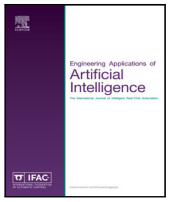
Engineering Applications of Artificial Intelligence 137 (2024) 109238

Contents lists available at ScienceDirect

Engineering Applications of Artificial Intelligence

journal homepage: www.elsevier.com/locate/engappai



Research Paper

Neural networks in closed-loop systems: Verification using interval arithmetic and formal prover

Federico Rossi *, Cinzia Bernardeschi, Marco Cococcioni

Department of Information Engineering, University of Pisa, Via G. Caruso 16, Pisa, 56127, PI, Italy

ARTICLE INFO

Keywords:
Cyber–physical systems
Neural networks
Closed-loop control systems
Formal verification
Interval arithmetic

ABSTRACT

Machine Learning approaches have been successfully used for the creation of high-performance control components of cyber–physical systems, where the control dynamics result from the combination of many subsystems. However, these approaches may lack the trustworthiness required to guarantee their reliable application in a safety-critical context. In this paper, we propose a combination of interval arithmetic and theorem-proving verification techniques to analyze safety properties in closed-loop systems that embed neural network components. We show the application of the proposed approach to a model-predictive controller for autonomous driving comparing the neural network verification performance with other existing tools. The results show that open-loop neural network verification through interval arithmetic can outperform existing approaches proving properties with a smaller time overhead. Furthermore, we demonstrate the capability of combining the two approaches to construct a formal model of the network in higher-order logic of the controlled system in a closed-loop.

1. Introduction

Cyber–physical systems (CPS) (Alur, 2015) are a combination of computational algorithms and physical components that interact as closed-loop systems. In the current landscape of CPS neural network integration is becoming more and more common (Rathore et al., 2021), changing the capabilities of these systems in various fields including smart grids, smart manufacturing (Huang et al., 2021), autonomous vehicles (AVs) (Cococcioni et al., 2021; Rossi et al., 2024), industrial automation (Bernardeschi et al., 2023), medical, and more.

A widespread approach is to apply neural computing to the controller part of CPSs, increasing modeling and adaptation capabilities (Antsaklis, 1990; Emami et al., 2022; Jin et al., 2018). Neural networks are also applied for the generation of surrogated components of CPSs. For example, in Song et al. (2024) authors proposed a neural-network based, fault-tolerant control design for a quadrotor unmanned aerial vehicle. Furthermore, in Song et al. (2023) authors proposed a quantization approach to neural network control for non-linear systems. Another potential application of machine learning to time-series analysis is predictive maintenance for machinery (Putnik et al., 2021). Furthermore, by examining real-time data from the physical system's sensors, recurrent neural networks (RNN) and long short-term memory (LSTM) techniques can be utilized to carry out anomaly detection (Jeffrey et al., 2023). The authors of Vereno et al. (2023) used reinforcement learning in conjunction with the co-simulation of a smart grid system; they used separate architectures for the power grid integration and the AI system. The use of neural network components in safety-critical CPS, however, raises concerns about their dependability and safety.

Formal verification techniques are gaining popularity as a means of ensuring neural network trustworthiness in CPS (Urban and Miné, 2021; Alur, 2011). Formal verification offers a logical and quantitative way to assess the accuracy and adherence to requirements of complex systems, providing a level of assurance that is essential for applications where errors might have catastrophic consequences.

Interval arithmetic (Hickey et al., 2001; Althoff, 2015; Kochdumper et al., 2023) provides a systematic approach for handling uncertainties in computations by representing quantities as intervals bounded by their lower and upper bounds. In the context of neural network verification, interval arithmetic offers an efficient way of bounding the output of a neural network given a range of possible inputs. In interval arithmetic-based verification, the goal is to compute guaranteed output intervals for a neural network given intervals representing the possible range of inputs.

In this work, we propose an approach which takes as input the trained model, a fully connected feed-forward neural network resulting

* Corresponding author.
E-mail address: federico.rossi@ing.unipi.it (F. Rossi).

[1] https://pvs.csl.sri.com/.

[2] https://pytorch.org/.

https://doi.org/10.1016/j.engappai.2024.109238
Received 16 April 2024; Received in revised form 14 July 2024; Accepted 29 August 2024
Available online 3 September 2024

Table 1
Summary of related works, grouped by approach to verification.

Work	Activation function	Open/Closed loop	Approach
Katz et al. (2017), Pulina and Tacchella (2010), Ehlers (2017), Katz et al. (2019) and Narodytska et al. (2018)	ReLU	Open	SMT
Ivanov et al. (2019, 2021)	Sigmoid	Open	Hybrid automata
Huang et al. (2019), Fan et al. (2020) and Xiang et al. (2018)	Any	Open	Monte Carlo simulation
Aleksandrov and Völlinger (2023) and Rossi et al. (2024)	Any	Open	Formal prover
Lopez et al. (2023b), Althoff (2015), Kochdumper et al. (2023), Bak and Tran (2022), Lopez et al. (2023a,c) and Bak (2021)	Any	Open/Closed	Reachability +Interval sets
Wang et al. (2021), Xu et al. (2020), Kotha et al. (2024), Zhang et al. (2022) and Shi et al. (2024)	Any	Open	Bound propagation

from well-known training algorithms, and automatically produces its formal representation using interval arithmetic. This allows the verification of user-defined properties on the closed-loop system controlled by the neural network with a theorem prover such as Prototype Verification System (PVS) (Owre et al., 1996; Masci and Muñoz, 2019).[1] The formal verification of the closed-loop system is a semi-automatic process that combines automatic procedures of the theorem prover with user knowledge. We show how our approach can be seamlessly integrated inside the PyTorch (Paszke et al., 2019; Ansel et al., 2024)[2] deep learning framework and how the timing overhead in verifying the neural network properties compared to other state-of-the-art verification tools is sensibly smaller in many cases. Furthermore, we show how the proved bounds of the neural network output can be used to provide a high-level abstraction of the neural network model inside the PVS prover to verify closed-loop properties of dynamic systems controlled by the neural network.

The paper is organized as follows: (i) Section 2 deepen on the current state of the art of formal verification and modeling of neural networks both in open and closed loop, (ii) Section 3 briefly summarizes the basic concepts of the [nome esteso messo prima sostituire - Prototype Verification System (PVS) - con - PVS] prover, neural networks and PyTorch neural network framework, (iii) Section 4 details on the use of interval arithmetic for analytically prove neural networks bounds, (iv) Section 5 shows the formalization in higher-order logic of neural networks, (v) Section 6 introduces and explains the approach of formalizing and modeling neural network properties in closed-loop systems and its integration within PyTorch, (vi) Section 7 shows an example use-case to validate the approach applied to an adaptive cruise control system where the core model predictive controller is substituted by a neural network controller, (vii) Section 8 concludes and states possible further work.

2. Related works

Several studies have focused on the formal or analytical validation of neural networks using a variety of theories, most of which use the feed-forward neural network's rectified linear unit (ReLU) activation function. Table 1 summarizes the different works grouped by the base approach/technology used for verification.

Satisfiability modulo theory (SMT) based tools. Authors of Katz et al. (2017) provide an effective technique for using SMT to validate relu-based deep (fully connected) neural networks. Authors of Pulina and Tacchella (2010) described a similar SMT method for multilayer perceptron verification, abstracting sigmoid-based feed-forward neural networks with a piece-wise linear activation function. Authors again used ReLU-like activation functions in Ehlers (2017) to propose an SMT approach to validate properties of piece-wise linear feed-forward neural networks. Similarly, based once more on ReLU activation functions, authors in Katz et al. (2019) introduced Marabou, an evolution of Katz et al. (2017) to extend verification capabilities to the convolutional and pooling layers. Other methods, such as Narodytska et al. (2018), use SAT solvers and Propositional Satisfiability Solvers to validate neural network properties by taking advantage of the boolean satisfiability theory.

Hybrid automata. Different classes of activation functions and neural networks, such as convolutional layers or non-linear activation functions, were the focus of other investigations. Specifically, the authors of Ivanov et al. (2019, 2021) formalized sigmoid-based neural networks using hybrid automata.

Simulation-based. In Huang et al. (2019), Fan et al. (2020) and Xiang et al. (2018), the authors use a Monte-Carlo-like simulation technique to empirically analyze the reachability set of a neural network. The authors of Lopez et al. (2023b) present a geometrical polyhedron-based verification technique that can only be used in conjunction with the ReLU activation function to assess the reachability set of a neural network across several deep learning models. Authors present a method for formal verification of neural networks and "learning enabled" components in a closed control loop system in Bak and Tran (2022) and Lopez et al. (2023a,c).

Formal provers. The Coq prover[3] was utilized by the authors of Aleksandrov and Völlinger (2023) to offer methods for feed-forward neural network verification. The PVS prover was used in Rossi et al. (2024) to prove neural network properties.

Interval-sets and bound propagation. Althoff (2015) and Kochdumper et al. (2023) suggest CORA, a tool for continuous reachability analysis using zonotopes, Taylor models, and interval arithmetic that can be used with both dynamic systems and neural networks. Authors of Bak (2021) propose nnenum, an high-performance, state-of-the-art verification tool for ReLU-based neural networks based on zonotopes overapproximations. In Wang et al. (2021), Xu et al. (2020), Kotha et al. (2024), Zhang et al. (2022) and Shi et al. (2024) authors proposed a bound propagation mechanism to verify neural network robustness against perturbations of the inputs. In particular (Wang et al., 2021) is one of the most representative works of bound-propagation for neural network verification, ranking as the winner in the International Verification of Neural Components (VNNCOMP) in 2021, 2022 and 2023.

In this work, we aim to combine the interval arithmetic approach and the higher abstraction of formal models, providing a thorough approach to prove safety properties in closed-loop systems. This approach encompasses the use of exponential-based activation functions, such as the hyperbolic tangent and the sigmoid.

3. Background

This Section provides details on the PVS prover and neural network notation that will be extensively used in the remainder of this work. Although our approach is not limited to fully connected neural networks, in this work we will focus on this class of network, more suitable for regression problems used to create neural network controllers. Details for interval arithmetic are shown in Section 4.

[3] https://coq.inria.fr/.

3.1. PVS language

PVS (Owre et al., 1996) is a mechanized environment for formal specification and verification. A PVS *specification* is a combination of one or more *theories*, where a theory is a set of formulas, variable declarations, and function declarations. The PVS language provides a large set of base types for variables, including naturals, integers, reals, booleans, and their operations, each defined in the fundamental library *prelude*, implicitly imported in every PVS theory. Complex and advanced data types, such as the matrices used in this work, have been provided by the Nasalib extensions (Dutertre, 1996). Function declarations are in the form `foo(arg: T1): T2`, where *foo* is the name, *arg* is the argument of type *T1*, and *T2* is the type returned by the function. The formulas are the sentences (named *THEOREM* or *LEMMA*) that users should prove to guarantee that a certain property of the system described in the specification holds starting from some valid *AXIOMS*.

The proof system of PVS is based on the sequent calculus (Smullyan, 1968). The sequent calculus works on expressions, called sequents, of this form:

$$A_1 \wedge A_2 \wedge \cdots \wedge A_n \vdash B_1 \vee B_2 \vee \cdots \vee B_m$$

with A_i being the antecedents and B_i bineg the consequents. Antecedent and consequent are separated by the turnstile "$\vdash$" symbol. The turnstile can be interpreted as "entails" or "yields". Except for another sequent, every antecedent or consequent is a formula in any form (in the underlying logic language).

Sequents can be transformed using the sequent calculus's inference rules. A proof can be visualized as a tree with sequents at the nodes and applications of inference rules at the arcs, where certain rules split a single sequent into two or more new ones. When every branch of the proof ends with a proven sequence, that is, when a formula appears as both an antecedent and a consequent, or when any antecedent is false, or when any consequent is true, the proof is considered to have terminated successfully.

3.2. Neural network

The input–output relation for a generic neural network can be expressed as follows:

$$nn : R^{m\times n} \rightarrow R^{p\times q}, \tag{1}$$

The neural network has two shapes: (m, n) and (p, q), which represent the input and output, respectively. For example, in a regression problem, the shapes will be $(1, n)$ and $(1, 1)$ when the network is trained to approximate a function of n arguments and one output. Now that we have more specifics, we can look at the less universal feed-forward network model. Eq. (1) can also be used to summarize this network; on the other hand, the single fully linked layers l_i functions can be furthered as follows:

$$\mathcal{L}_i : R^{m_i\times n_i} \rightarrow R^{p_i\times q_i}, \tag{2}$$

where the input and output shapes of layer l_i are denoted by (m_i, n_i) and (p_i, q_i), respectively. The forms in the regression problem example will be $(1, n_i)$ and $(1, q_i)$ once again. A neural network layer typically consists of a collection of biases B and weights W. Biases can be thought of as an extra neural network parameter degree of freedom. With the fully-connected layer l_i with weights and biases denoted by W_i, B_i, the function that further describes Eq. (2) is as follows:

$$\mathcal{L}_i(X) = \sigma_i\left(X \times W_i + B_i\right), \tag{3}$$

where the matrix addition operation is +, the matrix-matrix multiplication operation is ×, and the generic activation function is σ_i. $X \in R^{1,m_i}$, $W_i \in R^{m_i,n_i}$, and $B_i \in R^{1,n_i}$ are the variables in a regression problem. As a result, $\mathcal{L}_i(X) \in R^{1,m_i} \rightarrow R^{1,n_i}$ is the layer result.

```
Sequential(
    Linear(10, 100),
    Tanh(),
    Linear(100, 100),
    Tanh(),
    Linear(100, 30),
    Tanh(),
    Linear(30, 3),
    Tanh()
)
```

Listing 1: PyTorch example of a fully-connected neural network.

A sequence of fully-connected layers $\mathcal{L}_1, \mathcal{L}_2, \ldots, \mathcal{L}_n$ make up an n-layer feed-forward neural network. The transformation stated in (3) is applied by each of these layers to either the output of the preceding layer or, in the case of the input layer l_1, to the input data i. Using this idea, we can then express the relation (1):

$$nn(i) = \mathcal{L}_n(\mathcal{L}_{n-1}(\ldots\mathcal{L}_2(\mathcal{L}_1(i)))). \tag{4}$$

For example, a 3-layer neural network can be expressed as follows:

$$nn(i) = \mathcal{L}_3(\mathcal{L}_2\ (\mathcal{L}_1(i))) = \sigma(\sigma(\sigma(W_1 \times i + B_1) \times W_2 + B_2) \times W_3 + B_3).$$

3.3. PyTorch

In this work we use the PyTorch python framework (Paszke et al., 2019; Ansel et al., 2024) to create, train, and manipulate neural networks. A fully connected neural network can be created using PyTorch as shown in Listing 1, where an example of a neural network (called `Sequential` by PyTorch) with a hyperbolic tangent activation function is declared. The network has 4 fully connected layers (called `Linear` by PyTorch) with hyperbolic tangent activation, 10 input features, and 3 output classes (or neurons).

4. Analytical verification using interval arithmetic

In this Section, we detail the use of interval arithmetic to evaluate the bounds of a neural network output. Let $x = \{x^{(1)}, x^{(2)}, \ldots, x^{(n)}\}$ be the input features to a neural network. Each of the single $x^{(k)}$ feature is represented with its upper and lower bounds.

$$x^{(k)} = [\underline{x}^{(k)}, \overline{x}^{(k)}]$$

The objective is to compute an output interval $y = \{y^{(1)}, y^{(2)}, \ldots, y^{(q)}\}$ such that each of the single outputs $y^{(k)}$ are bounded by an upper and lower value.

$$y^{(k)} = [\underline{y}^{(k)}, \overline{y}^{(k)}]$$

This represents the possible range of outputs produced by the neural network for inputs within the input feature $x = \{x^{(1)}, x^{(2)}, \ldots, x^{(n)}\}$.

To compute output intervals using interval arithmetic, we propagate the input intervals from input x through each layer of the neural network. At each layer, we apply interval arithmetic operations that evaluate interval bounds for the intermediate activations. These interval bounds are then used as input intervals for the subsequent layer, continuing the propagation process until we obtain output intervals for the final layer. The following propagation rules are employed.

Sum: $[\underline{a}, \overline{a}] + [\underline{b}, \overline{b}] = [\underline{a} + \underline{b}, \overline{a} + \overline{b}]$

Example: $[-1, 1] + [2, 3] = [1, 4]$.

Subtraction: $[\underline{a}, \overline{a}] - [\underline{b}, \overline{b}] = [\underline{a} - \overline{b}, \overline{a} - \underline{b}]$

Example: $[2, 3] - [1, 2] = [0, 2]$.

Multiplication: $[\underline{a}, \overline{a}]\cdot[\underline{b}, \overline{b}] = [\min(\underline{a}\cdot\underline{b}, \underline{a}\cdot\overline{b}, \overline{a}\cdot\underline{b}, \overline{a}\cdot\overline{b}),\ \max(\underline{a}\cdot\underline{b}, \underline{a}\cdot\overline{b}, \overline{a}\cdot\underline{b}, \overline{a}\cdot\overline{b})]$

Example: $[-1,1]\cdot[2,3]=[-3,3]$.

Reciprocate: $\frac{1}{[\underline{a},\overline{a}]}=\left[\frac{1}{\overline{a}},\frac{1}{\underline{a}}\right]$

Example: $[2,3]^{-1}=[1/3,1/2]$.

Division: $\frac{[\underline{a},\overline{a}]}{[\underline{b},\overline{b}]}=[\underline{a},\overline{a}]\cdot\frac{1}{[\underline{b},\overline{b}]}$

Example: $[0,1]\cdot 1/[2,3]=[0,1/2]$.

let $\sigma:\mathfrak{R}\to\mathfrak{R}$ monotonic increasing, $\sigma\left(\left[\underline{a},\overline{a}\right]\right)=\left[\sigma(\underline{a}),\sigma(\overline{a})\right]$

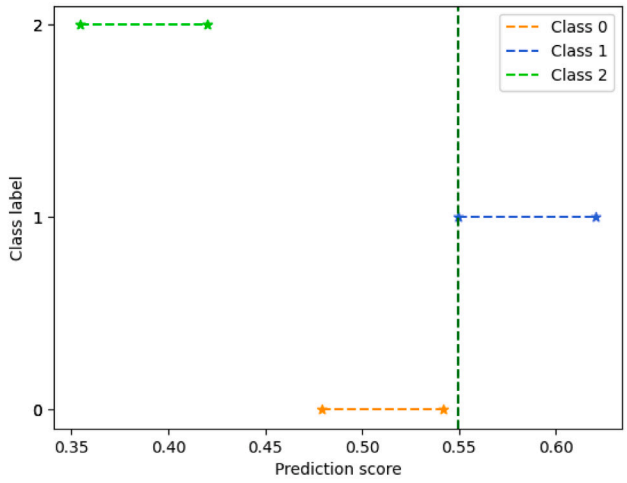
Example: $sigmoid([2,3])=[sigmoid(2),sigmoid(3)]$.

By propagating input intervals through the network layers until the output one, we can evaluate the output intervals to provide rigorous bounds on the network predictions. The output intervals $y=\{y^{(1)},y^{(2)},\ldots,y^{(q)}\}$ represent the possible range of outputs produced by the neural network for inputs within the provided input intervals x. These output intervals serve as verifiable bounds on the model's behavior, enabling the assessment of its reliability and robustness within specified input ranges. This enables us to analytically prove that, within a certain input interval, we can guarantee a certain prediction. Indeed, we just need to prove that, given the desired class label for the prediction, its bounded interval has no intersection with all the other bounded intervals and its lower bound is higher than all the other intervals' upper bounds. For example, if we want to prove that we guarantee class y_1 prediction in a given input interval, we need to show that:

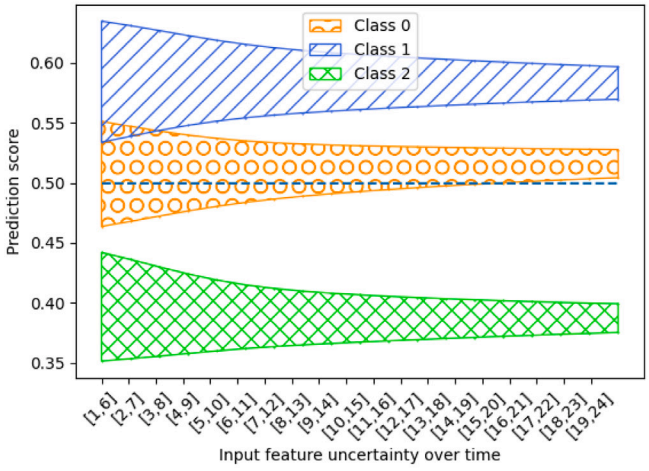
$$\forall i\in[2,M]:y_i\bigcap y_1=\varnothing\wedge\underline{y}_1>\overline{y}_i$$

Programmatically, this can be verified with $\mathcal{O}(M)$ time, where M is the number of network outputs or classes.

Fig. 1 shows an example visualization of using interval arithmetic within neural networks. In Fig. 1(a) we reported the prediction score intervals for a 3-label classifier. In the example, within the provided input intervals, we can guarantee that the prediction is always class 1. Fig. 1(b) shows the evolution over the time of the prediction score intervals when the input intervals vary. This kind of visualization is useful when dealing with dynamic systems that evolve over time and are influenced by the neural network output that, in turn, has an output based on some (or all) of the dynamic system outputs. This is typical of closed-loop systems when the neural network component acts as a controller on some parts of the system.



(a) Prediction score interval for a given input interval



(b) Prediction score interval evolution during time

Fig. 1. Examples of interval prediction scores.

5. Translation of neural networks to formal models

In this section, we start from the network as represented by Eq. (4). We make use of the `matrices` PVS theory to represent both 1-dimensional vectors and two-dimensional matrices, formalizing the application of each layer as a PVS function. Moreover we show the formalization of network input constraints and properties on the network outputs.

5.1. Network specification

Specifically, using Eq. (3), we can formalize the weights and biases of a layer $l_i:R^{1\times m_i}\to R^{1\times n_i}$. These may be found in Listing 2. PVS allows nested lists to be used to initialize dimensional matrices, which are represented by the `MatrixMN` type. For example, a 2×2 matrix can be initialized as `(: (: 1,1 :), (: 2,2 :) :)`, where `(: :)` is the Lisp notation for a list.

```
wi: MatrixMN(m,n) = (:(: w11, w12, ... , w1n :), ...
      , :)
bi: MatrixMN(1,n) = (: b1, b2, ..., bn :)
```

Listing 2: PVS formalization for a fully connected layer.

The PVS definition of layer activation functions, as shown in Listing 3, can be used to implement them simply. ReLU-like functions (Agarap, 2018) can be generalized as general piece-wise linear functions, or leaky relu when the argument's values are positive (i.e., $y=x, x\geq 0$), a leaky relu behaves as the identity function; when the argument's values are negative (i.e., $y=nslope\times x, x\leq 0$), it behaves as a straight line with a positive slope. Listing 3 illustrates how the ReLU function is a specific instance of the leaky relu if $nslope=0$.

```
leaky_relu(x: real, nslope: real):
    real = IF x >= 0 THEN x
           ELSE nslope*x ENDIF
relu(x: real) : real = leaky_relu(x,0)
```

Listing 3: PVS formalization for piece-wise linear activation functions.

By utilizing PVS's `(exp)` formulation of the exponential function, S-shaped functions like the sigmoid and hyperbolic tangent functions can also be expressed, as shown in Listing 4. The sigmoid function can be used as a building block to define the hyperbolic tangent.

Listing 5 formalizes the application of these scalar functions over an entire matrix, where `act_fun` is one of the previously encountered scalar functions.

```
sigmoid(x: real): real =
    1/(1 + exp(-x))
tanh(x: real): real =
    2*sigmoid(2*x) - 1
```

Listing 4: PVS formalization of s-shaped, sigmoid-based activation functions.

```
act(M: Matrix): MatrixMN(rows(M),columns(M)) =
    form_matrix(LAMBDA (i,j:nat):
        act_fun(entry(M)(i,j)), rows(M), columns(M));
```

Listing 5: PVS formalization for generic activation function on a matrix.

To bring everything together, we formalize the network forward pass in its entirety as seen in Eq. (4) and Listing 6.

```
net(input: Matrix): Matrix =
    act(act(input*w1+b1)*w2+b2)*w3+b3 ...
```

Listing 6: PVS formalization for generic activation function on a matrix.

5.2. Network constraints and properties

We can state certain limits on the input layer arguments and a theorem for the network's overall output as we are interested in proving properties on the network outputs. To confine the ith input x_i to be $x_i \in [lb_i, ub_i]$, we can constrain the neural network inputs inside a specific range. According to the theorem, the neural network's ith output is `entry(net(...)(0,i))`. We can constrain each of these outputs by connecting them with a conjunctive or disjunctive phrase as in Listing 7.

```
x1: TYPE = { r: real | r>=lb1 AND r<=ub1 }
x2: TYPE = { r: real | r>=lb2 AND r<=ub2 }
...
xn: TYPE = { r: real | r>lbn AND r<=ubn }
network_bounds: THEOREM
  FORALL (x0in: x1,x1in: x2, ...):
        entry( net( (:(:x0in,x1in, ... :):) ) )(0,0)
     <= ... AND
        entry( net( (:(:x0in,x1in, ... :):) ) )(0,1)
     <= ...
```

Listing 7: PVS constraints for the input variables and theorem on the output.

6. Integration inside PyTorch

In this Section, we detail the practical integration of the two aforementioned techniques within the PyTorch machine-learning framework.

6.1. Automatic theory generation

The previous approach can produce a functioning and provable theory for a feed-forward neural network; however, it is not tractable to manually write a theory when the number and size of layers scale up. Therefore we developed a Python-based tool that can automatically generate an entire PVS theory starting from a pre-trained PyTorch neural network model. The algorithmic complexity of generating a PVS theory from a network model is $\mathcal{O}(n_p)$, where n_p is the number of parameters in the model (e.g. generating the theory for the network in Listing 1 takes 2 s on a 3GHz desktop processor). In the next section, we will cover a more complex use case, with a bigger fully connected network, to highlight the potential capabilities of our approach. Some assumptions must be made before the generation of the theory from the network specifications:

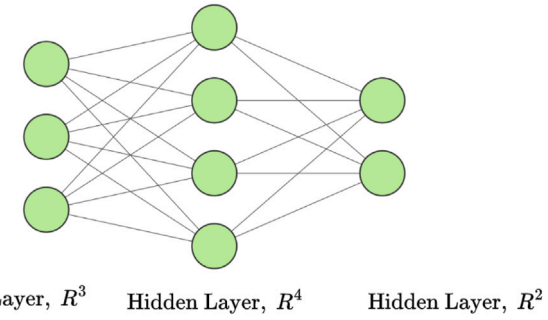


Fig. 2. Simple neural network with two hidden neurons and ReLU activation function.

- The neural network must be a feed-forward network (e.g. multi-layer perceptron).
- The number of neurons in the network must not exceed 60–70 for scalability issues of the prover itself.
- Linear-activation functions are preferable to provide tighter bounds and reduce over-approximations

Such assumptions are also stressed out again in Section 6.4.

6.2. Interval arithmetic computation

We instrumented PyTorch to process interval arithmetic by extending the base `Tensor` class with an `IntervalTensor` class where every tensor element is an instance of a `Interval` class. The actual interval arithmetic is provided by the `pyinterval` library.[4] The library implements an algebraically closed interval system on the extended real number set can be implemented in Python with the help of this package. According to this library's definition, interval objects are made up of a finite union of closed intervals that may or may not be unbound mathematically. In our case, the bounds are all mathematically bound and each interval object is made up of at most one closed interval since the only interval operations involved are the algebraic sum and the function activation. Indeed, the multiplication by a scalar preserves the number of finite closed intervals inside the interval object.

Such integration allows us to seamlessly use IntervalTensor in PyTorch models without the need to recompile the whole framework or change pre-trained models. Layer operation implementations are overridden inside the IntervalTensor class to provide a common interface that adheres to the original PyTorch Tensor one.

6.3. A small example of the two techniques

Starting from a very simple neural network, suppose we have a single-layer neural network as shown in Fig. 2. The correspondent PyTorch model summary is shown in Listing 8.

```
Sequential(
    nn.Linear(in_features=3,out_features=4, bias=
     False),
    nn.ReLU(),
    nn.Linear(in_features=4,out_features=2, bias=
     False)
)
```

Listing 8: PyTorch code for a simple network model with two hidden neurons.

We can produce the correspondent PVS theory with the automatic generation as shown in Listing 9. We then add some constraints on the input variables and a theorem to verify the network output.

[4] https://pyinterval.readthedocs.io/en/latest/.

```
mlp: THEORY
BEGIN
    IMPORTING matrices@matrices
    linear0: MatrixMN(3,4) = (:(:1.,-1.,1.,-1.:)
     ,...,(:1.,-1.,1.,-1.:):)
    linear2: MatrixMN(4,2) = (:(:1.,-1.:)
     ,...,(:1.,-1.:):)
    relu(x: real): real = IF x > 0 THEN x ELSE 0
     ENDIF
    act(M: Matrix): MatrixMN(rows(M),columns(M)) =
     ...
    net(input: Matrix): Matrix = act(input*linear0)*
     linear2
    % Manually added
    x1: TYPE = { r: real | r>=-1 AND r<=1}
    x2: TYPE = { r: real | r>=-1 AND r<=1}
    x3: TYPE = { r: real | r>=-1 AND r<=1}
    network_bounds: THEOREM
    FORALL (x: x1, y: x2):
        entry(net( (:(:x,y:):) ))(0,0) >= 0 AND
        entry(net( (:(:x,y:):) ))(0,1) <= 0
    %|- network_bounds: PROOF
    %|- (grind)
    %|- QED
END mlp
```

Listing 9: Full PVS theory for the simple network shown before.

```
network_bounds : PROOF
(then (grind))
QED network_bounds
```

Listing 10: Proof script for the theorem `network_bounds`

```
net = nn.Sequential(
    nn.Linear(3,4, bias=False),
    nn.ReLU(),
    nn.Linear(2,2, bias=False)
)

intervals = [[-1,1], [-1,1], [-1,1]]
itensor = IntervalTensor(intervals)
bounds = net(itensor)
```

Listing 11: Full PVS theory for the simple network shown before.

In Listing 9 there is the complete theory generated with the manual addition of constraints on the two inputs. Lines 17–19 show the script used to prove the theorem. The output of the prover, with the successful proof indication is reported in Listing 10, where the verb `QED` indicates the completion of the proof.

We can obtain a similar result by exploiting interval arithmetic as shown in Listing 11. Fig. 3 shows the outcome of the experiment, showing the same bounds proved before with the automatic theory generation and formal proof. The vertical line in $x = 0$ corresponds to the bounds specified also in Listing 11. This is verified numerically by the two output intervals for the two classes:

- Class label 1: $[-14, -0.4]$
- Class label 0: $[0.4, 14]$

6.4. Limits of the PVS theory generation and proof

When using the PVS theory generation we can provide a higher-order logic formalization of a neural network that can be formally evaluated to guarantee a given property. However, the approach presents some limitations, hence the development of the other interval arithmetic technique. The core limitations of the approach are the following: (i) poor scalability with the number of internal layers and neurons in particular when considering the ReLU activation function. Although computationally simple, for each ReLU evaluation the prover must fork into two branches, depending on the input sign. This means that we generate two branches of the proof tree for each intermediate layer output neuron. The branch evaluation then scales like $\mathcal{O}(2^L)$, with L being the number of intermediate outputs, leading to exponentially increasing computational complexity; (ii) poor scalability with the number of input neurons. For each of the branches to evaluate, the prover must find a solution to an inequality system with N variable, being N the number of network inputs (iii) automated reasoning on the network is limited when using other non-linear transcendental activation functions (e.g. sigmoid, tanh) since they need dedicated solution strategies (Titolo et al., 2023). These limitations can be solved using interval arithmetic to evaluate the network output bounds. Furthermore, this results in an over-approximation of the bounds, making the technique more conservative.

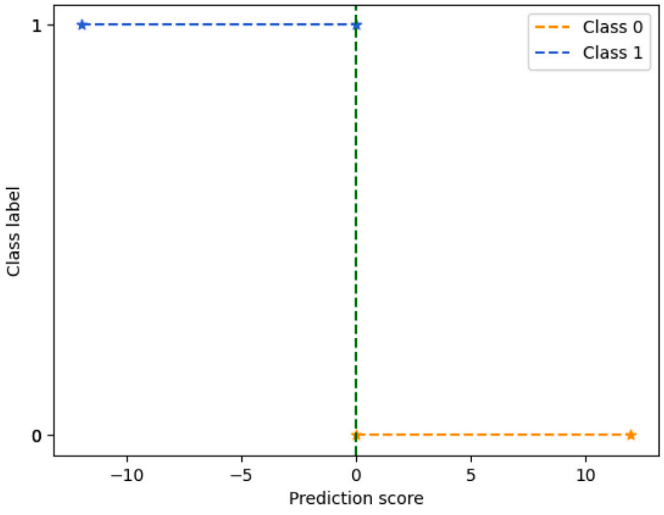


Fig. 3. Output bounds for the predictions of the two classes for the network shown in Fig. 2.

Fig. 4 shows how the two techniques can be combined to provide guarantees on a closed-loop system, obtaining the bounds with the interval arithmetic and modeling the network as a black box with such bounds in PVS, alongside the formal model of the rest of the system. We will refer to the model of the network as a black box (input interval to interval output relation) as "network formal model".

Figs. 5 and 6 details the step involved in the generation of the neural network formal model and the proof process: (i) the pre-trained network is read from the file, (ii) input intervals related to the verification scenario are read from the file and the neural network is evaluated with such intervals producing output intervals, (iii) a relation between input intervals and output (the neural network formal model) is generated and saved, (iv) the PVS prover takes as inputs the closed-loop system, the neural network formal model and the safety properties to be proven, (v) the proof is carried out with the PVS prover and (vi) the generated proof is saved to file.

As a final remark we stress that with the combination of interval arithmetic and PVS prover, the limitations expressed in Section 6.1 no longer apply. Indeed, through PyTorch, we can build any network architecture and perform inference using IntervalTensor through it, without any limits to the network size and topology. Therefore, the prover does not need to evaluate the entire network but only the formal model generated with the interval arithmetic, which is sensibly easier to handle. In the next section, we show a use-case application of this approach to an adaptive cruise control where the core model predictive controller is substituted by a neural network controller.

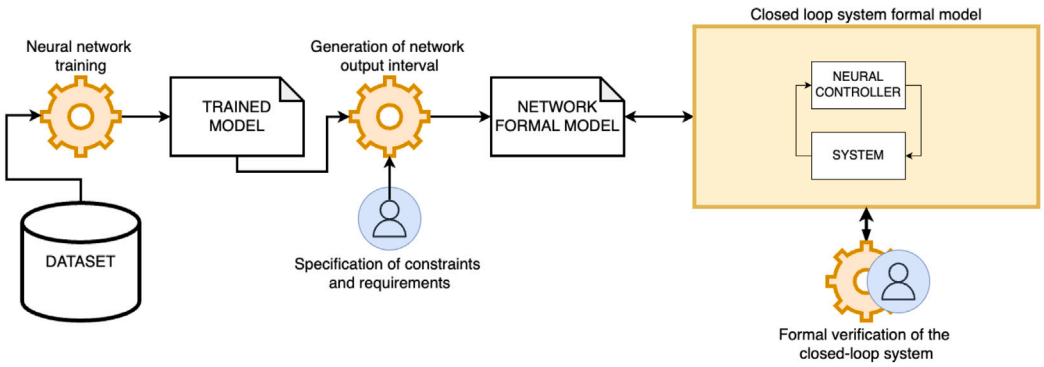


Fig. 4. Architectural components of the proposed approach.

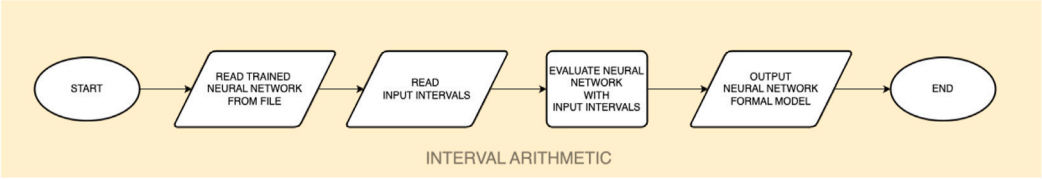


Fig. 5. Flowchart of the generation of the neural network formal model through interval arithmetic computation.

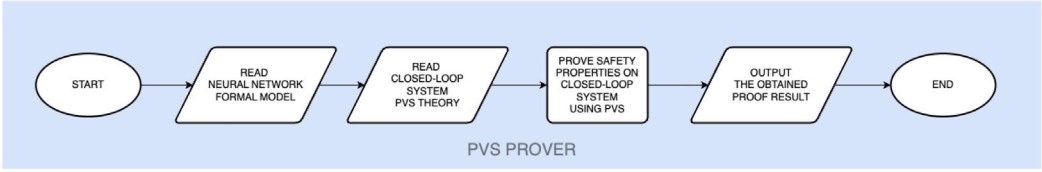


Fig. 6. Flowchart of the proof process using the neural network model and the closed-loop system PVS theory.

Fig. 7. Scheme for the MPC controlled cruise control application.

7. Use case: adaptive cruise control with surrogated controller

In this section, we take a look at the autonomous driving application[5] shown in Fig. 7. Note that the considerations done in this section are strictly related to the scenario considered and the model constraints and specifications and are not intended to apply generically to the adaptive cruise control application for every scenario.

Specifically, we take into consideration the straightforward scenario in which the autonomous driving system determines the ego car's acceleration by detecting the position and speed of the lead car. To solve this problem, for instance, the speed of the ego car is controlled using a model-predictive controller (MPC) in two operational modes: (i) adhering to the set velocity v_{set} and (ii) keeping a safe distance from the lead car equal to $d_{safe} = t_{gap} * v_{ego} + d_{default}$, where t_{gap} denotes a time constant, v_{ego} is the ego car's speed, and $d_{default}$ is the standstill spacing.

Three nominal inputs are accepted by the MPC controller: the lead car's relative velocity, relative distance, and the ego car's longitudinal velocity. The ego car's acceleration control is the only output, which is the only regulated quantity. Consequently, Fig. 8 illustrates how a surrogate model based on a neural network with the following topology might take the place of the MPC controller.

$$n(v_{ego}, d_{rel}, v_{rel}) = a_{ego}. \tag{5}$$

Consequently, by employing the method suggested in Bernardeschi et al. (2023), we can model the use case scenario and produce a set of tuples $v_{ego}, d_{rel}, v_{rel}, a_{ego}$ that will serve as input and output data for the training procedure. Using the ADAM optimizer (Kingma and Ba, 2014), we trained the network in Fig. 8 (140 learnable parameters, sigmoid activation function) for 1000 epochs, using the mean squared error (MSE) loss function and a learning rate of 0.001. A final MSE of 0.034 was obtained. The rationale behind substituting the model predictive controller with a neural network is well-known in the literature (Wang et al., 2022; Cheng et al., 2015). Based on the current condition of the plant, model predictive control (MPC) solves a constrained quadratic programming (QP) optimization problem in real-time. As MPC resolves

[5] https://it.mathworks.com/help/mpc/ug/adaptive-cruise-control-using-model-predictive-controller.html.

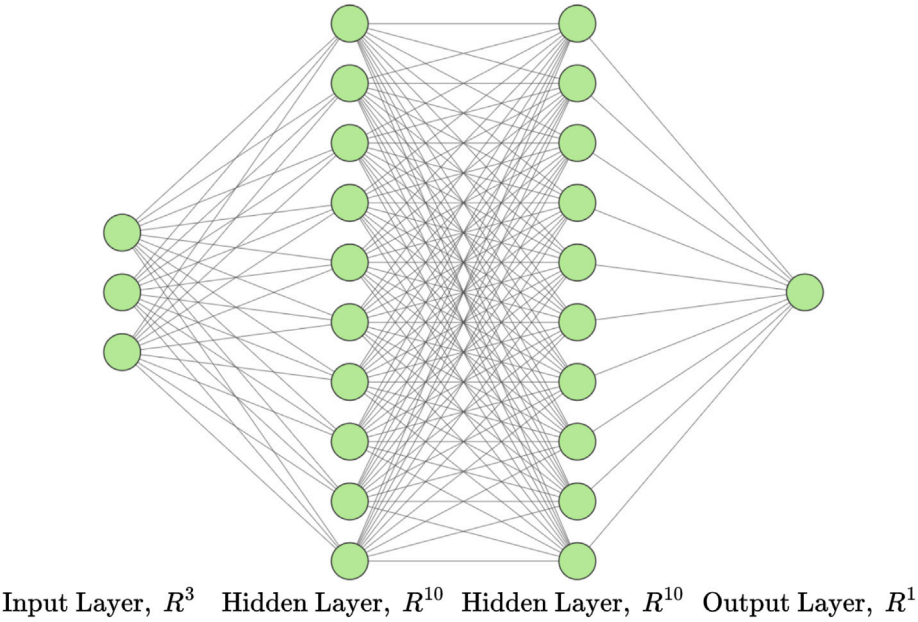


Fig. 8. Neural network for solving the MPC regression problem.

```
1  closed_loop_system: THEORY
2  BEGIN
3    x0_lead: posreal = 35
4    t: VAR posreal
5    v_lead(t): posreal;
6    x_lead(t): posreal = v_lead(t)*t + x0_lead;
7    a_lead(t): real = 0;
8    v_ego(t): real
9    a_controller(drel: real, vrel: real, vabs: real):
       real
10
11   x_ego(t): posreal
12   a_ego(t): real =
13     a_controller(x_lead(t) - x_ego(t), v_lead(t) -
       v_ego(t), v_ego(t))
14   ego_x_law: AXIOM
15       x_ego(t) = 1/2*( a_ego(t) * t^2 )
16   ego_v_law: AXIOM
17       v_ego(t) = a_ego(t)
18
19   safety_distance: THEOREM
20   FORALL(t): x_lead(t) >= x_ego(t)+35
21 END dynamics
```

Listing 12: Formalization for the closed-loop system model of ego and lead car

its optimization problem in an open-loop manner, a deep neural network may be used in place of the controller. Therefore, evaluating a deep neural network can potentially be more computationally efficient than real-time QP problem-solving, especially when employed in a constrained on-board embedded device.

7.1. Modeling and proving properties for the neural network

Listing 12 shows how we formalized the dynamic equations of the closed-loop system. In particular, at line 15 we declared the control law equation represented by the neural network controller.

Safety property: safety distance between the two cars. We want to ensure that the absolute position of the lead car is always greater than the absolute position of the ego car plus a safety distance in the considered scenario. For example, considering the maximum v_{ego} velocity shown during the training ($v_{ego} = 25$ m/s) and the default time gap $t_{gap} = 1.4$ s, it is possible to set the safety distance at $d_s = t_{gap} \times v_{ego} = 1.4$ s $\times$ 25 m/s = 35 m. Listing 13 shows the formulation for such a property.

```
safety_distance: THEOREM
  FORALL(t): x_lead(t) >= x_ego(t)+35
```

Listing 13: Theorem on maintaining the safety distance

7.2. Evaluation of the neural network bounds

To evaluate the boundaries of the neural network controller output in the considered scenario, we exploited the approach shown in Section 6 to analytically compute the output boundaries. Given $v_{ego} =$ 25 m/s, and $v_{rel} \in [-2, 2]$ $\frac{\text{m}}{\text{s}^2}$ we widely varied the d_{rel} input to assess the neural network output boundaries when varying the relative distance. This reflects on a scenario when the ego car is closer than the safety distance and should decelerate to keep $d_s \geq 35$ m. Fig. 9 shows the evolution of the boundaries over time, while the ego car is decelerating, hence pulling away from the lead car. As we can see the bounds guarantee us that the acceleration output is negative until around the safety distance is reached, then it becomes positive to be able to follow the lead car v_{set}.

To compare our open-loop verification approach on the neural network bounds we set up a similar experiment where different neural network provers were asked to prove the following property, where $n(v_{ego}, v_{rel}, d_{rel})$ is the neural network as defined in Eq. (5):

$$p(n)\text{: } d_{rel} \leq 35, v_{ego} = 25, v_{rel} \in [-2, 2] \Rightarrow n(v_{ego}, v_{rel}, d_{rel}) < 0$$

For the comparison we are interested in two metrics: (i) completion of the verification (i.e. if the verifier is able to verify the property or not), (ii) verification time (i.e. the time it took the verifier to complete the proof). Table 2 shows the time performance of this approach compared to the Marabou neural network verification tool (Katz et al., 2019), the nnenum tool (Bak, 2021) and alpha-beta-crown (Wang et al., 2021). We measured 1000 independent runs of the proof and measured the average time to proof completion as well as the standard deviation.

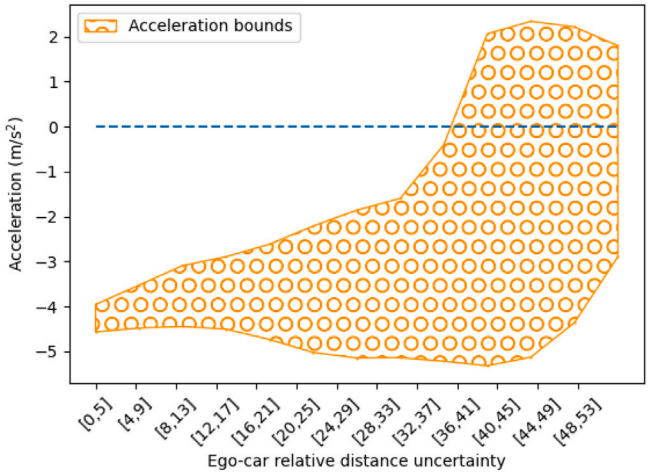


Fig. 9. Acceleration output uncertainty when varying the relative distance uncertainty over time.

Table 2
Time comparison with other neural network verification tools.

	Verified	Time (s)
Marabou (Katz et al., 2019)	Yes	0.1311 ± 0.015
nnenum (Bak, 2021)	Yes	0.057 ± 0.0044
alpha-beta-crown (Wang et al., 2021)	Yes	0.0044 ± 0.0023
This work	Yes	$\mathbf{0.0028 \pm 0.001}$

```
1 a_accel: {x: real | x > 0}
2 a_decel: {x: real | x < 0}
3 a_controller(drel: real, vrel: real, vabs: real):
      real =
4     IF drel < 37
5     THEN a_decel
6     ELSE a_accel
7     ENDIF
```

Listing 14: Behavior of the neural network controller

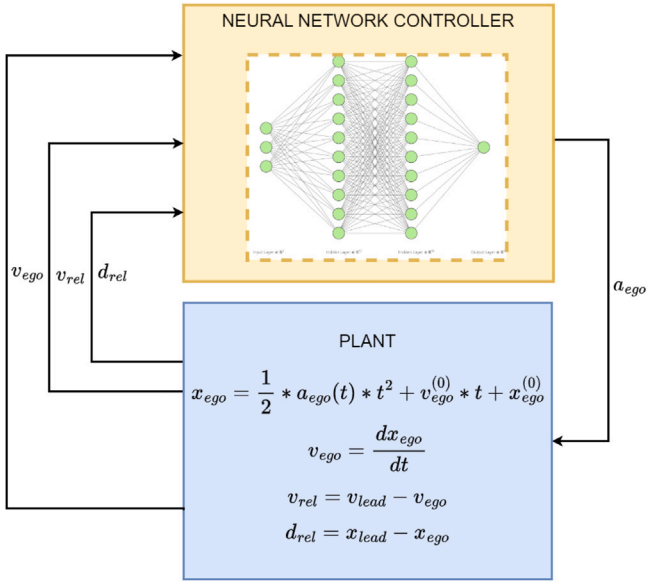


Fig. 10. Visualization of the closed-loop system with the ego car and the neural network controller.

```
1  %|- safety_distance: PROOF
2  %|- (skeep)
3  %|- (expand x_lead)
4  %|- (rewrite ego_x_law)
5  %|- (expand a_ego)
6  %|- (expand x_lead)
7  %|- (grind) % <- fork into two subgoals
8  %|- (lemma a_law_neg)
9  %|- (inst?)
10 %|- (field) % <- completion of first subgoal
11 %|- (field)
12 %|- (rewrite ego_x_law)
13 %|- (field) % <- completion of second subgoal
14 %|- QED % <- proof completion
```

Listing 15: Proof script for the `safety_distance` property

From that table it can be seen how all the tool are able to verify the property, but our approach takes lesss time.

We can use the analytically proven boundaries to model the neural network behavior in PVS, as shown in Listing 14. In lines 1–2 we modeled the acceleration and deceleration values as two sets of real numbers `a_accel, a_decel` that are, respectively positive and negative. This allows us to use them during the proof without actually knowing their magnitude.

7.3. Closed-loop verification

Once we integrated the formalization of the network behavior inside the PVS theory, we can prove the closed loop's safety property for the system shown in Fig. 10 for this particular scenario. PVS proof can be done interactively or by providing a proof script to the prover. Hereafter in Listing 15 is an example of the latter approach, with a proof script for the `safety_distance` theorem. We put `%|- QED` at the end of the script to mark that the proof should be completed at that point. If a completed proof is reached (i.e. all the sequent calculus implications are proved) the prover will complete. We report the proof tree for the prover output with the previous script, highlighting the successful verification of the safety property in Fig. 11 for this specific scenario.

7.4. Limitations and scalability of the approach

The presented approach can provide output bounds for any neural network that can be implemented through the PyTorch library. Furthermore, these bounds can be used in formal models using PVS to characterize the behavior of closed-loop dynamic systems. The applicability of interval arithmetic to neural networks can present some limitations that are intrinsic to the arithmetic itself. In general, interval arithmetic provides an over-approximation of the bounds. The over-approximation may grow numerically with the number of consecutive arithmetic operations on intervals. In detail, interval arithmetic suffers from the so-called dependency problem: if the same interval appears multiple times inside an expression the resulting interval will result in an overapproximation of the actual value interval. For example, suppose to have an interval $x = [-1, 1]$ and the expression $y = x - x$. Logically the result should be 0. However, if we apply the interval arithmetic rules, the result will be $y = [-2, 2]$ which contains the actual value but overapproximates it. In future works, we plan to mitigate this problem by constraining the overapproximation of the result.

From the scalability point of view, the complexity of deriving the network bounds is the same as performing inference on the neural network, with the added cost of handling interval arithmetic. While inference is already performed efficiently, we plan to optimize the inference of the intervals through the network exploiting GPU acceleration as much as possible in future works.

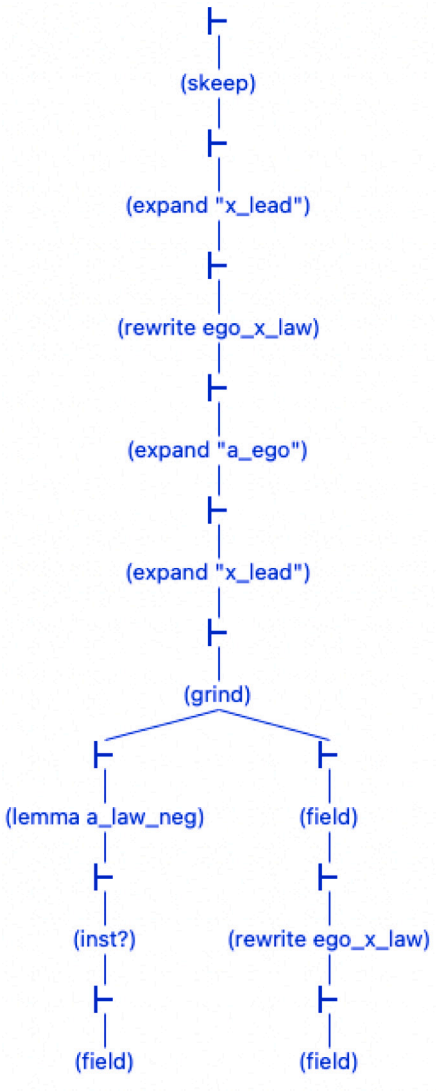


Fig. 11. PVS proof tree for the `safety_distance` property.

8. Conclusion

We introduced a novel verification approach that leverages the strengths of interval arithmetic and theorem-proving formal verification technique. This combination provides a robust framework for ensuring the safety and reliability of neural networks in closed-loop systems.

Our approach seamlessly integrates with the PyTorch deep learning framework, ensuring that it can be adopted without significant changes to existing workflows. This allowed us to formalize a closed-loop system of an adaptive cruise control application where the model-predictive controller was replaced by a well-trained neural network. Finally, we were able to specify and verify safety requirements on the system-controlled variable.

Our approach demonstrated significant improvements in verification performance, considerably reducing the time overhead when compared to other state-of-the-art approaches. Future works will include the optimization of the interval arithmetic method mitigating the limitations of the approach by constraining the overapproximation of the result and accelerating interval computation through GPU as much as possible. Moreover, further expansions of this work will add more detailed dynamics of the vehicle including modeling the behavior in PVS with a set of differential equations to consider more complex scenarios.

CRediT authorship contribution statement

Federico Rossi: Writing – review & editing, Writing – original draft, Methodology, Investigation, Data curation, Conceptualization. **Cinzia Bernardeschi:** Writing – review & editing, Conceptualization. **Marco Cococcioni:** Writing – review & editing, Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

Acknowledgments

We thank the reviewers for the insightful comments and suggestions.

Work funded by: the PNRR - M4C2 - Investimento 1.3, Partenariato Esteso PE00000013 - "FAIR - Future Artificial Intelligence Research" - Spoke 1 "Human-centered AI" under the NextGeneration EU programme; the European High Performance Computing Joint Undertaking (JU) under Framework Partnership Agreement No 800928 and Specific Grant Agreement No 101036168 (EPI SGA2); the Italian Ministry of University and Research (MUR) in the framework of the FoReLab and CrossLab projects (Departments of Excellence).

References

Agarap, A.F., 2018. Deep learning using rectified linear units (relu). arXiv preprint arXiv:1803.08375.

Aleksandrov, A., Völlinger, K., 2023. Formalizing piecewise affine activation functions of neural networks in Coq. In: Rozier, K.Y., Chaudhuri, S. (Eds.), NASA Formal Methods. In: Lecture Notes in Computer Science, Springer Nature Switzerland, Cham, pp. 62—-78. http://dx.doi.org/10.1007/978-3-031-33170-1_4.

Althoff, M., 2015. An introduction to CORA 2015. In: Proc. of the 1st and 2nd Workshop on Applied Verification for Continuous and Hybrid Systems. EasyChair, pp. 120–151. http://dx.doi.org/10.29007/zbkv.

Alur, R., 2011. Formal verification of hybrid systems. In: 2011 Proceedings of the Ninth ACM International Conference on Embedded Software. EMSOFT, pp. 273–278. http://dx.doi.org/10.1145/2038642.2038685.

Alur, R., 2015. Principles of Cyber-Physical Systems. MIT Press.

Ansel, J., Yang, E., He, H., Gimelshein, N., Jain, A., Voznesensky, M., Bao, B., Bell, P., Berard, D., Burovski, E., Chauhan, G., Chourdia, A., Constable, W., Desmaison, A., DeVito, Z., Ellison, E., Feng, W., Gong, J., Gschwind, M., Hirsh, B., Huang, S., Kalambarkar, K., Kirsch, L., Lazos, M., Lezcano, M., Liang, Y., Liang, J., Lu, Y., Luk, C., Maher, B., Pan, Y., Puhrsch, C., Reso, M., Saroufim, M., Siraichi, M.Y., Suk, H., Suo, M., Tillet, P., Wang, E., Wang, X., Wen, W., Zhang, S., Zhao, X., Zhou, K., Zou, R., Mathews, A., Chanan, G., Wu, P., Chintala, S., 2024. PyTorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In: 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2. ASPLOS'24, ACM, http://dx.doi.org/10.1145/3620665.3640366, URL: https://pytorch.org/assets/pytorch2-2.pdf.

Antsaklis, P., 1990. Neural networks for control systems. IEEE Trans. Neural Netw. 1 (2), 242–244. http://dx.doi.org/10.1109/72.80237.

Bak, S., 2021. Nnenum: Verification of ReLU neural networks with optimized abstraction refinement. In: NASA Formal Methods: 13th International Symposium, NFM 2021, Virtual Event, May 24–28, 2021, Proceedings. Springer-Verlag, Berlin, Heidelberg, pp. 19—-36. http://dx.doi.org/10.1007/978-3-030-76384-8_2.

Bak, S., Tran, H.-D., 2022. Neural network compression of ACAS xu early prototype is unsafe: Closed-loop verification through quantized state backreachability. In: Deshmukh, J.V., Havelund, K., Perez, I. (Eds.), NASA Formal Methods. In: Lecture Notes in Computer Science, Springer International Publishing, Cham, pp. 280—-298. http://dx.doi.org/10.1007/978-3-031-06773-0_15.

Bernardeschi, C., Cococcioni, M., Palmieri, M., Rossi, F., 2023. Training neural networks in cyber-physical systems using design space exploration and co-simulation. In: 4th International Conference on Electrical, Communication and Computer Engineering. ICECCE 2023, pp. 1–7. http://dx.doi.org/10.1109/ICECCE61019.2023.10442825.

Cheng, L., Liu, W., Hou, Z.-G., Yu, J., Tan, M., 2015. Neural-network-based nonlinear model predictive control for piezoelectric actuators. IEEE Trans. Ind. Electron. 62 (12), 7717–7727. http://dx.doi.org/10.1109/TIE.2015.2455026.

Cococcioni, M., Rossi, F., Ruffaldi, E., Saponara, S., de Dinechin, B.D., 2021. Novel arithmetics in deep neural networks signal processing for autonomous driving: Challenges and opportunities. IEEE Signal Process. Mag. 38 (1), 97–110. http://dx.doi.org/10.1109/MSP.2020.2988436.

Dutertre, B., 1996. Elements of mathematical analysis in PVS. In: Goos, G., Hartmanis, J., van Leeuwen, J., von Wright, J., Grundy, J., Harrison, J. (Eds.), Theorem Proving in Higher Order Logics. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 141–156. http://dx.doi.org/10.1007/BFb0105402.

Ehlers, R., 2017. Formal verification of piece-wise linear feed-forward neural networks. In: D'Souza, D., Narayan Kumar, K. (Eds.), Automated Technology for Verification and Analysis. In: Lecture Notes in Computer Science, Springer International Publishing, Cham, pp. 269–286. http://dx.doi.org/10.1007/978-3-319-68167-2_19.

Emami, S.A., Castaldi, P., Banazadeh, A., 2022. Neural network-based flight control systems: Present and future. Annu. Rev. Control 53, 97—-137. http://dx.doi.org/10.1016/j.arcontrol.2022.04.006.

Fan, J., Huang, C., Chen, X., Li, W., Zhu, Q., 2020. ReachNN*: A tool for reachability analysis of neural-network controlled systems. In: Hung, D.V., Sokolsky, O. (Eds.), Automated Technology for Verification and Analysis. Springer International Publishing, Cham, pp. 537–542. http://dx.doi.org/10.1007/978-3-030-59152-6_30.

Hickey, T., Ju, Q., Van Emden, M.H., 2001. Interval arithmetic: From principles to implementation. J. ACM 48 (5), 1038—-1068. http://dx.doi.org/10.1145/502102.502106.

Huang, C., Fan, J., Li, W., Chen, X., Zhu, Q., 2019. ReachNN: Reachability analysis of neural-network controlled systems. ACM Trans. Embed. Comput. Syst. 18 (5s), http://dx.doi.org/10.1145/3358228.

Huang, Z., Shen, Y., Li, J., Fey, M., Brecher, C., 2021. A survey on AI-Driven Digital Twins in industry 4.0: Smart manufacturing and advanced robotics. Sensors 21 (19), http://dx.doi.org/10.3390/s21196340.

Ivanov, R., Carpenter, T., Weimer, J., Alur, R., Pappas, G., Lee, I., 2021. Verisig 2.0: Verification of neural network controllers using taylor model preconditioning. In: Silva, A., Leino, K.R.M. (Eds.), Computer Aided Verification. In: Lecture Notes in Computer Science, Springer International Publishing, Cham, pp. 249—-262. http://dx.doi.org/10.1007/978-3-030-81685-8_11.

Ivanov, R., Weimer, J., Alur, R., Pappas, G.J., Lee, I., 2019. Verisig: verifying safety properties of hybrid systems with neural network controllers. In: Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control. HSCC '19, Association for Computing Machinery, New York, NY, USA, pp. 169—-178. http://dx.doi.org/10.1145/3302504.3311806.

Jeffrey, N., Tan, Q., Villar, J.R., 2023. A review of anomaly detection strategies to detect threats to cyber-physical systems. Electronics 12 (15), http://dx.doi.org/10.3390/electronics12153283.

Jin, L., Li, S., Yu, J., He, J., 2018. Robot manipulator control using neural networks: A survey. Neurocomputing 285, 23–34. http://dx.doi.org/10.1016/j.neucom.2018.01.002.

Katz, G., Barrett, C., Dill, D.L., Julian, K., Kochenderfer, M.J., 2017. Reluplex: An efficient SMT solver for verifying deep neural networks. In: Majumdar, R., Kunčak, V. (Eds.), Computer Aided Verification. Springer International Publishing, Cham, pp. 97–117. http://dx.doi.org/10.1007/978-3-319-63387-9_5.

Katz, G., Huang, D.A., Ibeling, D., Julian, K., Lazarus, C., Lim, R., Shah, P., Thakoor, S., Wu, H., Zeljić, A., Dill, D.L., Kochenderfer, M.J., Barrett, C., 2019. The marabou framework for verification and analysis of deep neural networks. In: Dillig, I., Tasiran, S. (Eds.), Computer Aided Verification. In: Lecture Notes in Computer Science, Springer International Publishing, Cham, pp. 443–452. http://dx.doi.org/10.1007/978-3-030-25540-4_26.

Kingma, D.P., Ba, J., 2014. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980.

Kochdumper, N., Schilling, C., Althoff, M., Bak, S., 2023. Open- and closed-loop neural network verification using polynomial zonotopes. In: NASA Formal Methods Symposium. Springer, pp. 16–36. http://dx.doi.org/10.1007/978-3-031-33170-1_2.

Kotha, S., Brix, C., Kolter, Z., Dvijotham, K., Zhang, H., 2024. Provably bounding neural network preimages. http://dx.doi.org/10.48550/arXiv.2302.01404.

Lopez, D.M., Althoff, M., Forets, M., Johnson, T.T., Ladner, T., Schilling, C., 2023a. ARCH-COMP23 category report: Artificial intelligence and neural network control systems (AINNCS) for continuous and hybrid systems plants. In: Frehse, G., Althoff, M. (Eds.), Proceedings of 10th International Workshop on Applied Verification of Continuous and Hybrid Systems. ARCH23, In: EPiC Series in Computing, vol. 96, EasyChair, pp. 89–125. http://dx.doi.org/10.29007/x38n.

Lopez, D.M., Choi, S.W., Tran, H.-D., Johnson, T.T., 2023b. NNV 2.0: The neural network verification tool. In: Enea, C., Lal, A. (Eds.), Computer Aided Verification. In: Lecture Notes in Computer Science, Springer Nature Switzerland, Cham, pp. 397—-412. http://dx.doi.org/10.1007/978-3-031-37703-7_19.

Lopez, D.M., Johnson, T.T., Bak, S., Tran, H.-D., Hobbs, K.L., 2023c. Evaluation of neural network verification methods for air-to-air collision avoidance. J. Air Transp. 31 (1), 1–17. http://dx.doi.org/10.2514/1.D0255.

Masci, P., Muñoz, C.A., 2019. An integrated development environment for the prototype verification system. Electron. Proc. Theor. Comput. Sci. 310, 35—-49. http://dx.doi.org/10.4204/eptcs.310.5.

Narodytska, N., Kasiviswanathan, S., Ryzhyk, L., Sagiv, M., Walsh, T., 2018. Verifying properties of binarized deep neural networks. Proc. AAAI Conf. Artif. Intell. 32 (11), http://dx.doi.org/10.1609/aaai.v32i1.12206.

Owre, S., Rajan, S., Rushby, J., Shankar, N., Srivas, M., 1996. PVS: combining specification, proof checking, and model checking. In: Alur, R., Henzinger, T. (Eds.), Computer-Aided Verification, CAV '96. In: Lecture Notes in Computer Science, vol. 1102, Springer, Berlin, Heidelberg, pp. 411–414.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., Chintala, S., 2019. PyTorch: An imperative style, high-performance deep learning library. In: Advances in Neural Information Processing Systems 32. Curran Associates, Inc., pp. 8024–8035. http://dx.doi.org/10.5555/3454287.3455008.

Pulina, L., Tacchella, A., 2010. An abstraction-refinement approach to verification of artificial neural networks. In: Touili, T., Cook, B., Jackson, P. (Eds.), Computer Aided Verification. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 243–257. http://dx.doi.org/10.1007/978-3-642-14295-6_24.

Putnik, G.D., Manupati, V.K., Pabba, S.K., Varela, L., Ferreira, F., 2021. Semi-Double-loop machine learning based CPS approach for predictive maintenance in manufacturing system based on machine status indications. CIRP Ann 70 (1), 365–368. http://dx.doi.org/10.1016/j.cirp.2021.04.046.

Rathore, M.M., Shah, S.A., Shukla, D., Bentafat, E., Bakiras, S., 2021. The role of AI, machine learning, and big data in digital twinning: A systematic literature review, challenges, and opportunities. IEEE Access 9, 32030–32052. http://dx.doi.org/10.1109/ACCESS.2021.3060863.

Rossi, F., Bernardeschi, C., Cococcioni, M., Palmieri, M., 2024. Towards formal verification of neural networks in cyber-physical systems. In: 16th NASA Formal Methods Symposium, Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). Vol. 14627, Springer Nature Switzerland, pp. 207–222. http://dx.doi.org/10.1007/978-3-031-60698-4_12.

Shi, Z., Jin, Q., Kolter, Z., Jana, S., Hsieh, C.-J., Zhang, H., 2024. Neural network verification with branch-and-bound for general nonlinearities. http://dx.doi.org/10.48550/arXiv.2405.21063.

Smullyan, R.M., 1968. First-Order Logic. Preliminaries. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 43–52. http://dx.doi.org/10.1007/978-3-642-86718-7_4.

Song, X., Sun, P., Song, S., Stojanovic, V., 2023. Quantized neural adaptive finite-time preassigned performance control for interconnected nonlinear systems. Neural Comput. Appl. 35 (21), 15429–15446. http://dx.doi.org/10.1007/s00521-023-08361-y.

Song, X., Wu, C., Song, S., Stojanovic, V., Tejado, I., 2024. Fuzzy wavelet neural adaptive finite-time self-triggered fault-tolerant control for a quadrotor unmanned aerial vehicle with scheduled performance. Eng. Appl. Artif. Intell. 131, 107832. http://dx.doi.org/10.1016/j.engappai.2023.107832.

Titolo, L., Moscato, M., Feliú, M.A., Dutle, A., Muñoz, C., 2023. Floating-point round-off error analysis of safety-critical avionics software. In: Arceri, V., Cortesi, A., Ferrara, P., Olliaro, M. (Eds.), Challenges of Software Verification. Springer Nature Singapore, Singapore, pp. 197–220. http://dx.doi.org/10.1007/978-981-19-9601-6_11.

Urban, C., Miné, A., 2021. A review of formal methods applied to machine learning. CoRR abs/2104.02466, URL: https://arxiv.org/abs/2104.02466.

Vereno, D., Harb, J., Neureiter, C., 2023. Paving the way for reinforcement learning in smart grid co-simulations. In: Software Engineering and Formal Methods. SEFM 2022 Collocated Workshops: AI4EA, F-IDE, CoSim-CPS, CIFMA, LNCS 13765. pp. 242—-257. http://dx.doi.org/10.1007/978-3-031-26236-4_21.

Wang, D., Shen, Z.J., Yin, X., Tang, S., Liu, X., Zhang, C., Wang, J., Rodriguez, J., Norambuena, M., 2022. Model predictive control using artificial neural network for power converters. IEEE Trans. Ind. Electron. 69 (4), 3689–3699. http://dx.doi.org/10.1109/TIE.2021.3076721.

Wang, S., Zhang, H., Xu, K., Lin, X., Jana, S., Hsieh, C.-J., Kolter, J.Z., 2021. Beta-CROWN: Efficient bound propagation with per-neuron split constraints for complete and incomplete neural network robustness verification. http://dx.doi.org/10.48550/arXiv.2103.06624.

Xiang, W., Tran, H.-D., Johnson, T.T., 2018. Output reachable set estimation and verification for multilayer neural networks. IEEE Trans. Neural Netw. Learn. Syst. 29 (11), 5777–5783. http://dx.doi.org/10.1109/TNNLS.2018.2808470.

Xu, K., Shi, Z., Zhang, H., Wang, Y., Chang, K.-W., Huang, M., Kailkhura, B., Lin, X., Hsieh, C.-J., 2020. Automatic perturbation analysis for scalable certified robustness and beyond. Adv. Neural Inf. Process. Syst. 33, 1129–1141. http://dx.doi.org/10.5555/3495724.3495820.

Zhang, H., Wang, S., Xu, K., Li, L., Li, B., Jana, S., Hsieh, C.-J., Kolter, J.Z., 2022. General cutting planes for bound-propagation-based neural network verification. http://dx.doi.org/10.48550/arXiv.2208.05740.