Formal Verification of Neural Networks: a Case Study about Adaptive Cruise Control

Stefano Demarchi, Dario Guidotti, Andrea Pitto, Armando Tacchella

KEYWORDS

Safety Evaluation, Dependable Systems, Neural Networks, Formal Verification.

ABSTRACT

Formal verification of neural networks is a promising technique to improve their dependability for safety critical applications. Autonomous driving is one such application where the controllers supervising different functions in a car should undergo a rigorous certification process. In this paper we present an example about learning and verification of an adaptive cruise control function on an autonomous car. We detail the learning process as well as the attempts to verify various safety properties using the tool NEVER2, a new framework that integrates learning and verification in a single easy-to-use package intended for practictioners rather than experts in formal methods and/or machine learning.

INTRODUCTION

Context and Motivation. Verification of neural networks (NNs) is currently a trending topic involving different areas of AI, including machine learning, constraint programming, heuristic search and automated reasoning. A relative recent survey [HKR+20] cites more than 200 papers, most of which have been published in the last few years, and more contributions are appearing with impressive progression — see, e.g., [DCJ+19], [KHI+19], [WWR+18], [NKR+18], [LM17], [WPW+18]. The reason of this growing interest is that, while the application of NNs in various domains [LBH15] have made them one of the most popular machine-learned models to date, concerns about their vulnerability to adversarial perturbations [SZS+14], [GSS15] have been accompanying them since their initial adoption, to the point of restraining their application in safety- and security-related contexts. Automated formal verification — see, e.g., [LNPT18b] for a survey — offers an effective answer to the problem of establishing correctness of a NN and opens the path to their adoption in applications where they are currently not popular. One such application is autonomous driving where different functions can indeed be learned from data. Examples include advanced functions such as automatic steering and automatic speeding/braking, and more mundane ones such as traction control, launch control and anti-lock system for brakes. While it is possible to learn these functions with NNs, it is not clear whether the rigorous certification procedures prescribed for car controllers can be passed by the learned controllers.

Objective. Our main research question is:

"Is it possible to leverage automated verification of neural networks in safety critical applications to improve confidence in the correctness of learned controllers?"

Clearly, automated verification together with standard testing techniques can provide reasonable confidence levels in a network only if the whole process that leads to a learned network is already bullet-proofed. In the following, we assume that a rigorous safety-by-design approach insures that scenario design, simulations, data-acquisitions and learning have been carried out so as to minimize errors in each phase and also in their integration. We do not expect to place blind trust in verification alone, but we expect NEVER2 and similar tools to be an essential ingredient of any safety conscious development process that involves learned controllers.

Contribution. This article is about learning and verification of a NN that replicates the function of an adaptive cruise control (ACC) similar to those used in real autonomous cars. The goal of the ACC is to maintain the vehicle at a speed set by the user and possibly adapt the speed considering other vehicles proceeding in front of it. We use our tool NEVER2 to learn and verify the controller, all in a single package. Our goal is to show how NEVER2 enables learning and verification for field practitioners who do not need to be experts in machine learning and/or verification. Our results show that NEVER2 is able to learn reasonable implementations of the ACC function (given the available data) and prove some interesting design requirements using abstraction techniques.

BACKGROUND

A. Basic Notation and Definitions

We denote n-dimensional *vectors* of real numbers $x \in \mathbb{R}^n$ — also *points* or *samples* — with lowercase letters like x, y, z. We write $x = (x_1, x_2, \ldots, x_n)$ to denote a vector with its *components* along the n coordinates. We denote $x \cdot y$ the *scalar product* of two vectors $x, y \in \mathbb{R}^n$ defined as $x \cdot y = \sum_{i=1}^{n} x_i y_i$. The *norm* $\|x\|$ of a vector is defined as $\|x\| = x \cdot x$. We denote sets of vectors $X \subseteq \mathbb{R}^n$ with uppercase letters like X, Y, Z. A set of vectors X is *bounded* if there exists $r \in \mathbb{R}, r > 0$ such that $\forall x, y \in X$ we have $d(x, y) < r$ where d is the *Euclidean norm* $d(x, y) = \|x - y\|$. A set X is *open* if for every point $x \in X$ there exists a positive real number ϵ_x such that a point $y \in \mathbb{R}^n$ belongs to X as long as $d(x, y) < \epsilon_x$. The complement of an open set is a *closed* set — intuitively, one that includes its boundary, whereas open sets do not; closed and bounded sets are *compact*. A set X is *convex* if for any two points $x, y \in X$ we have that also $z \in X$ $\forall z = (1 - \lambda)x + \lambda y$ with $\lambda \in [0, 1]$, i.e., all the points falling on the line passing through x and y are also

Stefano Demarchi, Dario Guidotti, Andrea Pitto and Armando Tacchella are with Università degli Studi di Genova, DIBRIS (Department of Informatics, Bioengineering, Robotics and Systems Engineering), Viale Causa 13, 16145 Genova. E-mail: stefano.demarchi@edu.unige.it, dario.guidotti@edu.unige.it, s3942710@studenti.unige.it (Andrea Pitto), armando.tacchella@unige.it. All authors contributed equally to the paper. The corresponding author is Armando Tacchella.

in X. Notice that the intersection of any family, either finite or infinite, of convex sets is convex, whereas the union, in general, is not. Given any non-empty set X, the smallest convex set $\mathcal{C}(\mathcal{X})$ containing X is the *convex hull of* X and it is defined as the intersection of all convex sets containing X. A *hyperplane* $H \subseteq \mathbb{R}^n$ can be defined as the set of points

$$H = \{x \in \mathbb{R}^n \mid a_1x_1 + a_2x_2 + \ldots + a_nx_n = b\}$$

where $a \in \mathbb{R}^n$, $b \in \mathbb{R}$ and at least one component of a is non-zero. Let $f(x) = a_1x_1 + a_2x_2 + \ldots + a_nx_n - b$ be the affine form defining H. The *closed half-spaces associated with* H are defined as

$$H_+(f) = \{x \in X \mid f(x) \geq 0\}$$
$$H_-(f) = \{x \in X \mid f(x) \leq 0\}$$

Notice that both $H_+(f)$ and $H_-(f)$ are convex. A *polyhedron* in $P \subseteq \mathbb{R}^n$ is a set of points defined as $P = \bigcap_{i=1}^{p} C_i$ where $p \in \mathbb{N}$ is a finite number of closed half-spaces C_i. A bounded polyhedron is a *polytope*: from the definition, it follows that polytopes are convex and compact in $\mathbb{R}^n$.

B. Neural Networks

Given a finite number p of functions $f_1 : \mathbb{R}^n \to \mathbb{R}^{n_1}, \ldots, f_p : \mathbb{R}^{n_{p-1}} \to \mathbb{R}^m$ — also called *layers* — we define a *feed forward neural network* as a function $\nu : \mathbb{R}^n \to \mathbb{R}^m$ obtained through the compositions of the layers, i.e., $\nu(x) = f_p(f_{p-1}(\ldots f_1(x)\ldots))$. The layer f_1 is called *input layer*, the layer f_p is called *output layer*, and the remaining layers are called *hidden*. For $x \in \mathbb{R}^n$, we consider only two types of layers:

- $f(x) = Ax + b$ with $A \in \mathbb{R}^{m\times n}$ and $b \in \mathbb{R}^m$ is an *affine layer* implementing the linear mapping $f : \mathbb{R}^n \to \mathbb{R}^m$;
- $f(x) = (\sigma_1(x_1), \ldots, \sigma_n(x_n))$ is a *functional layer* $f : \mathbb{R}^n \to \mathbb{R}^n$ consisting of n *activation functions* — also called *neurons*; usually $\sigma_i = \sigma$ for all $i \in [1, n]$, i.e., the function σ is applied componentwise to the vector x.

We consider the *ReLU* activation function defined as $\sigma(r) = max(0, r)$, which finds widespread adoption. For a neural network $\nu : \mathbb{R}^n \to \mathbb{R}^m$, the task of *classification* is about assigning to every input vector $x \in \mathbb{R}^n$ one out of m labels: an input x is assigned to a class k when $\nu(x)_k > \nu(x)_j$ for all $j \in [1, m]$ and $j \neq k$; the task of *regression* is about approximating a functional mapping from $\mathbb{R}^n$ to $\mathbb{R}^m$.

C. Verification task

Given a neural network $\nu : \mathbb{R}^n \to \mathbb{R}^m$ we wish to verify algorithmically that it complies to stated *post-conditions* on the output as long as it satisfies *pre-conditions* on the input. Without loss of generality[1], we assume that the input domain of ν is a bounded set $I \subset \mathbb{R}^n$. Therefore, the corresponding output domain is also a bounded set $O \subset \mathbb{R}^m$ because (i) affine transformations of bounded sets are still bounded sets, (ii) ReLU is a piecewise affine transformation of its input, (iii) the output of logistic functions is always bounded in the set $[0, 1]$, and the composition of bounded functions is still bounded. We require that the logic formulas defining pre- and post-conditions are interpretable as finite unions of bounded sets in the input and output domains. Formally, given p bounded sets $X_1, \ldots, X_p$ in I such that $\Pi = \bigcup_{i=1}^{p} X_i$ and s bounded sets $Y_1, \ldots, Y_s$ in O such that $\Sigma = \bigcup_{i=1}^{s} Y_i$, we wish to prove that

$$\forall x \in \Pi.\nu(x) \in \Sigma. \tag{1}$$

While this query cannot express some problems regarding neural networks, e.g., invertibility or equivalence [LNPT18b], it captures the general problem of testing robustness against *adversarial perturbations* [GSS15]. For example, given a network $\nu : I \to O$ with $I \subset \mathbb{R}^n$ and $O \subset \mathbb{R}^m$ performing a classification task, we have that separate regions of the input are assigned to one out of m labels by ν. Let us assume that region $X_j \in I$ is classified in the j-th class by ν. We define an *adversarial region* as a set $\hat{X}_j$ such that for all $\hat{x} \in \hat{X}$ there exists at least one $x \in X$ such that $d(x, \hat{x}) \leq \delta$ for some positive constant δ. The network ν is *robust* with respect to $\hat{X}_j \subseteq I$ if, for all $\hat{x} \in \hat{X}_j$, it is still the case that $\nu(x)_j > \nu(x)_i$ for all $i \in [1, m]$ with $i \neq j$. This can be stated in the notation of condition (1) by letting $\Pi = \{\hat{X}_j\}$ and $\Sigma = \{Y_j\}$ with $Y_j = \{y \in O \mid y_j \geq y_i + \epsilon, \forall i \in [1, n] \wedge i \neq j, \epsilon > 0\}$. Analogously, in a regression task we may ask that points that are sufficently close to any input vector in a set $X \subseteq I$ are also sufficiently close to the corresponding output vectors. To do this, given the positive constants δ and ϵ, we let $\hat{X} = \{\hat{x} \in I \mid \exists x.(x \in X \wedge d(\hat{x}, x) \leq \delta)\}$ and $\hat{Y} = \{\hat{y} \in O \mid \exists x.(x \in \hat{X} \wedge d(\hat{y}, \nu(x)) \leq \epsilon)\}$ to obtain $\Pi = \{\hat{X}\}$ and $\Sigma = \{\hat{Y}\}$. Notice that, given our definition, we consider adversarial regions and output images that may not be convex.

D. Case Study

Technically, an adaptive cruise control (ACC) is an autonomous driving function of level one[2], which controls the acceleration of the *ego car* — the car whereon the ACC is installed — along the longitudinal axis. An ACC has two competing objectives: keeping the ego car at the speed set by the user (*speed following mode*) and keeping a safe distance from the *exo car* in front (*car following mode*). The ACC that we consider has one output, i.e., the acceleration a suggested to the ego car in $m \cdot s^{-2}$, and 6 inputs:

- $v_p[m \cdot s^{-1}]$: the speed of the ego car.
- $S_r[m \cdot s^{-1}]$: the speed of the exo car relative to the ego car; when there is no exo car, this input has the value 0.
- $D[m]$: the actual distance between the ego car and the exo car; when there is no exo car or when the exo car is farther than $150m$ this input has the default value of $150m$.
- $TH[s]$: Minimum headway time; this is the minimum time gap between the exo car and the ego car: $TH \cdot v_p$ corresponds to D_s, i.e.,the *minimum safety distance*.
- $D_0[m]$: A safety margin to be added to the minimum safety distance D_s.

[1] Input domains must be bounded to enable implementation of neural networks on digital hardware; therefore, also data from physical processes, which are potentially ubounded, are normalized within small ranges in practical applications.

[2] "Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles", SAE Standards, J3016_202104.

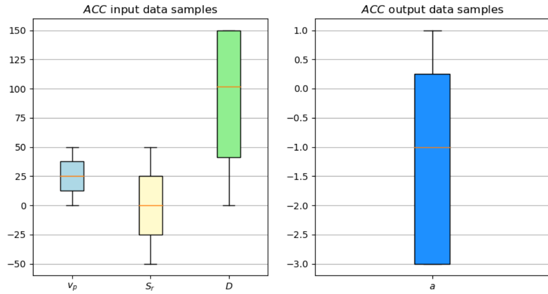

Fig. 1: Box plot for a million samples of the Adaptive Cruise Control data set ($TH = 1.5; D_0 = 5$)

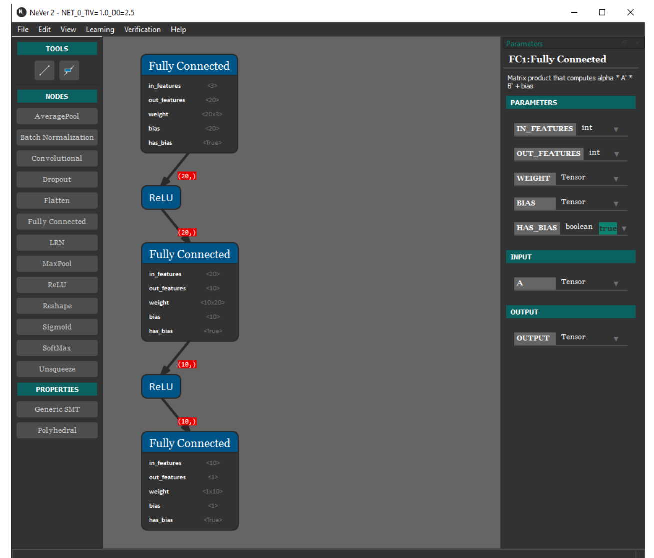

Fig. 2: NEVER2 representation of *Net0* architecture.

In production vehicles the ACC function is implemented using classical control laws. We view the production function — called ACC_o in the following — as a black-box whose behavior should be learned by a neural network.

LEARNING

Given the goal of learning ACC_o using a NN, we should generate several instances of input-output data using, e.g., a car simulator. Since a simulator was unavailable to us at the time of this writing, we generated the dataset to learn various NNs by drawing samples from uniform distributions over the input values of ACC_o, considering the following lower and upper bounds for v_p, v_r and D:

$$0 \leq v_p \leq 50 \quad -50 \leq v_r \leq 50 \quad 0 \leq D \leq 150 \qquad (2)$$

The values of TH and D_0 are kept fixed, and we obtain the corresponding output a by feeding ACC_o with the generated inputs. We generate 16 different data sets, each composed by a million samples, that feature 16 different combinations of TH and D_0, where $TH \in \{1, 1.5, 2, 2.5\}$, while $D_0 = \{2.5, 5, 7.5, 10\}$. Figure 1 shows the distributions of input and output samples using box plots in the case $TH = 1.5$ and $D_0 = 5$.

We tested three NN architectures comprised of affine and ReLU layers: we refer to them as *Net0*, *Net1* and *Net2* in the following. These NNs feature increasing complexity both in terms of the number of layers and in the amount of neurons per layer. An example is shown in Figure 2 for *Net0* on the canvas of NEVER2. The networks considered differs from each other only for the details of the hidden layers, which are the following:

- *Net0*: two affine layer of 20 and 10 neurons respectively, each followed by a ReLU layer;
- *Net1*: two affine layer of 50 and 40 neurons respectively, each followed by a ReLU layer;
- *Net2*: four affine layers of 20, 20, 20 and 10 neurons respectively, each followed by a ReLU layer;

The input of the network is in all the cases a three dimensional vector. All the networks present an output layers consisting of a linear layer of dimension 1 (without a following ReLU layer).

To learn the NNs we split the data sets in two parts, one for training and one for testing, with the ratio of 4:1. Our training phase lasts 100 epochs for each of the 16 data sets. We consider the *Adam* optimizer [KB14] and the *ReduceLROnPlateau* scheduler. For both our loss function and our precision metric we leveraged the *Mean Squared Error (MSE) loss*. We set batch sizes to 32 for training, validation, and test sets. In our setup, we dedicated 30% of the training set to the validation process. Concerning the optional parameters, we also set the learning rate to 0.01, the weight decay to 0.0001 and the training scheduler patience to 3, i.e., the number of consecutive epochs without loss decrease that triggers training procedure abortion. All the training is perfomed inside NEVER2 which, in turn, is based on the PYTORCH library. [3] For this reason, all the remaining parameters required by the learning algorithms are set to their default PYTORCH values.

VERIFICATION

To enable algorithmic verification of neural networks in NEVER2, we consider the abstract domain $\langle \mathbb{R}^n \rangle \subset 2^{\mathbb{R}^n}$ of polytopes defined in $\mathbb{R}^n$ to abstract (families of) bounded sets into (families of) polytopes. We provide corresponding abstractions for affine and functional layers to perform abstract computations and obtain consistent overapproximation of concrete networks.

Definition 1: (Abstraction) Given a bounded set $X \subset \mathbb{R}^n$, an abstraction is defined as a function $\alpha : 2^{\mathbb{R}^n} \to \langle \mathbb{R}^n \rangle$ that maps X to a polytope P such that $\mathcal{C}(\mathcal{X}) \subseteq \mathcal{P}$.

Intutively, the function α maps a bounded set X to a corresponding polytope in the abstract space such that the polytope always contains the convex hull of X. Depending on X, the enclosing polytope may not be unique. However, given the convex hull of any bounded set, it is always possible to find an enclosing polytope. As shown in [Zhe19], one could always start with an axis-aligned regular n simplex consisting of $n+1$ facets — e.g., the triangle in $\mathbb{R}^2$ and the tethraedron in $\mathbb{R}^3$ — and then refine the abstraction as needed by adding facets, i.e., adding half-spaces to make the abstraction more precise.

[3]https://pytorch.org

Definition 2: (Concretization) Given a polytope $P \in \langle\mathbb{R}^n\rangle$ a concretization is a function $\gamma : \langle\mathbb{R}^n\rangle \to 2^{\mathbb{R}^n}$ that maps P to the set of points cointained in it, i.e., $\gamma(P) = \{x \in \mathbb{R}^n \mid x \in P\}$.

Intutively, the function γ simply maps a polytope P to the corresponding (convex and compact) set in $\mathbb{R}^n$ comprising all the points contained in the polytope. As opposed to abstraction, the result of concretization is uniquely determined. We extend abstraction and concretization to finite families of sets and polytopes, respectively, as follows. Given a family of p bounded sets $\Pi = \{X_1, \ldots, X_p\}$, the abstraction of Π is a set of polytopes $\Sigma = \{P_1, \ldots, P_s\}$ such that $\alpha(X_i) \subseteq \bigcup_{i=1}^{s} P_i$ for all $i \in [1, p]$; when no ambiguity arises, we abuse notation and write $\alpha(\Pi)$ to denote the abstraction corresponding to the family Π. Given a family of s polytopes $\Sigma = \{P_1, \ldots, P_s\}$, the concretization of Σ is the union of the concretizations of its elements, i.e., $\bigcup_{i=1}^{s} \gamma(P_i)$; also in this case, we abuse notation and write $\gamma(\Sigma)$ to denote the concretization of a family of polytopes Σ.

Given our choice of abstract domain and a concrete network $\nu : I \to O$ with $I \subset \mathbb{R}^n$ and $O \subset \mathbb{R}^m$, we need to show how to obtain an *abstract neural network* $\tilde{\nu} : \langle I\rangle \to \langle O\rangle$ that provides a sound overapproximation of ν. To frame this concept, we introduce the notion of consistent abstraction.

Definition 3: (Consistent abstraction) Given a mapping $\nu : \mathbb{R}^n \to \mathbb{R}^m$, a mapping $\tilde{\nu} : \langle\mathbb{R}^n\rangle \to \langle\mathbb{R}^m\rangle$, abstraction function $\alpha : 2^{\mathbb{R}^n} \to \langle\mathbb{R}^m\rangle$ and concretization function $\gamma : \langle\mathbb{R}^m\rangle \to 2^{\mathbb{R}^m}$, the mapping $\tilde{\nu}$ is a consistent abstraction of ν over a set of inputs $X \subseteq I$ exactly when

$$\{\nu(x) \mid x \in X\} \subseteq \gamma(\tilde{\nu}(\alpha(X))) \tag{3}$$

The notion of consistent abstraction can be readily extended to families of sets as follows. The mapping $\tilde{\nu}$ is a consistent abstraction of ν over a family of sets of inputs $X_1 \ldots X_p$ exactly when

$$\{\nu(x) \mid x \in \cup_{i=1}^{p} X_i\} \subseteq \gamma(\tilde{\nu}(\alpha(X_1, \ldots, X_p))) \tag{4}$$

where we abuse notation and denote with $\tilde{\nu}(\cdot)$ the family $\{\tilde{\nu}(P_1), \ldots, \tilde{\nu}(P_s)\}$ with $\{P_1, \ldots, P_s\} = \alpha(X_1, \ldots X_p)$

To represent polytopes and define the computations performed by abstract layers in NEVER2, we resort to a specific subclass of *generalized star sets*, introduced in [BD17] and defined as follows — the notation is adapted from [TLM+19].

Definition 4: (Generalized star set) Given a *basis matrix* $V \in \mathbb{R}^{n\times m}$ obtained arranging a set of m *basis vectors* $\{v_1, \ldots v_m\}$ in columns , a point $c \in \mathbb{R}^n$ called *center* and a *predicate* $R : \mathbb{R}^m \to \{\top, \bot\}$, a generalized star set is a tuple $\Theta = (c, V, R)$. The set of points represented by the generalized star set is given by

$$[\![\Theta]\!] \equiv \{z \in \mathbb{R}^n \mid z = Vx + c \text{ such that } R(x_1, \ldots, x_m) = \top\} \tag{5}$$

In the following we denote $[\![\Theta]\!]$ also as Θ. Depending on the choice of R, generalized star sets can represent different kinds of sets, but in NEVER2 we consider only those such that $R(x) := Cx \leq d$, where $C \in \mathbb{R}^{p\times m}$ and $d \in \mathbb{R}^p$ for $p \geq 1$, i.e., R is a conjunction of p linear constraints as in [TLM+19]; we further require that the set $Y = \{y \in \mathbb{R}^m \mid Cy \leq d\}$ is bounded.

Given a generalized star set $\Theta = (c, V, R)$ such that $R(x) := Cx \leq d$ with $C \in \mathbb{R}^{p\times m}$ and $d \in \mathbb{R}^p$, if the set $Y = \{y \in \mathbb{R}^m \mid Cy \leq d\}$ is bounded, then the set of points represented by Θ is a polytope in $\mathbb{R}^n$, i.e., $\Theta \in \langle\mathbb{R}^n\rangle$. In the following, we refer to generalized star sets obeying our restrictions simply as *stars*.

The simplest abstract layer to obtain is the one abstracting affine transformations. As we have already mentioned, affine transformations of polytopes are still polytopes, so we just need to define how to apply an affine transformation to a star — the definition is adapted from [TLM+19].

Definition 5: (Abstract affine mapping) Given a star set $\Theta = (c, V, R)$ and an affine mapping $f : R^n \to R^m$ with $f = Ax + b$, the abstract affine mapping $\tilde{f} : \langle R^n\rangle \to \langle R^m\rangle$ of f is defined as $\tilde{f}(\Theta) = (\hat{c}, \hat{V}, R)$ where

$$\hat{c} = Ac + b \qquad \hat{V} = AV$$

Intuitively, the center and the basis vectors of the input star Θ are affected by the transformation of f, while the predicates remain the same. Given an affine mapping $f : \mathbb{R}^n \to \mathbb{R}^m$, the corresponding abstract mapping $\tilde{f} : \langle\mathbb{R}^n\rangle \to \langle\mathbb{R}^m\rangle$ provides a consistent abstraction over any bounded set $X \subset \mathbb{R}^n$, i.e., $\{f(x) \mid x \in X\} \subseteq \gamma(\tilde{f}(\alpha(X)))$ for all $X \subset \mathbb{R}^n$. we observe that the set $\alpha(X)$ is any polytope P such that $P \supseteq \mathcal{C}(\mathcal{X})$ — equality holds only when X is already a polytope, and thus $X \equiv \mathcal{C}(\mathcal{X}) \equiv \mathcal{P}$. Let $\Theta_P = (c_P, V_P, R_P)$ be the star corresponding to P defined as

$$c_P = 0^n \qquad V_P = I^n \qquad R_P = C_P x + d_P \leq 0$$

where 0^n is the n-dimensional zero vector, and I^n is the $n \times n$ identity matrix — the columns of I^n correspond to the standard orthonormal basis $e_1, \ldots, e_n$ of $\mathbb{R}^n$, i.e., $\|e_i\| = 1$ and $e_i \cdot e_j = 0$ for all $i \neq j$ with $i, j \in [1, n]$; the matrix $C_P \in \mathbb{R}^{q\times n}$ and the vector $d_P \in \mathbb{R}^q$ collect the parameters defining q half-spaces whose intersection corresponds to P. Given our choice of c and V, it is thus obvious that $\Theta_P \equiv P$. Recall that $f = Ax + b$ with $A \in \mathbb{R}^{m\times n}$ and $b \in \mathbb{R}^m$; from definition (5) we have that $\tilde{f}(\Theta_P) = \hat{\Theta}_P$ with $\hat{\Theta}_P = (\hat{c}_P, \hat{V}_P, R_P)$ and

$$\hat{c}_P = A0^n + b = b \qquad \hat{V}_P = AI^n = A$$

The concretization of $\hat{\Theta}_P$ is just the set of points contained in $\hat{\Theta}_P$ defined as

$$\gamma(\hat{\Theta}_P) = \{z \in \mathbb{R}^m \mid z = Ax + b \text{ such that } C_p x \leq d_P\} \tag{6}$$

Now it remains to show that $\{f(x) \mid x \in X\} \subseteq \gamma(\hat{\Theta}_P)$. This follows from the fact that, for a generic $y \in \{f(x) \mid x \in X\}$ there must exists $x \in X$ such that $y = Ax + b$; since x satisfies $C_p x \leq d_P$ by construction of P, it is also the case that $y \in \gamma(\hat{\Theta}_P)$ by definition (6).

Algorithm 1 defines the abstract mapping of a functional layer with n ReLU activation functions in NEVER2. The function COMPUTE_LAYER takes as input an indexed list of N stars $\Theta_1, \ldots, \Theta_N$ and an indexed list of n positive integers called *refinement levels*. For each neuron, the refinement level tunes the grain of the abstraction: level 0 corresponds to the coarsest abstraction that we consider — the

Algorithm 1 Abstraction of the ReLU activation function.

1: **function** COMPUTE_LAYER(*input* = $[\Theta_1, \ldots, \Theta_N]$, *refine* = $[r_1, \ldots, r_n]$)
2:     *output* = [ ]
3:     **for** $i = 1 : N$ **do**
4:         *stars* = $[\Theta_i]$
5:         **for** $j = 1 : n$ **do** *stars* = COMPUTE_RELU(*stars*, j, *refine*$[j]$, n)
6:         **end for**
7:         APPEND(*output*, *stars*)
8:     **end for**
9:     **return** *output*
10: **end function**

11: **function** COMPUTE_RELU(*input* = $[\Gamma_1, \ldots, \Gamma_M]$, j, *level*, n)
12:     *output* = [ ]
13:     **for** $k = 1 : M$ **do**
14:         (lb_j, ub_j) = GET_BOUNDS(*input*$[k]$, j)
15:         $M = [e_1 \ldots e_{j-1}\ 0\ e_{j+1} \ldots e_n]$
16:         **if** $lb_j \geq 0$ **then** S = *input*$[k]$
17:         **else if** $ub_j \leq 0$ **then** $S = M$ * *input*$[k]$
18:         **else**
19:             **if** *level* > 0 **then**
20:                 Θ_{low} = *input*$[k] \wedge z[j] < 0$; Θ_{upp} = *input*$[k] \wedge z[j] \geq 0$
21:                 $S = [M * \Theta_{low}, \Theta_{upp}]$
22:             **else**
23:                 $(c, V, Cx \leq d)$ = *input*$[j]$
24:                 $C_1 = [0\ 0 \ldots\ -1] \in \mathbb{R}^{1 \times m+1}$, $d_1 = 0$
25:                 $C_2 = [V[j,:]\ -1] \in \mathbb{R}^{1 \times m+1}$, $d_2 = -c_k[j]$
26:                 $C_3 = [\frac{-ub_j}{ub_j - lb_j} \cdot V[j,:]\ -1] \in \mathbb{R}^{1 \times m+1}$, $d_3 = \frac{ub_j}{ub_j - lb_j}(c[j] - lb_j)$
27:                 $C_0 = [C\ 0^{m \times 1}]$, $d_0 = d$
28:                 $\hat{C} = [C_0;\ C_1;\ C_2;\ C_3]$, $\hat{d} = [d_0;\ d_1;\ d_2;\ d_3]$
29:                 $\hat{V} = MV$, $\hat{V} = [\hat{V}\ e_j]$
30:                 $S = (Mc, \hat{V}, \hat{C}\hat{x} \leq \hat{d})$
31:             **end if**
32:         **end if**
33:         APPEND(*output*, S)
34:     **end for**
35:     **return** *output*
36: **end function**

greater the level, the finer the abstraction grain. In the case of ReLUs, all non-zero levels map to the same (precise) refinement, i.e., a piecewise affine mapping. Notice that, since each neuron features its own refinement level, algorithm 1 controls abstraction down to the single neuron, enabling the computation of levels with mixed degrees of abstraction. The output of function COMPUTE_LAYER is still an indexed list of stars, that can be obtained by independently processing the stars in the input list. For this reason, the **for** loop starting at line 3 can be parallelized to speed up actual implementations. Given a single input star $\Theta_i \in \langle R^n \rangle$, each of the n dimensions is processed in turn by the **for** loop starting at line 5 and involing the function COMPUTE_RELU. Notice that the stars obtained processing the j-th dimension are feeded again to COMPUTE_RELU in order to process the $j+1$-th dimension. For each star given as input, the function COMPUTE_RELU first computes the lower and upper bounds of the star along the j-th dimension by solving a linear-programming problem — function GET_BOUNDS at line 11. Independently from the abstraction level, if $lb_j \geq 0$ then the ReLU acts as an identity function (line 13), whereas if $lb_j \geq 0$ then the j-th dimension is zeroed (line 14). The operator "star" takes a matrix M, a star $\Gamma = (c, V, R)$ and returns the star (Mc, MV, R).

The linear-programming problem we need to solve in the GET_BOUNDS solver can be formalized as follows:

$$(min/max)\ z_j = \mathbf{V}[j,:]\mathbf{x} + c[j]$$
$$with \quad \mathbf{Cx} \leq \mathbf{d}$$

The problem must be solved as minimization and maximization to provide the lower bound and the upper bound respectively. It should be noted that the complexity of the problem increases with the number of variables of the predicate of the star of interest. As consequence the computational complexity of GET_BOUNDS increases as over-approximation increases, whereas for concrete stars the complexity remains the same.

Given a ReLU mapping $f : \mathbb{R}^n \rightarrow \mathbb{R}^n$, the corresponding abstract mapping $\tilde{f} : \langle \mathbb{R}^n \rangle \rightarrow \langle \mathbb{R}^n \rangle$ defined in algorithm 1 provides a consistent abstraction over any bounded set $X \subset \mathbb{R}^n$, i.e., $\{f(x) \mid x \in X\} \subseteq \gamma(\tilde{f}(\alpha(X)))$ for all $X \subset \mathbb{R}^n$.

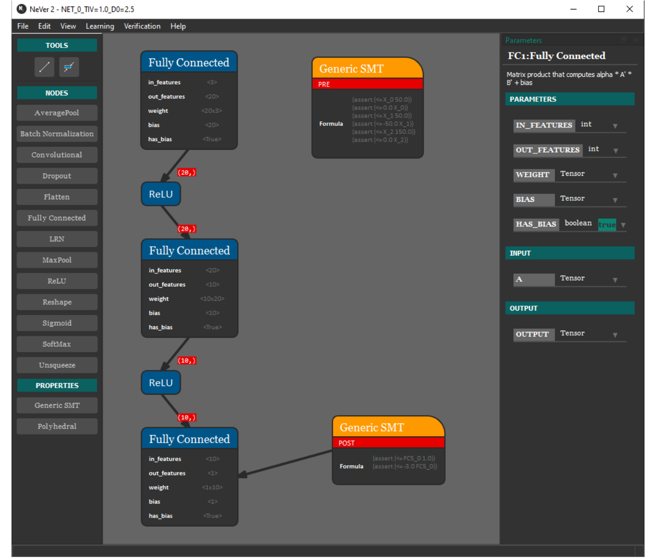

Fig. 3: Representation of the *OutBounds* property in NEVER2. Detached property blocks are treated as input pre-conditions, while property blocks linked to the NN are the output post-conditions.

EXPERIMENTS

We consider three properties for the ACC case study, and we verify them in NEVER2 with different NNs. The first property that we define, called *OutBounds* in the following, simply checks that the output acceleration does not exceed the bounds of the ACC_o function. Stated formally, this amounts to have NEVER2 check that, given the preconditions

$$\begin{aligned} 0 &\leq v_p \leq 50 \\ -50 &\leq v_r \leq 50 \\ 0 &\leq D \leq 150 \end{aligned} \tag{7}$$

the output a satisfies the postcondition

$$-3 \leq a \leq 1. \tag{8}$$

Figure 3 shows NEVER2 canvas with the additional property specification.

The second property we consider is called *Near0*, and it is aimed at making sure that the ACC system does not output

TABLE I: NEVER2 results for the ACC data set with $TH = 1$ and $D_0 = 5$, with $\epsilon = 0$ (left) and $\epsilon = 20$ (right). CPU time is in seconds rounded to the third decimal place. The best setting for each network and property is highlighted in boldface.

$TH = 1,5$ — $D_0 = 5$ — $\epsilon = 0$				
Network	**Property**	**Setting**	**Result**	**CPU Time**
Net0	*OutBounds*	over-approx	True	5.139
		mixed	True	**5.055**
		mixed2	True	5.112
		complete	True	6.273
	Near0	over-approx	False	5.666
		mixed	False	5.251
		mixed2	False	**5.203**
		complete	False	6.319
	Far0	over-approx	False	5.078
		mixed	False	**4.986**
		mixed2	False	5.139
		complete	False	5.186
Net1	*OutBounds*	over-approx	True	**5.931**
		mixed	True	6.662
		mixed2	True	7.309
		complete	True	51.683
	Near0	over-approx	False	**5.906**
		mixed	False	6.676
		mixed2	False	8.071
		complete	False	50.469
	Far0	over-approx	False	**5.709**
		mixed	False	5.888
		mixed2	False	6.301
		complete	False	13.041
Net2	*OutBounds*	over-approx	True	**9.525**
		mixed	True	10.482
		mixed2	True	12.525
		complete	True	26.958
	Near0	over-approx	False	**9.515**
		mixed	False	10.292
		mixed2	False	13.636
		complete	False	24.496
	Far0	over-approx	False	**9.753**
		mixed	False	9.944
		mixed2	False	12.148
		complete	False	13.27

$TH = 1.5$ — $D_0 = 5$ — $\epsilon = 20$				
Network	**Property**	**Setting**	**Result**	**CPU Time**
Net0	*OutBounds*	over-approx	True	**5.037**
		mixed	True	5.063
		mixed2	True	4.996
		complete	True	6.203
	Near0	over-approx	False	**5.034**
		mixed	False	5.101
		mixed2	False	4.965
		complete	False	5.345
	Far0	over-approx	False	5.008
		mixed	True	**5.016**
		mixed2	True	5.068
		complete	True	5.62
Net1	*OutBounds*	over-approx	True	**5.948**
		mixed	True	6.934
		mixed2	True	7.232
		complete	True	52.318
	Near0	over-approx	False	**5.436**
		mixed	False	5.797
		mixed2	False	5.955
		complete	False	7.667
	Far0	over-approx	False	**5.344**
		mixed	False	5.776
		mixed2	False	6.226
		complete	False	8.212
Net2	*OutBounds*	over-approx	True	**9.532**
		mixed	True	10.149
		mixed2	True	12.065
		complete	True	26.794
	Near0	over-approx	False	9.379
		mixed	False	9.872
		mixed2	False	11.653
		complete	True	**10.696**
	Far0	over-approx	False	**9.453**x
		mixed	False	9.848
		mixed2	False	11.558
		complete	False	10.854

positive accelerations when the vehicle ahead is too close. We frame this concept via the precondition

$$\begin{aligned} 0 \leq v_p \leq 50 \\ -50 \leq v_r \leq 50 \\ 0 \leq D \leq 150 \\ TH \cdot v_r + D_0 \geq D + \epsilon \end{aligned} \tag{9}$$

where $\epsilon \in \mathbb{R}^+$ is a positive tolerance value in the last inequality. Notice that the input bounds are the same as *Outbound*. The last inequality stems from the fact that $TH \cdot v_r$ is the safety distance required to stop the ego car in time if the exo car brakes, and D_0 is a buffer value which, like TH, is constant for each data set. The corresponding output postcondition for *Near0* is

$$-3 \leq a \leq 0. \tag{10}$$

Intuitively, we do not want the network to output positive accelerations in this case.

Finally, the last property we consider is *Far0*, which is symmetrical with respect to *Near0*. The precondition is

$$\begin{aligned} 0 \leq v_p \leq 50 \\ -50 \leq v_r \leq 50 \\ 0 \leq D \leq 150 \\ TH \cdot v_r + D_0 \leq D - \epsilon \end{aligned} \tag{11}$$

where $\epsilon \in \mathbb{R}^+$ is still a tolerance value and the input bounds coincide with *OutBounds* and *Near0* properties. In this case, we want to verify that when the ego car is too far from the exo car (or there is no vehicle ahead at all), the NN does not suggests negative accelerations. The output postcondition is

$$0 \leq a \leq 1. \tag{12}$$

In addition to the properties themselves, we also define different configuration for NEVER2 verification algorithms. In particular we consider 3 settings: *over-approximated*, *mixed*, and *complete*. The over-approximated setting corresponds to running algorithm 1 with *level* greater than zero whereas the complete setting amounts to choose *level* $= 0$. In the former case the output image of the NN computed by NEVER2 given the input preconditions is an overapproximation of the concrete one. In this case, checking whether the output image satisfies the postcondition gives us a sufficient condition only, i.e., if the inequality holds the NN is safe with respect to that property. On the other hand, if the inequality is not verified, the NN may still be safe and the check may have failed because of the loss of precision in the abstraction process. In the complete setting, on the other hand, NEVER2 computes the actual output image of the network: if the inequality in the postcondition does not hold, we are sure that the NN in not safe. However, the complete setting in algorithm 1 potentially causes the exponential blow-up in the number of stars generated, and thus the computation might simply not be feasible. The mixed setting

strikes a trade-off between complete and over-approximated setting: using an heuristic detailed in [GPT21], NEVER2 tries to concretize the least number of stars that enable proving the property without blowing the computation time. In our experiments, we consider two different sub-settings for mixed, called *mixed* and *mixed2* which differ in the number of neurons to refine, either 1 or 2, respectively.

We run our tests on a workstation featuring two Intel Xeon Gold 6234 CPU, three NVIDIA Quadro RTX 6000/8000 GPUs (with CUDA enabled), and 125.6 GiB of RAM running Ubuntu 20.04.03 LTS. For the sake of brevity, we are only going to report here a fraction of the experiments we ran in Table I for the data set with $TH = 1.5$ and $D_0 = 5$, considering $\epsilon = 0$ and $\epsilon = 20$. The results we show here are consistent with the results obtained on other data sets that we do not report. In particular, looking at Table I we can observe that:

- All the properties can be checked on all the networks in reasonable time by NEVER2: less than one minute of CPU time is required independently from the network architecture and the specific setting considered.
- The complete setting is the most expensive in computational terms; given the considerations above this should come at no surprise, but in one case, namely *Net2* on property *Near0*, the complete setting is able to prevail over the others, i.e., it certifies that the property is true; indeed mixed and over-approximated settings (shortened as over-approx in Table I) take less time, but state that the property is false because they do not manage to reach enough precision to state the correct result.
- The over-approximated setting is often faster than the other ones: 6 out of 9 cases for $\epsilon = 0$ and 7 out of 9 cases for $\epsilon = 20$; however it must be noted that its results are definite only when the property is true: 3 out of 9 cases for both values of ϵ and always for the (simplest) property *OutBounds*.
- The mixed setting is at times faster than the over-approximated one, but only in one case, namely property *Far0* on *Net0* it is able to provide a definite answer while outperforming both the complete and over-approximated settings.

Overall we can conclude that while further research is needed to improve on the capability of NEVER2 to provide definite answers with faster techniques involving over-approximation, still the tool is able to check a number of interesting properties in networks involving hundreds of neurons in a relatively small amount of CPU time. We view this as a positive result and an enabler for preliminary testing of NEVER2 at industrial settings featuring networks of comparable size to our ACC case study.

CONCLUSIONS

In this article we developed a concrete example of how our system NEVER2 can learn and verify NNs. The application we consider is, to the best of our knowledge, one of the examples of formal verification for NNs which are closest to industrial application. We have shown convincing experimental evidence that it is possible to learn an adaptive cruise control function and verify some interesting properties, all in a single package that supports the process through an easy to use graphical user interface. In future works, we intend to deepen our research and find more applications that require NNs to be learned and verified, possibly with more complex architectures to stress NEVER2 capabilities.

REFERENCES

[BD17] Stanley Bak and Parasara Sridhar Duggirala. Simulation-equivalent reachability of large linear systems with inputs. In *International Conference on Computer Aided Verification*, pages 401–420. Springer, 2017.

[DCJ+19] Souradeep Dutta, Xin Chen, Susmit Jha, Sriram Sankaranarayanan, and Ashish Tiwari. Sherlock - A tool for verification of neural network feedback systems: demo abstract. In *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control, HSCC*, pages 262–263, 2019.

[GPT21] Dario Guidotti, Luca Pulina, and Armando Tacchella. pynever: A framework for learning and verification of neural networks. In *Automated Technology for Verification and Analysis - 19th International Symposium, ATVA*, pages 357–363, 2021.

[GSS15] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. In *3rd International Conference on Learning Representations, ICLR (Poster)*, 2015.

[HKR+20] Xiaowei Huang, Daniel Kroening, Wenjie Ruan, James Sharp, Youcheng Sun, Emese Thamo, Min Wu, and Xinping Yi. A survey of safety and trustworthiness of deep neural networks: Verification, testing, adversarial attack and defence, and interpretability. *Computer Science Review*, 37:100270, 2020.

[KB14] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[KHI+19] Guy Katz, Derek A. Huang, Duligur Ibeling, Kyle Julian, Christopher Lazarus, Rachel Lim, Parth Shah, Shantanu Thakoor, Haoze Wu, Aleksandar Zeljic, David L. Dill, Mykel J. Kochenderfer, and Clark W. Barrett. The marabou framework for verification and analysis of deep neural networks. In *Computer Aided Verification - 31st International Conference, CAV*, pages 443–452, 2019.

[LBH15] Yann LeCun, Yoshua Bengio, and Geoffrey E. Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[LM17] Alessio Lomuscio and Lalit Maganti. An approach to reachability analysis for feed-forward relu neural networks. *CoRR*, abs/1706.07351, 2017.

[LNPT18a] Francesco Leofante, Nina Narodytska, Luca Pulina, and Armando Tacchella. Automated Verification of Neural Networks: Advances, Challenges and Perspectives. *arXiv e-prints*, page arXiv:1805.09938, May 2018.

[LNPT18b] Francesco Leofante, Nina Narodytska, Luca Pulina, and Armando Tacchella. Automated verification of neural networks: Advances, challenges and perspectives. *CoRR*, abs/1805.09938, 2018.

[NKR+18] Nina Narodytska, Shiva Prasad Kasiviswanathan, Leonid Ryzhyk, Mooly Sagiv, and Toby Walsh. Verifying properties of binarized deep neural networks. In *Thirty-Second AAAI Conference on Artificial Intelligence (AAAI-18)*, pages 6615–6624, 2018.

[SZS+14] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus. Intriguing properties of neural networks. In *2nd International Conference on Learning Representations, ICLR*, 2014.

[TLM+19] Hoang-Dung Tran, Diago Manzanas Lopez, Patrick Musau, Xiaodong Yang, Luan Viet Nguyen, Weiming Xiang, and Taylor T Johnson. Star-based reachability analysis of deep neural networks. In *International Symposium on Formal Methods*, pages 670–686. Springer, 2019.

[WPW+18] Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. Efficient formal safety analysis of neural networks. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada*, pages 6369–6379, 2018.

[WWR+18] Min Wu, Matthew Wicker, Wenjie Ruan, Xiaowei Huang, and Marta Kwiatkowska. A game-based approximate verification of deep neural networks with provable guarantees. *CoRR*, abs/1807.03571, 2018.

[Zhe19] Yu Zheng. Computing bounding polytopes of a compact set and related problems in n-dimensional space. *Computer-Aided Design*, 109:22–32, 2019.