An Abstraction-Refinement Approach to Verification of Artificial Neural Networks [*]

Luca Pulina and Armando Tacchella

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

Abstract. A key problem in the adoption of artificial neural networks in safety-related applications is that misbehaviors can be hardly ruled out with traditional analytical or probabilistic techniques. In this paper we focus on specific networks known as Multi-Layer Perceptrons (MLPs), and we propose a solution to verify their safety using abstractions to Boolean combinations of linear arithmetic constraints. We show that our abstractions are consistent, i.e., whenever the abstract MLP is declared to be safe, the same holds for the concrete one. Spurious counterexamples, on the other hand, trigger refinements and can be leveraged to automate the correction of misbehaviors. We describe an implementation of our approach based on the HYSAT solver, detailing the abstraction-refinement process and the automated correction strategy. Finally, we present experimental results confirming the feasibility of our approach on a realistic case study.

1 Introduction

Artificial neural networks are one of the most investigated and well-established Machine Learning techniques, and they find application in a wide range of research and engineering domains – see, e.g., [1]. However, in spite of some exceptions, neural networks are confined to systems which comply only to the lowest safety integrity levels, achievable with standard industrial best practices [2]. The main reason is the absence of effective safety assurance methods for systems using neural networks. In particular, traditional analytical and probabilistic methods can be ineffective in ensuring that outputs do not generate potential hazards in safety-critical applications [3] .

In this paper we propose a formal method to verify safety of neural networks. We consider a specific kind of feed-forward neural netldork known as Multi-Layer Perceptron (MLP), and we state that an MLP is safe when, given every possible input value, its output is guaranteed to range within specific bounds. Even if we consider MLPs with a fairly simple topology, the *Universal Approximation Theorem* [4] guarantees that, in principle, such MLPs can approximate every non-linear real-valued function of n real-valued inputs. Also, our notion of safety is representative of all the cases in which an out-of-range response is unacceptable, such as, e.g., minimum and maximum reach of

[*] 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 the anonymous reviewers who helped us to improve the quality of the original manuscript.

an industrial manipulator, lowest and highest percentage of a component in a mixture, and minimum and maximum dose of a drug that can be administered to a patient.

Our first contribution, in the spirit of [5], is the abstraction of MLPs to corresponding Boolean combinations of linear arithmetic constraints. 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 [6]. Even considering rational approximations of real numbers, the amount of computational resources required to reason with realistic networks could still be prohibitive. For the MLPs that we consider, we show that our abstraction mechanism yields consistent over-approximations of concrete networks, i.e., once the abstract MLP is proven to be safe, the same holds true for the concrete one. Clearly, abstraction opens the path to spurious counterexamples, i.e., violations of the abstract safety property which fail to realize on the concrete MLP. In these cases, since we control the "coarseness" of the abstraction through a numeric parameter, it is sufficient to modify such parameter to refine the abstraction and then retry the verification. While our approach is clearly inspired by counterexample guided abstraction-refinement (CEGAR) [7], in our case refinement is not guided by the counterexample, but just caused by it, so we speak of counterexample *triggered* abstraction-refinement (CETAR).

Our second contribution is a strategy for automating MLP *repair* – a term borrowed from [8] that we use to indicate modifications of the MLP synthesis attempting to correct its misbehaviors. The idea behind repair is simple, yet fairly effective. The problem with an unsafe network is that it should be redesigned to improve its performances. This is more of an art than a science, and it has to do with various factors, including the knowledge of the physical domain in which the MLP operates. However, spurious counterexamples open an interesting path to automated repair, because they are essentially an input vector which would violate the safety constraints if the concrete MLP were to respond with less precision than what is built in it. Intuitively, 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. We provide strong empirical evidence in support of this intuition, and also in support of the fact that adding spurious counterexamples to the training set yields MLPs which are safer than the original ones.

We implemented the above ideas in the tool NEVER (for **Ne**ural networks **Ver**ifier) [9] which leverages HYSAT [6] to verify abstract networks and the SHARK library [10] to provide MLP infrastructure, including representation and support for evaluation and repairing. In order to test the effectiveness of our approach, we experiment with NEVER on a case study about learning the forward kinematics of an industrial manipulator. We aim to show that NEVER can handle realistic sized MLPs, as well as support the MLP designer in establishing or, at least, in improving the safety of his design in a completely automated way. The paper is structured as follows. Section 2 is a crash-course on MLPs – introducing basic notation, terminology and methodologies – and includes a detailed description of our case study. In Section 3 we describe MLP abstraction, and we prove its consistency. We also describe the basic CETAR algorithm, and we show some experiments confirming its feasibility. In Section 4, we extend the basic algorithm with automated repair, we provide empirical evidence to support the correctness of our

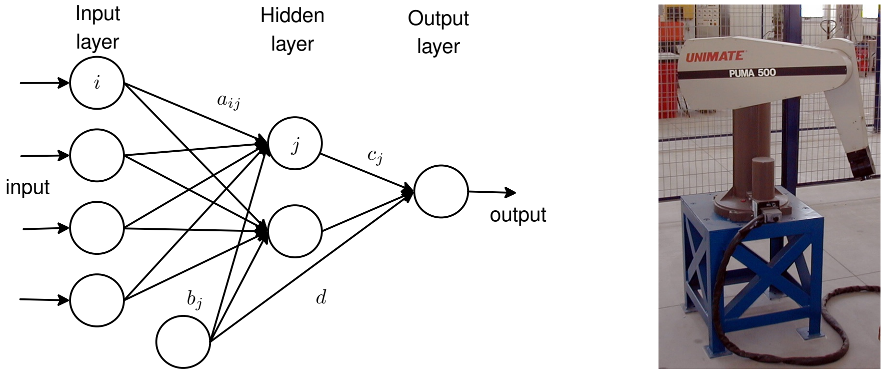

Fig. 1. Left: our MLP architecture of choice; neurons and connections are represented by circles and arrows, respectively. Right: PUMA 500 industrial manipulator.

approach, and we show experiments confirming its effectiveness in our case study. We conclude the paper in Section 5 with some final remarks and a comparison of our work with related literature.

2 Preliminaries

Structure Multi-Layer Perceptrons (MLPs) [11] are probably the most widely studied and used type of artificial neural network. An MLP is composed of a system of interconnected computing units (neurons), which are organized in layers. Figure 1 (left) 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. An *output layer* composed of at least a 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. An MLP processes the information as follows. 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 y_j received by neuron j, with $j = \{1, \ldots, m\}$, is called *induced local field* (ILF) and it is defined as

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

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 [4]. A commonly used activation function [11] is the *logistic function*

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

Therefore, the output of the MLP is

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

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 $f : \mathbb{R}^n \rightarrow \mathbb{R}$ 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.

Training and Validation 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) [11]. 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. Indeed, 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., [11]. 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. [11]).

Case Study The experiments that we present[1] concern a realistic case study about the control of a Unimate PUMA 500 industrial manipulator – see Figure 1 (right). This is a 6 degrees-of-freedom 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. Our case study focuses on learning *forward kinematics*, i.e., the mapping from joint angles to end-effector position along a single coordinate of a Cartesian system having origin in the center of the robot's workspace. Our desiderata is thus to build an MLP predicting the final position of the end-effector knowing the joint angles. 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. An 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 have $y \in [-0.35, 0.35]$. We have built the training set using the ROBOOP library [12] which provides facilities for simulating the PUMA manipulator. The MLP was trained using the IRpropPlus algorithm [13], which is a modern implementation of BP. Inside our system, training an MLP to perform forward kinematics takes 0.64s across 500 epochs, yielding a RMSE estimate of the generalization error $\hat{\epsilon} = 0.024$m – 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. It is worth noticing that such generalization error would be considered very satisfactory in MLP applications.

3 Verifying MLPs with abstraction

Given an MLP ν with n inputs and a single output we define

- the *input domain* of ν 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 $a_i, b_i \in \mathbb{R}$; and
- the *output domain* of ν as a closed interval $\mathcal{O} = [a, b]$ bounded by $a, b \in \mathbb{R}$.

[1] Our empirical analysis is 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`.

In the definition above, and throughout the rest of the paper, a closed interval $[a, b]$ bounded by $a, b \in \mathbb{R}$ is the set of real numbers comprised between a and b, i.e, $[a, b] = \{x \mid a \leq x \leq b\}$ with $a \leq b$. We thus consider any MLP ν as a function $\nu : \mathcal{I} \rightarrow \mathcal{O}$, and we say that ν is *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. Testing exhaustively all the input vectors in $\mathcal{I}$ to make sure that ν respects condition (5) is untenable. On the other hand, statistical approaches based on sampling input vectors – see, e.g., [14] – can only give a probabilistic guarantee. In the spirit of [5], we propose to verify 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. This comes at the price of *spurious counterexamples*, i.e., there may exist some abstract counterexamples that do not correspond to concrete ones. A spurious counterexample calls for a refinement of the abstraction which, in turn, can make the verification process more expensive. In practice, we hope to be able to either verify ν or exhibit a counterexample within a reasonable number of refinements.

Following the framework of [5], we build 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 we have the usual containment relation "$\sqsubseteq$" such that given two intervals $[a, b] \in [\mathbb{R}]$ and $[c, d] \in [\mathbb{R}]$ we have 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 $\alpha : 2^{\mathbb{R}} \rightarrow [\mathbb{R}]$ such that

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

In other words, given a set $X \subseteq \mathbb{R}$, $\alpha(X)$ is the smallest interval encompassing all the elements of X, i.e., for all $x \in X$, x ranges within $\alpha(X)$ and there is no $[a, b] \sqsubseteq \alpha(X)$ for which the same holds unless $[a, b]$ coincides with $\alpha(X)$. Conversely, given $[a, b] \in [\mathbb{R}]$, *concretization* is defined as the mapping $\gamma : [\mathbb{R}] \rightarrow 2^{\mathbb{R}}$ 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 indeed a Galois connection because the following four properties follow from definitions (6) and (7):

1. Given two sets $X, Y \in 2^{\mathbb{R}}$, if $X \subseteq Y$ then $\alpha(X) \sqsubseteq \alpha(Y)$.
2. Given two intervals $[a, b] \in [\mathbb{R}]$ and $[c, d] \in [\mathbb{R}]$, if $[a, b] \sqsubseteq [c, d]$ then $\gamma([a, b]) \subseteq \gamma([c, d])$.
3. Given a set $X \in 2^{\mathbb{R}}$, we have that $X \subseteq \gamma(\alpha(X))$.
4. Given an interval $[a, b] \in [\mathbb{R}]$, we have that $\alpha(\gamma([a, b])$ coincides with $[a, b]$.

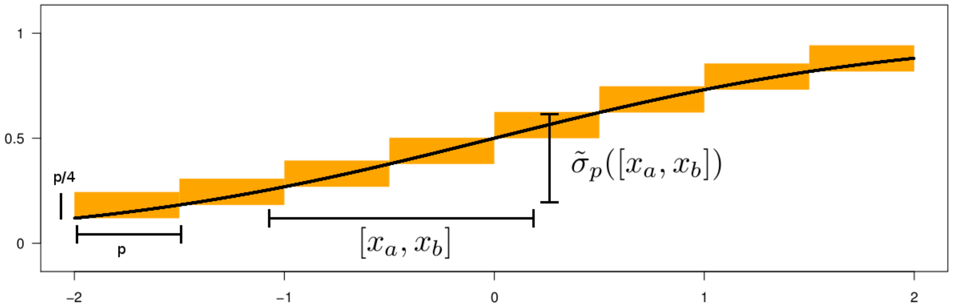

Fig. 2. Activation function $\sigma(x)$ and its abstraction $\tilde{\sigma}_p(x)$ in the range $x \in [-2, 2]$. The solid line denotes σ, while the boxes denote $\tilde{\sigma}_p$ with $p = 0.5$.

Let $\nu : \mathcal{I} \to \mathcal{O}$ denote the MLP for which we wish to prove safety in terms of (5). We refer to ν as the *concrete* MLP. Given a concrete domain $D = [a, b]$, the corresponding abstract domain is $[D] = \{[x, y] \mid a \leq x \leq y \leq b\}$, and we denote with $[x]$ a generic element of $[D]$. We can naturally extend the abstraction to Cartesian products of domains, i.e., given $\mathcal{I} = D_1 \times \ldots \times D_n$, we define $[\mathcal{I}] = [D_1] \times \ldots \times [D_n]$, and we denote with $[\underline{x}] = \langle [x_1], \ldots, [x_n] \rangle$ the elements of $[\mathcal{I}]$ that we call *interval vectors*. If $X \subseteq \mathcal{I}$ with $X = \{\underline{x}_1, \ldots, \underline{x}_k\}$ is a set of input vectors, then we can extend the abstraction function α by considering

$$\alpha(X) = \langle [\min_{1 \leq j \leq k} \{x_{1j}\}, \max_{1 \leq j \leq k} \{x_{1j}\}], \ldots, [\min_{1 \leq i \leq k} \{x_{nj}\}, \max_{1 \leq j \leq k} \{x_{nj}\}] \rangle \qquad (8)$$

where x_{ij} denotes the i-th component ($1 \leq i \leq n$) of the j-th vector in X ($1 \leq j \leq k$). The result of $\alpha(X)$ is thus the interval vector whose components are n intervals, each obtained by considering minimum and maximum of the corresponding components in the input vectors. An *abstract* MLP $\tilde{\nu}$ is a function $\tilde{\nu} : [\mathcal{I}] \to [\mathcal{O}]$. Given a set of input vectors $X \subseteq \mathcal{I}$, $\tilde{\nu}$ provides a *consistent abstraction* of ν if it satisfies

$$\{\nu(\underline{x}) \mid \underline{x} \in X\} \subseteq \gamma(\tilde{\nu}(\alpha(X))) \qquad (9)$$

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. Given our safety thresholds $l, h \in \mathcal{O}$, if we can prove

$$\forall [\underline{x}] \in [\mathcal{I}] : \tilde{\nu}([\underline{x}]) \sqsubseteq [l, h] \qquad (10)$$

then, from (9) and the definition of γ, it immediately follows that

$$\{\nu(\underline{x}) \mid \underline{x} \in \mathcal{I}\} \subseteq \{y \mid l \leq y \leq h\} \qquad (11)$$

which implies that condition (5) is satisfied by ν, because ν may not output a value outside $[l, h]$ without violating (11).

We abstract the concrete MLP ν assuming 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 $p \in \mathbb{R}^+$, the *abstract activation function* $\tilde{\sigma}_p$ can be obtained by considering the maximum increment of σ over intervals of length p. Since σ is a monotonically increasing function, and its first derivative is maximum in the origin, we can use the increment of σ in the origin as the upper bound on the increment of σ elsewhere. The tangent to σ in the origin has slope $1/4$ so we have that

$$\forall x \in \mathbb{R} : 0 \leq \sigma(x + p) - \sigma(x) \leq \frac{p}{4} \tag{12}$$

for any choice of the parameter $p \in \mathbb{R}^+$. Now let x_0 and x_1 be the values that satisfy $\sigma(x_0) = p/4$ and $\sigma(x_1) = 1 - p/4$, respectively. We define $\tilde{\sigma}_p : [\mathbb{R}] \to [\mathcal{O}_\sigma]$ as follows:

$$\tilde{\sigma}_p([x_a, x_b]) = \begin{cases} [0, p/4] & \text{if } x_b \leq x_0 \\ [0, \sigma(\lfloor \frac{x_b}{p} \rfloor) + \frac{p}{4}] & \text{if } x_a \leq x_0 \text{ and } x_b < x_1 \\ [\sigma(\lfloor \frac{x_a}{p} \rfloor), \sigma(\lfloor \frac{x_b}{p} \rfloor) + \frac{p}{4}] & \text{if } x_0 < x_a \text{ and } x_b < x_1 \\ [\sigma(\lfloor \frac{x_a}{p} \rfloor), 1] & \text{if } x_0 < x_a \text{ and } x_1 \leq x_b \\ [1 - p/4, 1] & \text{if } x_a \geq x_1 \end{cases} \tag{13}$$

Figure 2 gives a pictorial representation of the above definition. As we can see, $\tilde{\sigma}_p$ is a consistent abstraction of σ because it respects property (9) by construction. According to (13) we can control how much $\tilde{\sigma}_p$ over-approximates σ, since large values of p correspond to coarse-grained abstractions, whereas small values of p correspond to fine-grained ones. Formally, if $p < q$ then for all $[x] \in [\mathbb{R}]$, we have that $\tilde{\sigma}_p([x]) \sqsubseteq \tilde{\sigma}_q([x])$. We can now define $\tilde{\nu}_p : [\mathcal{I}] \to [\mathcal{O}]$ as

$$\tilde{\nu}_p([\underline{x}]) = \sum_{j=1}^{m} c_j \tilde{\sigma}_p(\tilde{y}_j([\underline{x}])) + d \tag{14}$$

where $\tilde{y}_j([\underline{x}]) = \sum_{i=1}^{n} a_{ji}[x_i] + b_j$, and we overload the standard symbols to denote products and sums, e.g., we write $x + y$ to mean $x \tilde{+} y$ when $x, y \in [\mathbb{R}]$. 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, defining $\tilde{\nu}_p$ as in (14) provides a consistent abstraction of ν. This means that our original goal of proving the safety of ν according to (5) can be now recast, modulo refinements, to the goal of proving its abstract counterpart (10).

We can leverage the above definitions to provide a complete abstraction-refinement algorithm to prove MLP safety. The pseudo-code in Figure 3 is at the core of our tool NEVER[2] which we built as proof of concept. NEVER 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; the latter is set to TRUE when an abstract counterexample can be realized on the

[2] NEVER is available for download at `http://www.mind-lab.it/never`. NEVER is written in C++, and it uses HYSAT to verify abstract MLPs and the SHARK library to handle representation, training, and repairing of the concrete MLPs.

```
NEVER($\mathit{\Delta}$, $\mathit{\Pi}$, $[l, h]$, $p$, $r$)
 1  isSafe ← FALSE; isFeasible ← FALSE
 2  $\nu$ ← TRAIN($\mathit{\Delta}$, $\mathit{\Pi}$)
 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      isFeasible ← CHECKFEASIBILITY($\nu$, $\tilde{s}$)
 8      if (not isFeasible) then
 9        $p$ ← $p$ / $r$
10  until isSafe or (not isSafe and isFeasible)
11  return isSafe
```

Fig. 3. Pseudo-code of NEVER.

concrete MLP. In line 2, a call to the function TRAIN yields a concrete MLP ν from the set $\mathit{\Delta}$. The set $\mathit{\Pi}$ 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 [10]. Lines 4 to 11 are the CETAR loop. Given p, the function ABSTRACT computes $\tilde{\nu}_p$ exactly as shown in (14) 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 $\tilde{\nu}_p : [\mathcal{I}] \rightarrow [\mathcal{O}]$, and defining the domains $[\mathcal{I}]$ and $\mathcal{O}$, plus a further constraint encoding the safety condition. 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.

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[3] from $\tilde{s}$. If the abstract counterexample can be realized 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 – and we restart the loop.

We conclude this section with an experimental account of NEVER using the case study introduced in Section 2. Our main target is to find a region $[l, h]$ within which

[3] We consider a vector whose components are the midpoints of the components of the interval vector emitted by HYSAT as witness.

l	h	Result	# CETAR	Time	
				Total	HySAT
-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

Table 1. Safety checking with NeVer. The first two columns ("l" and "h") report lower and upper safety thresholds, respectively. The third column reports the final result of NeVer, and column "# CETAR" 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.

we can guarantee a safe calculation of the forward kinematics by means of a trained MLP. To do so, we set the initial abstraction parameter to p = 0.5 and the refinement rate to r = 1.1, and we train an MLP with 3 neurons in the hidden layer. 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 with respect to given bounds, we enlarge the bounds. Once we have reached a safe configuration, we try to shrink the bounds, until we reach the tightest bounds that we can consider safe. The results of the above experiment are reported in Table 1. 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 these bounds, then NeVer is always able to find a set of inputs that makes the MLP exceed the bounds. 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. In both cases, the number of abstraction-refinement loops is also larger than other configurations that we tried.

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 keep 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 consider the range $[-0.65, 0.65]$ for which Table 1 reports that 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 1. 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 when 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. As we show in the following, efficiency of the automated repair heuristic is also another compelling reason behind this choice.

4 Repairing MLPs using spurious counterexamples

In the previous Section 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 bounds in which we are able to guarantee safety would not be very satisfactory in a practical application, since they are about 64% larger than the desired ones. This result begs the question of whether it is possible to improve MLPs response using the output of NEVER. In this section, we provide strong empirical evidence that adding spurious counterexamples to the dataset $\mathit{\Delta}$ and training a new MLP, yields a network whose safety bounds are tighter than the original ones. We manage to show this because our forward kinematics dataset is obtained with a simulator, so 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 position of the end-effector 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 target function $f : \mathcal{I} \rightarrow \mathcal{O}$ is unknown. However, 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. Our experiments show that adding spurious counterexamples to the dataset $\mathit{\Delta}$ and training a new MLP inside the CETAR loop, also yields networks whose safety bounds are tighter than the original ones. Since $\mathit{\Delta}$ 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 to $\mathit{\Delta}$. 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 $\mathit{\Delta}$. As we can see in Figure 3, if $\tilde{s}$ is a spurious counterexample, the computation of $\underline{s}$ already comes for free because it is needed to check feasibility (line 7).

Our first experiment shows that leveraging spurious counterexamples together with their true response – a process that we call *manual-repair* in the following – yields MLPs with improved safety bounds. We consider the tightest SAFE interval in Table 1 ($[-0.575, 0.575]$), and we proceed as follows:

1. We train a new MLP ν_1 using the dataset $\mathit{\Delta}_1 = \mathit{\Delta} \cup (\underline{s}_1, f(\underline{s}_1))$ where $\mathit{\Delta}$ 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 ten 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 $\mathit{\Gamma}_i = \mathit{\Delta} \cup (\underline{r}_i, f(\underline{r}_i))$ where $\mathit{\Delta}$

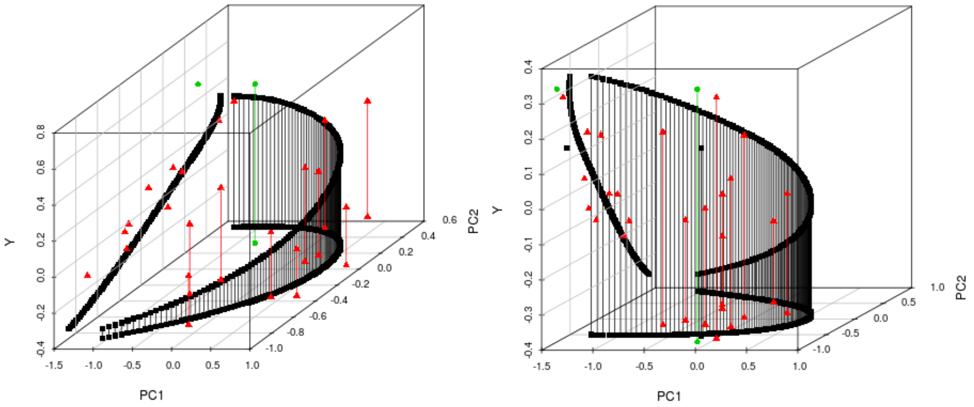

Fig. 4. Representation of ROBOOP and MLPs input-output in the manual-repair experiment. The plane (PC1-PC2) at the bottom is a two-dimensional projection of the input domain obtained considering only the first two components of a Principal Component Analysis (PCA) of the input vectors – see, e.g., Chap. 7 of [15] 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.

and f are the same as above; finally we train ten different MLPs $\{\mu_1, \ldots, \mu_{10}\}$, where μ_i is trained on Γ_i for $1 \leq i \leq 10$.

Given the MLP ν_1 and the control MLPs $\{\mu_1, \ldots, \mu_10\}$, 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[4]. 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 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

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

l	h	Result	# CETAR	Time		
				Total	MLP	HySAT
-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

Table 2. Safety checking with NeVer and repair. The table is organized as Table 1, with the only exception of column "MLP", which reports the CPU time used to train the MLP.

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 again that 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 randomly chosen input patterns and that 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 confirm the same safety bounds that could be proven before. In other words, safety is not guaranteed to be preserved when adding patterns and retraining.

To automate repairing, we modify NeVer by replacing lines 6-9 in the pseudo-code of Figure 3 with the following:

```
6 if (not isSafe) then
7   $o \leftarrow$ NIL; isFeasible $\leftarrow$ CHECKFEASIBILITY($\nu$, $\tilde{s}$, $o$)
8   if (not isFeasible) then
9     $p \leftarrow p\,/\,r$; $\Delta \leftarrow$ UPDATE($\Delta$, $\tilde{s}$, $o$); $\nu \leftarrow$ TRAIN($\Delta$, $\Pi$)
```

The parameter o is used to store the answer of ν when given $\tilde{s}$ as input. The rest of the code is meant to *update* the concrete MLP by (i) adding the input pattern extracted from the spurious counterexample $\tilde{s}$ and the corresponding output o to the set Δ, and (ii) training a new network on the extended set.

After this modification, we run a new experiment similar to the one shown in Section 3, with the aim of showing that we can improve the safety of the MLP in a completely automated, yet fairly efficient, way. Our goal is again finding values of l and h as close as possible to the ones for which the controller was trained. Table 2 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 $[-0.450, 0.450]$, which is "only" 28% larger than the desired one. Using this repairing heuristic in NeVer we are thus able to shrink the safety bounds of about 0.125m with respect to those 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 1, 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 2. 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.

Since we have shown in the previous Section 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 this can be less effective than repair coupled with refinement. Let us consider the results reported in Table 2, and let $p = 0.5$ for each loop. We report the 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 also incur into a high risk of overfitting the MLP.

5 Conclusion and Related Work

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 repair heuristic can improve the safety of MLPs in a completely automated way. To the best of our knowledge, this is the first time in which formal verification of a functional Machine Learning technique is investigated. Contributions that are close to ours include a series of paper by Gordon, see e.g. [8], which focus on the domain of discrete-state systems with adaptive components. Since MLPs are stateless and defined over continuous variables, the results of [8] 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., [16]. 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 Machine Learning and Robotics – see, e.g., [14, 3] – 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 resemblances with the counterexample-driven inductive synthesis presented in [17], and the abstraction-guided synthesis presented in [18]. 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 [17, 18] and, more in general, with the field of inductive programming.

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. Z. Kurd, T. Kelly, and J. Austin. Developing artificial neural networks for safety critical systems. *Neural Computing & Applications*, 16(1):11–19, 2007.
4. K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.
5. 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.
6. 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.
7. 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.
8. D.F. Gordon. Asimovian adaptive agents. *Journal of Artificial Intelligence Research*, 13(1):95–153, 2000.
9. L. Pulina and A. Tacchella. NEVER: A tool for Neural Network Verification, 2010. `http://www.mind-lab.it/never`.
10. Christian Igel, Tobias Glasmachers, and Verena Heidrich-Meisner. Shark. *Journal of Machine Learning Research*, 9:993–996, 2008.
11. S. Haykin. *Neural networks: a comprehensive foundation*. Prentice Hall, 2008.
12. R. Gordeau. Roboop – a robotics object oriented package in C++, 2005. `http://www.cours.polymtl.ca/roboop`.
13. C. Igel and M. Husken. Empirical evaluation of the improved Rprop learning algorithms. *Neurocomputing*, 50(1):105–124, 2003.
14. 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.
15. I.H. Witten and E. Frank. *Data Mining (2nd edition)*. Morgan Kaufmann, 2005.
16. G. Pappas and H. Kress-Gazit, editors. *ICRA Workshop on Formal Methods in Robotics and Automation*, 2009.
17. 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.
18. 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.