $\textit{expr} = c * x$, then $\textit{expr}^{\mu} = c *^{\mu} x^{\mu}$;
  - if $\textit{expr} = \textit{expr}_1\ \textit{addop}\ \textit{expr}_2$ then a $\textit{expr}^{\mu} = \textit{expr}_1^{\mu}\ \textit{addop}^{\mu}\ \textit{expr}_2^{\mu}$;

There are various algorithmic approaches that can be used to solve CSPs defined over linear arithmetic constraints. The details of these approaches are beyond the scope of this paper (see, e.g., [21,22]). Here we just mention that HySAT, the constraint solver used in our prototype, uses a combination of interval constraint propagation (ICP) techniques – see, e.g., [23] – to manage feasibility of arithmetic constraints over real-valued variables, combined with modern Boolean solvers – see, e.g., [24] – to manage the underlying Boolean satisfiability problem. The idea is that a Boolean solver finds truth assignments to atoms such that all the constraints are satisfied. Since atoms correspond to arithmetic constraints, a truth assignment is equivalent to a feasibility problem which can be solved using a specialized procedure – ICP in the case of HySAT. This approach, known as "satisfiability modulo theory" (SMT) [25], is nowadays a well-established approach to deal with constraint satisfaction over various kinds of arithmetic constraints, and this topic is receiving increasing attention in recent years – see, e.g., [26].

3 NeVer: abstraction, algorithm and architecture

3.1 Abstraction

In [13] we proposed an abstraction of MLPs to corresponding Boolean combinations of linear constraints, following the abstract interpretation framework described in [27]. Abstraction is a key enabler for verification because MLPs are compositions of non-linear and transcendental real-valued functions, and the theories to handle such functions are undecidable [17]. In [13] we showed that an abstraction mechanism can be devised to yield consistent overapproximations of concrete networks, i.e., once the abstract MLP is proven to be safe, the same holds true for the concrete one. We now outline the abstraction mechanism whose details can be found in [13].

Given an MLP ν with n inputs and a single output, the *input domain* of ν is defined as a Cartesian product $\mathcal{I} = D_1 \times \ldots \times D_n$ where for all $1 \leq i \leq n$ the i-th element of the product $D_i = [a_i, b_i]$ is a closed interval bounded by $a_i, b_i \in \mathbb{R}$. Similarly, the *output domain* of ν is defined as a closed interval $\mathcal{O} = [a, b]$ bounded by $a, b \in \mathbb{R}$. Any MLP ν can be thus considered as a function $\nu : \mathcal{I} \to \mathcal{O}$, and ν is defined *safe* if it satisfies the property

$$\forall \underline{x} \in \mathcal{I} : \nu(\underline{x}) \in [l, h] \tag{5}$$

where $l, h \in \mathcal{O}$ are *safety thresholds*, i.e., constants defining an interval wherein the MLP output is to range, given all acceptable input values.

The purpose is the verification of a *consistent abstraction* of ν, i.e., a function $\tilde{\nu}$ such that if the property corresponding to (5) is satisfied by $\tilde{\nu}$ in a suitable abstract domain, then it must hold also for ν. As in any abstraction-based approach to verification, the key point is that verifying condition (5) in the abstract domain is feasible, possibly without using excessive computational resources. In [13], we built abstract interpretations of MLPs where the *concrete domain* $\mathbb{R}$ is the set of real numbers, and the corresponding *abstract domain* $[\mathbb{R}] = \{[a, b] \mid a, b \in \mathbb{R}\}$ is the set of (closed) intervals of real numbers. In the abstract domain there is the usual containment relation "$\sqsubseteq$" such that given two intervals $[a, b] \in [\mathbb{R}]$ and $[c, d] \in [\mathbb{R}]$, it holds that $[a, b] \sqsubseteq [c, d]$ exactly when $a \geq c$ and $b \leq d$, i.e., $[a, b]$ is a subinterval of – or it coincides with – $[c, d]$. Given any set $X \subseteq \mathbb{R}$, *abstraction* is defined as the mapping $\alpha : 2^{\mathbb{R}} \to [\mathbb{R}]$ such that

$$\alpha(X) = [\min\{X\}, \max\{X\}] \tag{6}$$

Conversely, given $[a, b] \in [\mathbb{R}]$, *concretization* is defined as the mapping $\gamma : [\mathbb{R}] \to 2^{\mathbb{R}}$ such that

$$\gamma([a, b]) = \{x \mid x \in [a, b]\} \tag{7}$$

which represents the set of all real numbers comprised in the interval $[a, b]$. Given the posets $\langle 2^{\mathbb{R}}, \subseteq\rangle$ and $\langle[\mathbb{R}], \sqsubseteq\rangle$, the pair $\langle\alpha, \gamma\rangle$ is a Galois connection.

If $\nu : \mathcal{I} \to \mathcal{O}$ is the *concrete* MLP for which safety in terms of (5) is to be proved, abstraction amounts to check whether the *abstract* MLP $\tilde{\nu}$ – defined as a function $\tilde{\nu} : [\mathcal{I}] \to [\mathcal{O}]$ – is safe, in the sense that

$$\{\nu(\underline{x}) \mid \underline{x} \in X\} \subseteq \gamma(\tilde{\nu}(\alpha(X))) \tag{8}$$

In words, when given the interval vector $\alpha(X)$ as input, $\tilde{\nu}$ outputs an interval which corresponds to a superset of the values that ν would output if given as input all the vectors in X. At the end, given the safety thresholds $l, h \in \mathcal{O}$, if we can prove

$$\forall[\underline{x}] \in [\mathcal{I}] : \tilde{\nu}([\underline{x}]) \sqsubseteq [l, h] \tag{9}$$

then, from (8) and the definition of γ, it immediately follows that

$$\{\nu(\underline{x}) \mid \underline{x} \in \mathcal{I}\} \subseteq \{y \mid l \leq y \leq h\} \tag{10}$$

which implies that condition (5) is satisfied by ν, because ν may not output a value outside the interval $[l, h]$ without breaking (10).

In order to obtain $\tilde{\nu}$, we assume that the activation function of the hidden-layer neurons is the logistic function (2), where $\sigma(x) : \mathbb{R} \to \mathcal{O}_\sigma$ and $\mathcal{O}_\sigma = (0, 1)$. Given an abstraction parameter $p \in (0, 1)$, the *abstract activation function* $\tilde{\sigma}_p : [\mathbb{R}] \to [\mathcal{O}_\sigma]$ can be obtained as the following interval extension of σ

$$\tilde{\sigma}_p([x_a, x_b]) = \begin{cases} [0, p/4] & \text{if } x_b \leq x_0 \\ [0, \sigma(p\lfloor \frac{x_b}{p} \rfloor) + \frac{p}{4}] & \text{if } x_a \leq x_0 \text{ and } x_b < x_1 \\ [\sigma(p\lfloor \frac{x_a}{p} \rfloor), \sigma(p\lfloor \frac{x_b}{p} \rfloor) + \frac{p}{4}] & \text{if } x_0 \leq x_a \text{ and } x_b \leq x_1 \\ [\sigma(p\lfloor \frac{x_a}{p} \rfloor), 1] & \text{if } x_0 < x_a \text{ and } x_1 \leq x_b \\ [1 - p/4, 1] & \text{if } x_a \geq x_1 \\ [0, 1] & \text{if } x_a \leq x_0 \text{ and } x_1 \leq x_b \end{cases} \tag{11}$$

where x_0 and x_1 are the values that satisfy $\sigma(x_0) = p/4$ and $\sigma(x_1) = 1 - p/4$, respectively. Notice that we have considered the maximum slope of the tangent to σ – $1/4$ in the origin – in order to overestimate the increment of σ over intervals of length p. The parameter p can then be used to control how much $\tilde{\sigma}_p$ over-approximates σ: Large values of p correspond to coarse-grained abstractions, whereas small values of p correspond to fine-grained ones. As shown in [13], since $\tilde{\sigma}_p$ is a consistent abstraction of σ, and products and sums on intervals are consistent abstractions of the corresponding operations on real numbers, the following definition of $\tilde{\nu}_p$ is a consistent abstraction of ν:

$$\tilde{\nu}_p([\underline{x}]) = \sum_{j=1}^{m} c_j \tilde{\sigma}_p(\tilde{y}_j([\underline{x}])) + d \tag{12}$$

where $\tilde{y}_j([\underline{x}]) = \sum_{i=1}^{n} a_{ji}[x_i] + b_j$ is the interval abstraction of the ILF[1]. Abstraction opens the path to spurious counterexamples, i.e., violations of the abstract safety property which fail to realize on ν. In these cases, since the "coarseness" is controlled by the abstraction parameter p, it is sufficient to modify it to refine the abstraction and then retry the verification. While our approach is clearly inspired by counterexample guided

[1] Notice that the usual symbols for products and sums are overloaded to denote products and sums across intervals.

```
NeVer($\Delta$, $\Pi$, $[l,h]$, $p$, $r$)
 1 isSafe ← FALSE; isFeasible ← FALSE
 2 $\nu$ ← TRAIN($\Delta$, $\Pi$)
 3 repeat
 4   $\tilde{\nu}_p$ ← ABSTRACT($\nu$, $p$)
 5   $\tilde{s}$ ← NIL; isSafe ← CHECKSAFETY($\tilde{\nu}_p$, $[l,h]$, $\tilde{s}$)
 6   if (not isSafe) then
 7     $o$ ← NIL; isFeasible ← CHECKFEASIBILITY($\nu$, $\tilde{s}$, $o$)
 8     if (not isFeasible) then
 9       $p \leftarrow p$ / $r$
10       $\Delta$ ← UPDATE($\Delta$, $\tilde{s}$, $o$)
11       $\nu$ ← TRAIN($\Delta$, $\Pi$)
12 until isSafe or (not isSafe and isFeasible)
13 return isSafe
```

Fig. 2 Pseudo-code of NeVer.

abstraction-refinement (CEGAR) [28], in our case refinement is not guided by the counterexample, but just caused by it, so in the following we speak of counterexample *triggered* abstraction-refinement (CETAR).

3.2 Core algorithm

Leveraging the definitions in the previous subsection, we provide a complete abstraction-refinement algorithm to prove MLP safety. The pseudo-code in Figure 2 is at the core of our tool NeVer. It takes as input a training set Δ, a set of MLP parameters Π, the safety thresholds $[l, h]$, the initial abstraction parameter p, and the refinement rate r. In line 1, two Boolean flags are defined, namely *isSafe* and *isFeasible*. The former is set to TRUE when verification of the abstract network succeeds, while the latter is set to TRUE when an abstract counterexample can be realized on the concrete MLP. In line 2, a call to the function TRAIN yields a concrete MLP ν from the set Δ. The set Π must supply parameters to control topology and training of the MLP, i.e., the number of neurons in the hidden layer and the number of BP epochs. The result ν is the MLP with the least cumulative error among all the networks obtained across the epochs [16]. Lines 4 to 13 are the core of the CETAR loop. Given p, the function ABSTRACT computes $\tilde{\nu}_p$ exactly as shown in (12) and related definitions. In line 5, CHECKSAFETY is devoted to interfacing with the HySAT solver in order to verify $\tilde{\nu}_p$. In particular, HySAT is supplied with a Boolean combination of linear arithmetic constraints modeling $\tilde{\nu}_p : [\mathcal{I}] \to [\mathcal{O}]$, and defining the domains $[\mathcal{I}]$ and $[\mathcal{O}]$, plus a further constraint encoding the safety condition – see Appendix B for details. In particular, this is about finding some interval $[\underline{x}] \in [\mathcal{I}]$ such that $\tilde{\nu}([\underline{x}]) \not\sqsubseteq [l, h]$. CHECKSAFETY takes as input also a variable $\tilde{s}$ that is used to store the abstract counterexample, if any. CHECKSAFETY returns one of the following results:

- If the set of constraints supplied to HySAT is unsatisfiable, then for all $[\underline{x}] \in [\mathcal{I}]$ we have $\tilde{\nu}_p([\underline{x}]) \sqsubseteq [l, h]$. In this case, the return value is TRUE, and $\tilde{s}$ is not set.
- If the set of constraints supplied to HySAT is satisfiable, this means that there exists an interval $[\underline{x}] \in [\mathcal{I}]$ such that $\tilde{\nu}([\underline{x}]) \not\sqsubseteq [l, h]$. In this case, such $[\underline{x}]$ is collected in $\tilde{s}$, and the return value is FALSE.

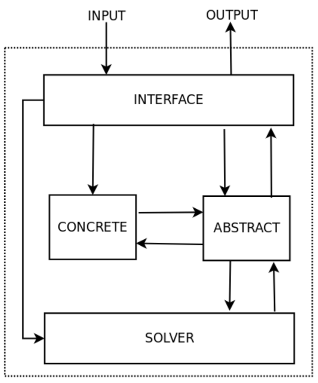

Fig. 3 NeVer architecture. The dotted box represents the whole system and, inside it, each solid box represent a module. Arrows denote functional connections between modules.

If *isSafe* is TRUE after the call to CHECKSAFETY, then the loop ends and NeVer exits successfully. Otherwise, the abstract counterexample $\tilde{s}$ must be checked to see whether it is spurious or not. This is the task of CHECKFEASIBILITY, which takes as input the concrete MLP ν, and a concrete counterexample extracted[2] from $\tilde{s}$. The parameter o is used to store the answer of ν when given $\tilde{s}$ as input. If the abstract counterexample can be realized ($\nu(\tilde{s}) \notin [l, h]$) then the loop ends and NeVer exits reporting an unsuccessful verification. Otherwise, we update the abstraction parameter p according to the refinement rate r – line 9. The next two lines are devoted to automate repairing. This code is meant to *update* the concrete MLP by

– adding the input pattern extracted from the spurious counterexample $\tilde{s}$ and the corresponding output o to the set Δ, and
– training a new network on the extended set.

Once updated and retrained the MLP, we restart the loop. These steps are meant to automatically correct the misbehavior of the input MLP. Intuitively, this is because spurious counterexamples obtained during CETAR loops correspond to input vectors which would violate the safety constraints if the concrete MLP were to respond with less precision than what is built in it. Since the abstract MLP consistently over-approximates the concrete one, a spurious counterexample is a weak spot of the abstract MLP which could be critical also for the concrete one. In Section 4 we show some empirical evidence that adding spurious counterexamples to the training set yields MLPs which are safer than the original ones.

3.3 Architecture

Figure 3 presents NeVer[3] architecture. NeVer is written in C++, which makes for an easier interfacing with the SHARK library (version 2.3.1), and opens the possibility

[2] We consider a vector whose components are the midpoints of the components of the interval vector emitted by HySAT as witness.
[3] NeVer is available for download at `http://www.mind-lab.it/never`.

to integrating efficient constraint solvers written in the same language via API calls. Looking at Figure 3, we can see that NeVer is composed by four modules:

`INTERFACE`: This module manages the input received by the user, and manages the output of NeVer to make it human-readable. It parses the input file, dispatching all the collected settings to the remaining NeVer modules, as denoted by the outgoing arrows. In particular, `INTERFACE` collects parameters concerning the training of the MLP, specific settings of HySAT, and parameters devoted to control the abstraction-refinement routine. `INTERFACE` sends these parameters to `CONCRETE`, `SOLVER`, and `ABSTRACT` modules, respectively.

`CONCRETE`: This module provides concrete MLP infrastructure, including representation and support for evaluation and repairing. The module is built on top of the shark library. In particular, in the current implementation NeVer uses the MLP predefined model `LinOutFFNet`. We chose this model because it is powerful enough to approximate non-linear functions of a general nature. It receives from `INTERFACE` the training set, and it implements routines related to concrete MLP training (using IRprop^{+}), testing, and updating.

`ABSTRACT`: This module is the core of NeVer. It interacts with every other module, and it enables potential extensions with different ML algorithms and constraint solvers without having to change the core. `ABSTRACT` takes as input the parameters of the trained concrete MLP coming from `CONCRETE`, together with the abstraction-refinement parameters from `INTERFACE`. `ABSTRACT` contains routines for the following computations:
- The abstraction of the MLP.
- The encoding of the abstract MLP into the HySAT input format, augmented with the safety target coming from `INTERFACE`.
- Interaction with the module `SOLVER`. The outgoing one is implemented by a system call to the HySAT solver, while the incoming one is implemented by a parser for the output of HySAT. We implemented this solution because HySAT is available to us only in the form of external executable program.
- A feasibility check, implemented querying the module `CONCRETE`.

`SOLVER`: In the current implementation, this module is composed by the executable of the HySAT solver (version 0.8.6).

4 Experimental analysis

In this Section we provide empirical results[4] about NeVer. We consider three different case studies. The purpose of the first is to asses the scalability of NeVer, whereas the other two show its performances in realistic applications. In the following, we briefly describe each case study in some detail and we give also pointers to related resources.

Polynomial extrapolation. For this case, we compute five synthetic training sets using the tool RapidMiner [29]. We call these sets Δ_i, where $i = \{4, 8, 16, 32, 64\}$ denotes the polynomial degree. Each training set (X, Y) is composed by 200 entries, and the patterns $\underline{x} \in X$ are real numbers in the range $[-1, 1]$, while for all the responses

[4] All the experimental results are obtained on a family of identical Linux workstations comprised of 10 Intel Core 2 Duo 2.13 GHz PCs with 4GB of RAM running Linux `Debian 2.6.18.5`

$y \in Y$ we have $y \in [-1, 2]$. For each Δ_i, a pattern is composed by i features (one per polynomial degree), and NeVer computes a concrete MLP having i neurons in the input layer, and $\frac{i}{2}$ neurons in the hidden layer. As condition to test, we check by using NeVer if there exist a $\tilde{\nu}_p([\underline{x}]) \in [-1, 2]$. Notice that this is not a safety check as defined in 5, which would be trivial with artificially generated polynomials. However, such a test still gives us meaningful results in terms of computation time versus MLP size.

PUMA manipulator. This case study is about learning the forward kinematics of a PUMA 500 manipulator. The PUMA 500 is a 6 degrees-of-freedom industrial manipulator with revolute joints, which has been widely used in industry and it is still common in academic research projects. The joints are actuated by DC servo motors with encoders to locate angular positions. Learning forward kinematics amounts to learning the mapping from joint angles to end-effector position. In our case, for the sake of simplicity, we confine ourselves to predict the end-effector position along a single coordinate of a Cartesian system having origin in the center of the robot's workspace. Since we learn the mapping using examples inside a region that we consider to be safe for the manipulator's motion, we expect the MLP to never emit a prediction that exceeds the safe region. A MLP failing to do so is to be considered unsafe. To train the MLP, we consider a training set (X, Y) collecting 141 entries. The patterns $\underline{x} \in X$ are vectors encoding the 6 joint angles, i.e., $\underline{x} = \langle \theta_1, \ldots, \theta_6 \rangle$ (in radians), and the labels are the corresponding end-effector coordinate (in meters). The range that we consider to be safe for motion goes from -0.35m to 0.35m, thus for all $y \in Y$ we must have $y \in [-0.35, 0.35]$. We have built the training set using the ROBOOP library [30] which provides facilities for simulating the PUMA 500 manipulator.

Concrete compressive strength. This case study is extracted from [18], and it is about the prediction of concrete compressive strength (CCS) given the proportion of various elements in the actual concrete mix. CCS is the staying power of concrete against pushing forces, and it is an important parameter in construction engineering because when maximum CCS is exceeded, concrete construction elements may get crushed. The issues related to this case study are two: On one hand, overestimation may imply an unjustified cost of the concrete mix, i.e., the task is money-critical; on the other hand, an underestimation may imply that the final construction lacks stability, i.e. the task is safety-critical. The prediction of CCS turns out to be a difficult task, as reported in [18]. Still in [18], authors show how such prediction can be done using MLPs in order to review the effects of each component on the concrete mix. In our experiments, we train the MLP using a subset of the "Concrete Compressive Strength dataset" – CCS dataset in the following – from the UCI repository[5], to which we refer for further explanations. Our task is to obtain CCS in the range $[30, 70]$ MPa, where 30 MPa is the standard lower bound for commercial structures, and 70 MPa is a motorway-tunnel standard. In order to do that, from the CCS dataset we select a subset of 596 entries – out of 1030 – whose label is in the range $[30, 70]$ MPa.

In the following, we first describe the results obtained on the two real world case studies *without* using automated repair, and then we consider adding repair in order to evaluate the improvements brought by this technique.

[5] Available at `http://archive.ics.uci.edu/ml/datasets/Concrete+Compressive+Strength`.

i	TIME			VARIABLES	CLAUSES
	TOTAL	MLP	HySAT		
4	0.55	0.53	0.00	390	1371
8	1.66	1.42	0.19	806	2904
16	6.69	4.48	2.11	1746	6260
32	344.59	15.03	329.27	4002	21626
64	TO	55.87	–	–	–

Table 1 Scalability test on NeVer. The table is organized as follows. The first column ("i") denotes the total amount of input neurons of the MLP, and it is followed by a group of three columns ("Time"): column "Total" reports the overall CPU seconds spent by NeVer, columns "MLP" and "HySAT" report the fraction of CPU seconds used to train the concrete MLP, and to verify the safety condition in the abstract MLP, respectively. The last two columns ("Variables" and "Clauses"), denote the total amount of variables and clauses composing the encoded abstract MLP, as reported in the HySAT log. TO denotes that NeVer exceeded the CPU time limit (48 hours), and a dash ("–") means that there are no available data.

4.1 Assessing scalability

Table 1 shows the results of the experiment about polynomial extrapolation. From the table, we can see that the time spent by NeVer grows together with the size of the network. In particular, we report a noticeable increase in CPU time when the number of inputs switches from 16 to 32. Moreover, while in the previous steps the total amount of clauses generated by HySAT doubled at each step, between 16 and 32, it grows of about a factor 4. We conclude reporting the performance of NeVer related to $i = 64$, where, even if the growth of MLP computation time is in line with the other results, NeVer was not able to conclude after 2 days of computation. In this case, as well as for $i = 32$, the bottleneck is the excessive complexity of the encoding to be handled by HySAT. It is important to notice, however, that in practical applications, it is not typical to find networks of this size (see, e.g. [31]).

4.2 Checking safety

Our experiments about safety checking aim to find a region $[l, h]$ that we can guarantee to be safe. We set the abstraction parameter to $p = 0.5$ and the refinement rate to $r = 1.1$. Since we experiment the verification algorithm without repair, we consider the procedure in Figure 2 excluding lines 10 and 11.

Concerning the PUMA dataset, the MLP has 3 neurons in the hidden layer, and it takes 0.64s for training across 500 epochs, yielding a MLP with an RMSE estimate of the generalization error $\hat{\epsilon} = 0.024$m (by using leave-one-out) – the error distribution ranges from a minimum of 3.2×10^{-5}m to a maximum of 0.123m, with a median value of 0.020m. In order to find l and h, we start by considering the interval $[-0.35, 0.35]$ – recall that this is the interval in which we consider motion to be safe. Whenever we find a counterexample stating that the network is unsafe in a given bound, we enlarge the bound. Once we have reached a safe configuration, we try to shrink the bound, until we reach the tightest bound that we can consider safe. The results of the above experiment are reported in Table 2 (top). In the table, we can see that NeVer is able to guarantee that the MLP is safe in the range $[-0.575, 0.575]$. If we try to shrink this bound, then NeVer is always able to find a set of inputs that makes the MLP

	l	h	Result	#	Time	
					Total	HySAT
PUMA	-0.350	0.350	UNSAFE	8	1.95	1.01
	-0.450	0.450	UNSAFE	9	3.15	2.10
	-0.550	0.550	UNSAFE	12	6.87	5.66
	-0.575	0.575	SAFE	11	6.16	4.99
	-0.600	0.600	SAFE	1	0.79	0.12
	-0.650	0.650	SAFE	1	0.80	0.13
CCS	30	70	UNSAFE	4	169598.40	169589.28
	25	75	UNSAFE	4	16.07	7.48
	20	80	–	7	TO	–
	15	85	–	7	TO	–
	10	90	–	3	TO	–
	5	95	–	10	TO	–

Table 2 Safety checking of the PUMA (top) and CCS (bottom) dataset with NeVer. Second and third columns ("l" and "h") report lower and upper safety thresholds, respectively. The fourth column reports the final result of NeVer, and column "#" indicates the number of abstraction-refinement loops. The two columns under "Time" report the total CPU time (in seconds) spent by NeVer and by HySAT, respectively. Finally, TO denotes that NeVer exceeded the CPU time limit (48 hours), and a dash ("–") means that there are no available data.

exceed the bound. Notice that the highest total amount of CPU time corresponds to the intervals $[-0.550, 0.550]$ and $[-0.575, 0.575]$, which are the largest unsafe one and the tightest safe one, respectively. Considering the number of abstraction-refinement loops, we can see that in both cases their number is larger than other configurations that we tried.

Concerning the CCS dataset, we train MLPs with 8 neurons in the hidden layer as in [18]. NeVer takes 8.15s for training across 500 epochs, yielding an MLP with an RMSE estimate of the generalization error $\hat{\epsilon} = 4.88$ MPa – the error distribution ranges from a minimum of 0.002 MPa to a maximum of 21.65 MPa with a median value of 4.19 MPa. We start by considering the interval $[30, 70]$ – see Table 2 (bottom). Looking at the Table, we can see that NeVer is able to establish unsafety of the intervals $[30, 70]$ and $[25, 75]$, whereas it exceeds the CPU time bound for all the remaining intervals that we tried. In all these cases, performances heavily depend on HySAT computation time when solving the constraint network corresponding to the abstract MLP. For instance, the time spent checking the interval $[25, 75]$ is several orders of magnitude smaller than the one reported for $[30, 70]$ beacuse HySAT is much faster on the former. Indeed, since proving unsafety is about finding a solution to the constraint network, an occasionally good heuristic choice may have a dramatic impact on runtime.

Given that there is only one parameter governing the abstraction, we may consider whether starting with a precise abstraction, i.e., setting a relatively small value of p, would bring any advantage. However, we should take into account that the smaller is p, the larger is the HySAT internal propositional encoding to check safety in the abstract domain. As a consequence, HySAT computations may turn out to be unfeasibly slow if the starting value of p is too small. To see this, let us focus on the PUMA case study. Considering the range $[-0.65, 0.65]$ we see that Table 2 reports that it is safe, HySAT solves the abstract safety check with $p = 0.5$ in 0.13 CPU seconds, and NeVer performs a single CETAR loop. The corresponding propositional encoding accounts for 599 variables and 2501 clauses in this case. If we consider the same safety check using

$p = 0.05$, then we still have a single CETAR loop, but HySAT now runs for 30.26 CPU seconds, with an internal encoding of 5273 variables and 29322 clauses. Notice that the CPU time spent by HySAT in this single case is already more than the *sum* of its runtime across all the cases in Table 2. Setting $p = 0.005$ confirms this trend: HySAT solves the abstract safety check in 96116 CPU seconds (about 27 hours), and the internal encoding accounts for 50774 variables and 443400 clauses. If we consider the product between variables and clauses as a rough estimate of the encoding size, we see that a $10\times$ increase in precision corresponds to at least a $100\times$ increase in the size of the encoding. Regarding CPU times, there is more than a $200\times$ increase going from $p = 0.5$ to $p = 0.05$, and more than a $3000\times$ increase when going from $p = 0.05$ to $p = 0.005$. In light of these results, it seems reasonable to start with coarse abstractions and let the CETAR loop refine them as needed.

4.3 Improving safety with repair

In the previous subsection we have established that, in spite of a very low generalization error, there are specific inputs to the MLP which trigger a misbehavior. As a matter of fact, the bound in which we are able to guarantee safety would not be very satisfactory in a practical application: For instance, in the case of PUMA dataset it is about 64% larger than the desired one. This result begs the question of whether it is possible to improve MLPs response using the output of NeVer. In this subsection, we provide strong empirical evidence that adding spurious counterexamples to the dataset Δ and training a new MLP, yields a network whose safety bounds are tighter than the original one.

Let us focus on the PUMA 500 case study. Since our forward kinematics dataset is obtained with a simulator, whenever a spurious counterexample is found, i.e., a vector of joint angles causing a misbehavior in the abstract network, we can compute the *true* response of the system, i.e., the final position of the manipulator along a single axis. While this is feasible in our experimental setting, the problem is that MLPs are useful exactly in those cases where the process which generated the dataset cannot be simulated with standard numerical techniques – as in the case of CCS case study. However, for the moment we defer this problem and try to prove that repair is *in principle* a good technique. In particular, we wish to show that adding spurious counterexamples to the dataset Δ and training a new MLP inside the CETAR loop, may yield networks whose safety bounds are tighter than the original ones. Since Δ must contain patterns of the form $(\langle\theta_1, \ldots, \theta_6\rangle, y)$, and counterexamples are interval vectors of the form $\tilde{s} = \langle[\theta_1], \ldots, [\theta_6]\rangle$ we have the problem of determining the pattern corresponding to $\tilde{s}$ which must be added in Δ. Let ν be the MLP under test, and $\tilde{s}$ be a corresponding spurious counterexample. We proceed in two steps: First, we extract a concrete input vector $\underline{s} = \langle\theta_1, \ldots, \theta_6\rangle$ from $\tilde{s}$ as described in the previous Section. Second, we compute $\nu(\underline{s})$, and we add the pattern $(\underline{s}, \nu(\underline{s}))$ to Δ. As we can see in Figure 2, if $\tilde{s}$ is a spurious counterexample, the computations of $\underline{s}$ and $\nu(\underline{s})$ come for free because they are needed to check feasibility (line 7).

Given the considerations above, our first experiment leverages spurious counterexamples together with the true response of the system – a process that we call *manual-repair* in the following. We start by considering the tightest SAFE interval in Table 2 (top) ($[-0.575, 0.575]$), and we proceed as follows:

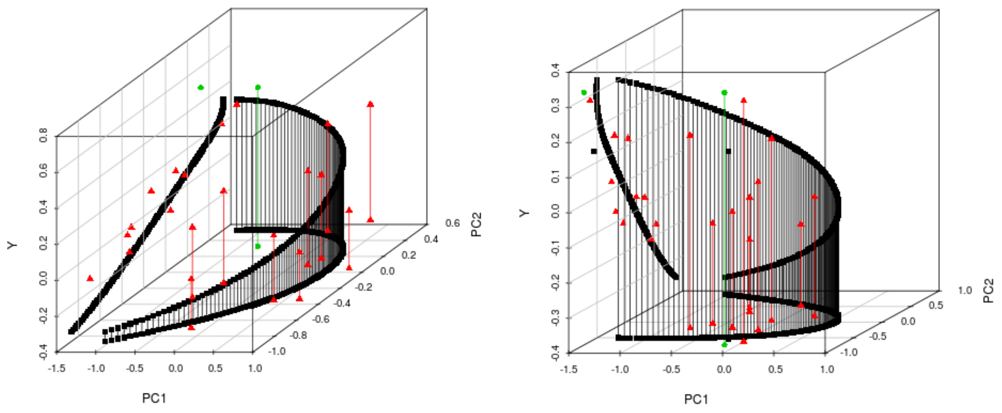

Fig. 4 Representation of ROBOOP and MLPs input-output in the manual-repair experiment. The plane (PC1-PC2) at the bottom is a two-dimensional projection of the input space obtained considering only the first two components of a Principal Component Analysis (PCA) of the input vectors – see, e.g., Chap. 7 of [32] for an introduction to PCA. The Y axis is the output of ROBOOP and the MLPs under test. The plane (Y-PC2) on the left shows the output vs. the second principal component. All square points in space are the output of ROBOOP corresponding to the input vectors, and we also show them projected onto the (Y-PC2) plane. Circles and triangles in space are the output of the MLPs under test: circles correspond to spurious counterexamples obtained by NeVer; triangles correspond to random input samples that we use as control; for both of them we also show their projection onto the (Y-PC2) plane. For all data points, a line joins the output of the system – either ROBOOP or the MLPs under test – to the corresponding input pattern in the (PC1-PC2) plane.

1. We train a new MLP ν_1 using the dataset $\Delta_1 = \Delta \cup (\underline{s}_1, f(\underline{s}_1))$ where Δ is the original dataset, $\underline{s}_1$ is extracted from $\tilde{s}$ after the first execution of the CETAR loop during the check of $[-0.575, 0.575]$, and $f(\underline{s}_1)$ is the output of the simulator.
2. We sample 10 different input vectors $\underline{r}_1, \ldots, \underline{r}_{10}$, uniformly at random from the input space; for each of them, we obtain a dataset $\Gamma_i = \Delta \cup (\underline{r}_i, f(\underline{r}_i))$ where Δ and f are the same as above; finally we train 10 different MLPs $\mu_1, \ldots, \mu_{10}$, where μ_i is trained on Γ_i for $1 \leq i \leq 10$.

Given these new MLPs, we check for their safety with NeVer. In the case of ν_1 we are able to show that the range $[-0.4, 0.4]$ is safe, which is already a considerable improvement over $[-0.575, 0.575]$. On the other hand, in the case of $\mu_1, \ldots, \mu_{10}$ the tightest bounds that we can obtain range from $[-0.47, 0.47]$ to $[-0.6, 0.6]$. This means that a targeted choice of a "weak spot" driven by a spurious counterexample turns out to be winning over a random choice. This situation is depicted in Figure 4 (left), where we can see the output of the original MLP ν corresponding to $\underline{s}_1$ (circle dot) and to $\underline{r}_1 \ldots \underline{r}_{10}$ (triangle dots). As we can see, $\nu(\underline{s}) = 0.484$ is outside the target bound of $[-0.35, 0.35]$ – notice that $f(\underline{s}) = 0.17$ in this case. On the other hand, random input vectors do not trigger, on average, an out-of-range response of ν[6]. We repeat steps 1 and 2 above, this time considering Δ_1 as the initial dataset, and thus computing a new dataset $\Delta_2 = \Delta_1 \cup (\underline{s}_2, f(\underline{s}_2))$ where $\underline{s}_2$ is extracted from $\tilde{s}$ after the second

[6] Notice that $\underline{s}$ is still spurious in this case because we are aiming to the bound $[-0.575, 0.575]$.

	l	h	Result	#	Time		
					Total	MLP	HySAT
PUMA	-0.350	0.350	UNSAFE	11	9.50	7.31	1.65
	-0.400	0.400	UNSAFE	7	6.74	4.68	1.81
	-0.425	0.425	UNSAFE	13	24.93	8.74	1.52
	-0.450	0.450	SAFE	3	3.11	1.92	1.10
CCS	30	70	SAFE	9	85.16	81.07	1.65

Table 3 Safety checking with NeVer and repair. The table is organized as Table 2, with the only exception of column "MLP", which reports the CPU time used to train the MLP.

execution of the CETAR loop. We consider a new MLP ν_2 trained on Δ_2, as well as other ten networks trained adding a random input pattern to Δ_1. Checking safety with NeVer, we are now able to show that the range $[-0.355, 0.355]$ is safe for ν_2, while the safety intervals for the remaining networks range from $[-0.4, 0.4]$ to $[-0.56, 0.56]$. In Figure 4 (right) we show graphically the results of this second round, where we can see that again, the response of $\nu_1(\underline{s}_2)$ is much closer to the target bound than the response of ν_1 when considering random input patterns. In the end, the above manual-repair experiment provides strong empirical evidence that spurious counterexamples are significantly more informative than other input patterns and they can help in improving the original safety bounds. However, a precise theoretical explanation of the phenomenon remains to be found. In this regard, we also notice that there are cases in which training on a dataset enlarged by a single pattern may cause NeVer to be unable to prove the same safety bound that could be proven before. In other words, safety is not guaranteed to be preserved when adding patterns and retraining.

As we mentioned before, however, the real problem with repair is that the true response of the system is not accessible. In the following, we show that even in such cases the original MLP can be repaired, at least to some extent, by leveraging spurious counterexamples *and* the response of the concrete MLP under test. Intuitively, this makes sense because the concrete MLP ought to be an accurate approximation of the target function. In order to test automated repair, we run a new experiment similar to the one shown in the previous subsection, with the aim of assessing whether we can improve the safety of the MLP in a completely automated, yet efficient, way. Consequently, we are now considering the whole procedure depicted in Figure 2.

Concerning the PUMA dataset, our goal is again finding values of l and h as close as possible to the ones for which the controller was trained. Table 3 (top) shows the result of the experiment above. As we can see in the Table, we can now claim that the MLP prediction will never exceed the range $[-0.450, 0.450]$, an interval which is "only" 28% larger than the desired one. Using the repairing facility in NeVer we have been able to shrink the safe interval of about 0.125m with respect to the result obtained without repairing. This gain comes at the expense of more CPU time spent to retrain the MLP, which happens whenever we find a spurious counterexample, independently of whether NeVer will be successful in repairing the network. For instance, considering the range $[-0.350, 0.350]$ in Table 2, we see that the total CPU time spent to declare the network unsafe is 1.95s without repairing, whereas the same result with repairing takes 9.50s in Table 3. Notice that updating the MLP also implies an increase of the total amount of CETAR loops (from 8 to 11). On the other hand, still considering the range $[-0.350, 0.350]$, we can see that the average time spent by HySAT to check the abstract network is about the same for the two cases.

Concerning the CCS dataset, looking at Table 3 (bottom), we are able to claim that the MLP prediction will never exceed the range for which it was designed, i.e. $[30, 70]$. As for the PUMA dataset, from the Table we can see that most of the CPU time needed to check safety (95%) is used to retrain the MLP. In this case, repairing allows NeVer to return a result that was not possible to obtain without it. For the sake of completeness, we report that the same result can be obtained considering all the intervals listed in Table 2, this time without exceeding the CPU time bound.

Since we have shown in the previous subsection that reducing p is bound to increase HySAT runtimes substantially, automated repairing with a fixed p could be an option. Indeed, the repair procedure generates a new ν at each execution of the CETAR loop, independently from the value of p. Even if it is possible to repair the original MLP without refinement, our experiments show that it can be less effective than repair coupled with refinement. Let us consider the results reported in Table 3, and let $p = 0.5$ for each loop. We report that NeVer returns SAFE for the interval $[-0.450, 0.450]$ after 59.12s and 36 loops. The first consideration about this result concerns the CPU time spent, which is one order of magnitude higher than repair with refinement, and it is mainly due to the higher number of retrainings. The second consideration is about the total amount of loops. Considering that the proportion of new patterns with respect to the original dataset is about 25%, and also considering that $p = 0.5$ is rather coarse, we incur into a high risk of overfitting the MLP.

5 Discussion and related work

This work builds on and extends [13]. Novel contributions include a presentation of NeVer architecture and a more thorough experimental analysis, including experiments about scalability. Contributions that are close to ours include a series of papers by Gordon, see e.g. [33], which focus on the domain of discrete-state systems with adaptive components. Since MLPs are stateless and defined over continuous variables, the results of [33] and subsequent works are unsuitable for our purposes. Robot control in the presence of safety constraints is a topic which is receiving increasing attention in recent years – see, e.g., [34]. However, the contributions in this area focus mostly on the verification of traditional, i.e., non-adaptive, methods of control. While this is a topic of interest in some fields of ML and Robotics – see, e.g., [3,4] – such contributions do not attack the problem using formal methods. Finally, since learning the weights of the connections among neurons can be viewed as synthesizing a relatively simple parametric program, our repairing procedure bears some resemblance with the counterexample-driven inductive synthesis presented in [14], and the abstraction-guided synthesis presented in [15]. In both cases the setting is quite different, as the focus is on how to repair concurrent programs. However, it is probably worth investigating further connections of our work with [14,15] and, more in general, with the field of inductive programming.

6 Conclusions

Summing up, the abstraction-refinement approach that we proposed allows the application of formal methods to verify and repair MLPs. The two key results of our work are (*i*) showing that a consistent abstraction mechanism allows the verification of realistic MLPs, and (*ii*) showing that our approach can improve the safety of the MLP in a completely automated way.

Our future work will focus on improving the automated repairing technique for MLPs and on extending NeVer to work with other ML algorithms.

Acknowledgments

This research has received funding from the European Community's Information and Communication Technologies Seventh Framework Programme [FP7/2007-2013] under grant agreement N. 215805, the CHRIS project. We would like to thank Dr. Lorenzo Natale for useful discussions and comments on the repair algorithm, and the anonymous reviewers who helped us to improve the quality of the original manuscript

References

1. G.P. Zhang. Neural networks for classification: a survey. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, 30(4):451–462, 2000.
2. D.J. Smith and K.G.L. Simpson. *Functional Safety – A Straightforward Guide to applying IEC 61505 and Related Standards (2nd Edition)*. Elsevier, 2004.
3. J. Schumann, P. Gupta, and S. Nelson. On verification & validation of neural network based controllers. In *Proc. of International Conf. on Engineering Applications of Neural Networks (EANN'03)*, 2003.
4. Z. Kurd, T. Kelly, and J. Austin. Developing artificial neural networks for safety critical systems. *Neural Computing & Applications*, 16(1):11–19, 2007.
5. E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):263, 1986.
6. J. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *International Symposium on Programming*, pages 337–351. Springer, 1982.
7. T. Schubert. High level formal verification of next-generation microprocessors. In *Proceedings of the 40th annual Design Automation Conference*. ACM, 2003.
8. T. Ball, B. Cook, V. Levin, and S.K. Rajamani. SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. In *Integrated Formal Methods*, pages 1–20. Springer, 2004.
9. A. Armando, R. Carbone, and L. Compagna. LTL Model Checking for Security Protocols. In *20th IEEE Computer Security Foundations Symposium*, pages 385–396, 2007.
10. R. Alur, T.A. Henzinger, and P. Ho. Automatic Symbolic Verification of Embedded Systems. In *IEEE Real-Time Systems Symposium*, pages 2–11, 1993.
11. E.M. Clarke, O. Grumberg, and D.A. Peled. *Model checking*. Springer, 1999.
12. K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.
13. L. Pulina and A. Tacchella. An Abstraction-Refinement Approach to Verification of Artificial Neural Networks. In *22nd International Conference on Computer Aided Verification (CAV 2010)*, volume 6174 of *Lecture Notes in Computer Science*, pages 243–257. Springer, 2010.
14. A. Solar-Lezama, C.G. Jones, and R. Bodik. Sketching concurrent data structures. In *2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 136–148. ACM, 2008.
15. M. Vechev, E. Yahav, and G. G. Yorsh. Abstraction-guided synthesis of synchronization. In *37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 327–338. ACM, 2010.
16. C. Igel, T. Glasmachers, and V. Heidrich-Meisner. Shark. *Journal of Machine Learning Research*, 9:993–996, 2008.

17. M. Franzle, C. Herde, T. Teige, S. Ratschan, and T. Schubert. Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure. *Journal on Satisfiability, Boolean Modeling and Computation*, 1:209–236, 2007.
18. I.C. Yeh. Modeling of strength of high-performance concrete using artificial neural networks. *Cement and Concrete research*, 28(12):1797–1808, 1998.
19. S. Haykin. *Neural networks: a comprehensive foundation*. Prentice Hall, 2008.
20. A.K. Mackworth. Consistency in networks of relations. *Artificial intelligence*, 8(1):99–118, 1977.
21. Pascal Van Hentenryck. Numerica: A modeling language for global optimization. In *Fifteenth International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1642–1650, 1997.
22. F. Rossi, P. Van Beek, and T. Walsh. *Handbook of constraint programming*. Elsevier Science Ltd, 2006.
23. V. Barichard and J.K. Hao. A population and interval constraint propagation algorithm. In *Evolutionary Multi-Criterion Optimization, Second International Conference (EMO 2003)*, pages 88–101. Springer, 2003.
24. J. Marques-Silva, I. Lynce, and S. Malik. Conflict-driven clause learning SAT solvers. *Handbook of Satisfiability, IOS Press, Amsterdam*, pages 131–153, 2009.
25. C. Barrett, R. Sebastiani, S.A. Seshia, and C. Tinelli. Satisfiability modulo theories. *Handbook of Satisfiability, IOS Press, Amsterdam*, pages 825–885, 2009.
26. C. Jermann, D. Sam-Haroud, and G. Trombettoni, editors. *CP Workshop on Interval analysis, constraint propagation, applications (IntCP 2009).*, 2009.
27. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, 1977.
28. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM (JACM)*, 50(5):794, 2003.
29. I. Mierswa, M. Wurst, R. Klinkenberg, M. Scholz, and T. Euler. Yale: Rapid prototyping for complex data mining tasks. In *12th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD'06)*, pages 935–940, New York, NY, USA, 2006. ACM.
30. R. Gordeau. Roboop – a robotics object oriented package in C++, 2005. `http://www.cours.polymtl.ca/roboop`.
31. J.R. Rabunal and J. Dorrado. *Artificial neural networks in real-life applications*. Idea Group Pub, 2006.
32. I.H. Witten and E. Frank. *Data Mining (2nd edition)*. Morgan Kaufmann, 2005.
33. D.F. Gordon. Asimovian adaptive agents. *Journal of Artificial Intelligence Research*, 13(1):95–153, 2000.
34. G. Pappas and H. Kress-Gazit, editors. *ICRA Workshop on Formal Methods in Robotics and Automation*, 2009.

A NeVer input format

```
<input>  ::= <pars> <atts> <class> <data> <safety_condition> <EOF>
<pars>   ::= <prob> <mlp> <solver> <abs>
<prob>   ::= P <PINT> <PINT> <EOL>
<mlp>    ::= M <PINT> <PINT> <EOL>
<solver> ::= S <PREAL> <PREAL> <EOL>
<abs>    ::= A <PREAL> <PREAL> <EOL>

<atts>    ::= ATTRIBUTES <EOL>
              <att_set>
              END_ATTRIBUTES <EOL>
<att_set> ::= <att> <att_set> | <att>
<att>     ::= <text> <PREAL> <PREAL> <EOL>
<class>   ::= CLASS <EOL>
              <att>
              END_CLASS <EOL>

<data>      ::= DATA <EOL>
                <patterns>
                END_DATA <EOL>
<patterns>  ::= <pattern> <patterns> | <pattern>
<pattern>   ::= <REAL>,<REAL> <EOL>  | <real_list>,<REAL> <EOL>
<real_list> ::= <REAL>,<real_list>   | <REAL>

<safety_condition> ::= SAFETY_CONDITIONS <EOL>
                       <text>
                       END_SAFETY_CONDITIONS <EOL>

<TEXT>  ::= {A sequence of non-special ASCII characters}
<PINT>  ::= {An integer greater than 0}
<PREAL> ::= {A real greater than 0}
<REAL>  ::= {A real}
<EOL>   ::= {The "End Of Line" character}
```

B The encoding

In the following we sketch the encoding of $\tilde{\nu}$ in the input language of HySAT[7] which is a syntactic variant of the logic language presented in Section 2. For the sake of explanation, let us consider the PUMA 500 case study. In this case, the input vector $\underline{x} = \langle \theta_1, \ldots, \theta_6 \rangle$ and the output of the network *out* can be described to HySAT as

```
float [-0.92, 0.39]     theta1;
float [-0.04, 0]        theta2;
float [-0.06, 0.27]     theta3;
float [-0.0001, 0.0001] theta4;
float [-0.24, 0.06]     theta5;
float [-0.39, 0.92]     theta6;
float [-2, 2]           out;
```

The above constraints state that each of the input/output signal is a real-valued variable (type `float`) taking values in some interval. This corresponds to pair of unary constraints of the form, e.g., $\theta_1 \geq -0.92$ and $\theta_1 \leq -0.39$.

Next, the encoding must deal with the weights computed by training the MLP. These values are known before the verification phase, and they do not change during it, so we can encode them as symbolic constants. In HySAT this can be done using the `define` keyword, which is similar to a macro definition in programming languages. For instance, the weights a_{ji} and the bias b_j of the first hidden neuron ($j = 1$) are encoded as

```
define a_11 = -1.54809;
define a_12 =  2.95285;
define a_13 = -0.05130;
define a_14 =  0.05023;
define a_15 = -1.59864;
define a_16 =  1.50692;
define b_1  = -0.11367;
```

and similar definitions need to be repeated for all the parameters of the MLP. As for the ILFs, we need constraints expressing Definition 1 in Section 2. For instance, the ILF of the first hidden neuron $\tilde{y_1}$ is defined as

```
y_1 = (theta1 * a_11) + (theta2 * a_12) + (theta3 * a_13) +
      (theta4 * a_14) + (theta5 * a_15) + (theta6 * a_16) + b_1;
```

To encode $\tilde{\sigma}_p$, we provide a set of constraints that correspond to the over-approximation of σ, the coarseness of which is determined by the value of p. For instance, if $p = 0.5$, and focusing on the range $x \in [-1, 1]$, we have

```
...
(y_1 > -1   and y_1 <= -0.5) ->
(sigma_y_1 >= 0.26894 and sigma_y_1 <= 0.39394);
(y_1 > -0.5 and y_1 <= 0)    ->
(sigma_y_1 >= 0.37754 and sigma_y_1 <= 0.50254);
(y_1 > 0    and y_1 <= 0.5)  ->
(sigma_y_1 >= 0.5      and sigma_y_1 <= 0.625);
(y_1 > 0.5  and y_1 <= 1)    ->
(sigma_y_1 >= 0.62246 and sigma_y_1 <= 0.74746);
...
```

[7] HySAT user guide is available at `http://hysat.informatik.uni-oldenburg.de/user_guide/hysat-user-guide.pdf`.

where the operator "$\rightarrow$" is Boolean implication, and `sigma_y_1` is a fresh variable used to store the result of the application of $\tilde{\sigma}_p$ to the related ILF. The encoding of the MLP is complete once we consider the output value

```
    out = (c_1 * sigma_y_1) + ... + d;
```

When it comes to check safety thresholds, we consider all the constraints above plus another, representing the safety property to be checked. From Section 2, we recall that the output of the MLP must range in $[-0.35, 0.35]$. The corresponding constraint that we add to HySAT is

```
    (out < -0.35) or (out > 0.35);
```

Intuitively, if HySAT manages to satisfy the above constraint network, this means that the output of the abstract MLP *can* exceed the safety threshold, so the concrete MLP is potentially unsafe. On the other hand, if HySAT fails to satisfy the constraint network, then the abstract MLP is safe, and so is the concrete one.