ECAI 2024
U. Endriss et al. (Eds.)

doi:10.3233/FAIA240633

Automated Synthesis of Certified Neural Networks

Matteo Zavatteri[a,*], **Davide Bresolin**[a,**] **and Nicolò Navarin**[a,***]

[a]Department of Mathematics, University of Padova, Italy

Abstract. Neural networks find applications in many safety-critical systems that raise concerns about their deployment: Are we sure the network will never advise doing anything violating a set of safety constraints? Formal verification has been recently applied to prove whether an existing neural network is certified for some property (i.e., if it satisfies the property for all possible inputs) or not. Formal verification can prove that a network respects the property, but cannot fix a network that does not respect it. In this paper we focus on the automated synthesis of certified neural networks, that is, on how to automatically build a network that is guaranteed to respect some required properties. We exploit a Counter Example Guided Inductive Synthesis (CEGIS) loop that alternates Deep Learning, Formal Verification, and a novel data generation technique that augments the training data to synthesize certified networks in a fully automatic way. An application of a proof-of-concept implementation of the framework shows the feasibility of the approach.

1 Introduction and related work

Applications of neural networks include fields where they act as action advisors and controllers for safety-critical systems, where safety is paramount, and errors can be extremely expensive or dangerous. When a neural network is deployed in a safety-critical system, it is of extreme importance to prove that it satisfies the desired properties in all possible scenarios and under all possible inputs, as testified, for instance, in a recent technical paper on the use of formal methods for neural networks certification in aviation [13]. Moreover, the European Union is working on the text of the AI act [14] that will require for all AI systems categorized as *high-risk* to be assessed before being put on the market and also throughout their lifecycle.

Formal verification provides algorithms and methodologies that can exhaustively explore the state space of a system to guarantee that the system itself respects the desired properties. Hence, the certification of neural networks has recently received attention from the formal verification community: the annual competition VNN-COMP [5] and the standard format VNN-LIB [31] have been established to compare state-of-the-art neural network verification tools and share benchmarks and test cases.

Many of the current approaches to verify a neural network reduce the problem to finding a solution to a constraint satisfaction problem (CSP). Such a CSP consists of a set of constraints encoding the neural network and a set of constraints encoding the property, and it is typically formulated within the frameworks of Mixed Integer Linear Programming (MILP), Satisfiability Modulo Theories (SMT), or more generally Constraint Programming (CP). Some selected references of these (and other) formal verification methods can be found in [2]. Some tools such as RELUPLEX [20] and Marabou [21] provide specialized versions of the SIMPLEX method to boost scalability of the formal verification phase.

However, verification alone can only give a yes or no answer and cannot *synthesize* a neural network that is guaranteed to respect the property when the answer to the verification problem is negative. A few attempts in the literature to deal with the synthesis of certified neural networks have been provided. Some of them are restricted to general monotonicity of the network [25, 29], whereas others are limited to constraining the output for Hierarchical Multi-label classification Problems [11, 12, 15]. Considering more general rules, some works [23, 24, 33, 36, 35] propose to inject constraints in the loss function of a neural network in such a way that the training of the model should prefer solutions where the constraints are satisfied. [26] generates counterexamples (solving an optimization problem) that violate some constraints that are expressed in first-order logic (FOL) and include them in the training set in NLP tasks. [10] expresses constraints as logical clauses directly incorporated in the network structure as an additional layer that adds trainable parameters (clause weights) that represent the satisfiability level of constraints, providing explainability to the model as well as increased generalization. Some other methods additionally provide guarantees on the considered constraints, even though they either pose limitations on the constraints [18, 36], or rely on an external component to enforce the desired properties at run time [3, 9, 38]. Finally, a recent review of deep learning with logical constraints is provided in [16].

The problem of automatically generating a computational system that provably satisfies a given high-level specification has been studied in computer since the seminal works of Church [8]. One relevant methodology developed by this line of research is the Counter Example Guided Inductive Synthesis (CEGIS) workflow, initially proposed in [30] for the automated synthesis of programs. The CEGIS workflow has been recently applied to prove stability of neural controllers [1, 6] and to generate invariants for dynamical systems [7, 28]. While these works exploit a learning-verify loop similar to our approach, they start from an analytical description of the dynamical system, and they consider only the problem of learning a network that satisfies the desired properties. In our setting we start from a dataset sampled from an unknown distribution, and we aim to preserve the accuracy of the network as much as possible. To do so, we develop a data generation technique to augment the training data that turns out to be crucial to keep the final network as close as possible to the one trained on the unknown data-generating distribution.

* Corresponding Author. Email: matteo.zavatteri@unipd.it
** Corresponding Author. Email: davide.bresolin@unipd.it
*** Corresponding Author. Email: nicolo.navarin@unipd.it

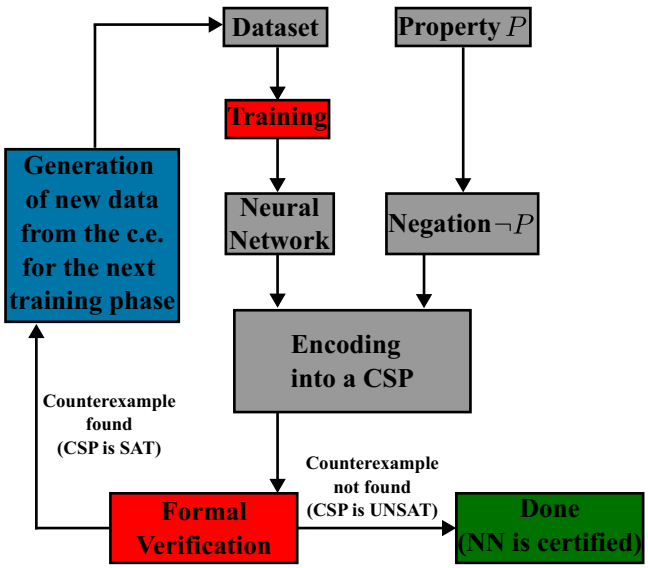


Figure 1. High level view of the CEGIS workflow employed for the synthesis of certified neural networks.

Contribution and organization

This paper combines Deep Learning with Formal Methods and employs the Counter Example Guided Inductive Synthesis (CEGIS) workflow for the automated synthesis of certified neural networks. Section 2 provides a general description of the synthesis workflow and identifies, in a technology-independent way, the modules that we need to obtain our goal. Section 3 sums up how to encode a ReLU neural network into a first-order linear real arithmetic formula (LRA). Section 4 identifies a fragment of LRA to express properties and formally state the problem we want to solve. Section 5 discusses how to generate new data starting from the counterexamples found by the verification phase when the network does not respect the property and how to add the new data to the training set for the subsequent training phase. Sections 6 and 7 discuss a proof-of-concept implementation of our workflow and show how to certify two properties on an existing dataset from a robotic scenario, and for different learning rates and structure of the network (number of hidden layers and nodes per layer). Section 8 concludes and discusses future work.

2 CEGIS workflow

The general framework that we use in this paper is based on the Counter Example Guided Inductive Synthesis (CEGIS) workflow shown in Figure 1. CEGIS was initially proposed in [30] for the automated synthesis of programs. In general, CEGIS works by generating a candidate solution for the problem at hand (a program in [30], a neural network in our case) and then formally verifying whether the candidate solution respects the desired properties or not. If not, the formal verification phase generates one or more counterexamples that are used to generate a new, better, solution, until the candidate solution is proved to respect the required properties. The CEGIS workflow is thus not a new concept. What is new here is how we set up CEGIS for the neural context and how we use the counterexamples to generate new data with which we add to the training set until a certified neural network is produced.

Our CEGIS approach starts from a dataset D of input-output pairs disjointly partitioned in training, validation, and test set. We then create a neural network N as a result of some training algorithm that works on training and validation sets only. After that, we encode N into a suitable constraint satisfaction problem (CSP) along with the negation of the property P that we want N to satisfy. The output of the formal verification part can lead to two situations:

1. the CSP is unsatisfiable; this means that no counterexample exists and thus N is certified for P;
2. the CSP is satisfiable; this means that we found a counterexample: an input $\mathbf{x}$ where the output $N(\mathbf{x})$ does not satisfy P.

In the former case, the process is completed and we have proof that N respects the property P *for all possible values in the input domain*, and not only for the inputs contained in the (finite) dataset D. In the latter case, we use the counterexample $(\mathbf{x}, N(\mathbf{x}))$ to extend the training set with new input-output pairs $(\mathbf{x}', \mathbf{y}')$ that satisfy P, before proceeding with the next iteration of the CEGIS workflow. These new pairs are obtained by first replacing the output of the network in the counterexample with an output $\mathbf{y}$ such that $(\mathbf{x}, \mathbf{y})$ satisfies the property, and then sampling around it to increase the number of new data points that are added to the training set.

3 Encoding ReLU neural networks

A directed weighted graph $N = (V, E, w)$ consists of a set of nodes V, a set of directed edges E, and a weight function $w\colon E \mapsto \mathbb{R}$ associating real numbers to edges. A feedforward ReLU neural network is built on top of a directed weighted graph by imposing restrictions on the form of the graph as well as adding further characteristics to the nodes as follows.

- The network always includes an *input layer*, an *output layer*, and may include *hidden layers* in between. A network with exactly one hidden layer is *shallow*; it is *deep* if there exist at least two hidden layers. We denote with d the number of layers in the network. Layer 1 is the input layer, while layer d is the output layer. Layers $2, \ldots, d-1$ are the hidden layers.
- A layer is a vector of *nodes*. We denote with ℓ_i the size of the i-th layer, that is, the number of nodes in the layer. The j-th node of layer i is denoted by $v_{i,j}$. To ease readability, we omit the subscript j when the corresponding layer consists of only one node.
- Every non-output node is connected to all nodes in the following layer through weighted edges, whereas every non-input node is connected to all nodes in the preceding layer through weighted edges.
- Hidden nodes serve as computational units. Every hidden node $v_{i,j}$ also has an associated *bias* $b_{i,j}$ and a non-linear *activation function*. The node computes a linear function over the output of the nodes in the previous layer, and then it applies the activation function to generate its output. In this paper, we consider the following activation function: $\mathrm{ReLU}(x) := \max(0, x)$.
- Output nodes also have associated biases but no activation functions. In this way, they compute a linear combination of the nodes in the last hidden layer. Networks with activation functions in the output layer (e.g., a sigmoid for binary outputs) can be certified as well if the property can be rewritten to consider only the input of the activation function.

Figure 2 gives an example of a small neural network with a single input node $v_{1,1}$, a single output node $v_{3,1}$, and a hidden layer containing 2 nodes $v_{2,1}, v_{2,2}$ with ReLU activation functions. Biases are shown above the nodes, and weights are above the edges. The

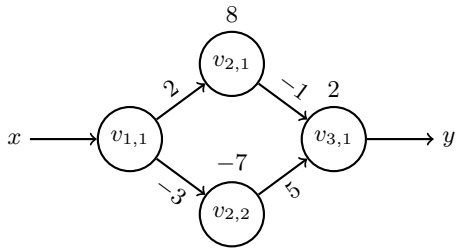


Figure 2. A neural network with one hidden layer.

network computes the function:

$$N(x) := -\overbrace{(\max(0, 2x + 8))}^{\text{computed by } v_{2,1}} + 5\overbrace{(\max(0, -3x - 7))}^{\text{computed by } v_{2,2}} + 2$$

where x is the single input of the network.

Taking inspiration from the encoding used in [20] to verify neural networks and in [34] to solve planning problems, we use the theory of linear real arithmetic (LRA) to encode the network and the properties. LRA formulas are first-order logic formulas built from atoms that are linear constraints $\sum_{i=1}^{n} cx_i \bowtie b$, where c, b are rational constants, and $\bowtie \in \{<, \leq, =, >, \geq, \neq\}$.

Let N be a ReLU neural network with d layers, n inputs represented as the vector $\mathbf{x} = (x_1, \ldots, x_n) \in \mathcal{X}$, and m outputs represented as the vector $\mathbf{y} = (y_1, \ldots, y_m) \in \mathcal{Y}$. To encode the network into an LRA formula we proceed as follows.

For each node $v_{i,j}$ of N we add two auxiliary variables $v_{i,j}^{in}$ and $v_{i,j}^{out}$, that represent respectively the input and the output of the node. To improve readability, we use $x_1, \ldots, x_n$ instead of the auxiliary variables $v_{1,1}^{in}, \ldots, v_{1,n}^{in}$ for the input layer, and $y_1, \ldots, y_m$ instead of the auxiliary variables $v_{d,1}^{out}, \ldots, v_{d,m}^{out}$ for the output layer.

Input nodes simply forward the input to the output. Hence, for each input node we add the conjunct:

$$x_j = v_{1,j}^{out}$$

For each hidden node $v_{i,j}$, where $1 < i < d$, we add the following conjuncts:

$$v_{i,j}^{in} = b_{i,j} + \sum_{k=1}^{\ell_{i-1}} w(v_{i-1,k}, v_{i,j}) \cdot v_{i-1,k}^{out}$$
$$v_{i,j}^{out} = \max(0, v_{i,j}^{in})$$

where $a = \max(0, b)$ can be encoded in LRA as:

$$a \geq 0 \wedge a \geq b \wedge (a = 0 \vee a = b).$$

For each output node $v_{d,j}$, we add the following conjuncts:

$$v_{d,j}^{in} = b_{d,j} + \sum_{k=1}^{\ell_{d-1}} w(v_{d-1,k}, v_{d,j}) \cdot v_{d-1,k}^{out}$$
$$y_j = v_{d,j}^{in}$$

Finally, the LRA formula $F_N(\mathbf{x}, \mathbf{y})$ that encodes the network N is obtained by conjoining all equations above and then existentially quantifying on all auxiliary variables $v_{i,j}^{in}$ and $v_{i,j}^{out}$. The resulting formula has only input variables $x_1, \ldots, x_n$ and output variables $y_1, \ldots, y_m$ as free, and it is true if and only if $\mathbf{y} = N(\mathbf{x})$.

The LRA encoding of Figure 2 is the following formula:

$$\begin{aligned} F_N(x, y) := \exists v_{1,1}^{out}, v_{2,1}^{in}, v_{2,1}^{out}, v_{2,2}^{in}, v_{2,2}^{out}, v_{3,1}^{in} \big(& x = v_{1,1}^{out} \wedge \\ & v_{2,1}^{in} = 8 + 2v_{1,1}^{out} \wedge v_{2,1}^{out} = \max(0, v_{2,1}^{in}) \wedge \\ & v_{2,2}^{in} = -7 - 3v_{1,1}^{out} \wedge v_{2,2}^{out} = \max(0, v_{2,2}^{in}) \wedge \\ & v_{3,1}^{in} = 2 - v_{2,1}^{out} + 5v_{2,2}^{out} \wedge y = v_{3,1}^{in} \big) \end{aligned}$$

4 Constraint language and formal verification

In this work, we use LRA not only to encode neural networks but also to specify properties. The formulae expressing the properties of a network N that we consider have the form $F_{pre}(\mathbf{x}) \Rightarrow F_{post}(\mathbf{x}, \mathbf{y})$, where $F_{pre}(\mathbf{x})$ is a quantifier-free LRA formula over input variables only, and $F_{post}(\mathbf{x}, \mathbf{y})$ is a quantifier-free LRA formula over both input and output variables. In this setting, a network is certified for a property if for all inputs where the precondition $F_{pre}(\mathbf{x})$ is satisfied, then the output of the network $\mathbf{y} = N(\mathbf{x})$ is such that the postcondition $F_{post}(\mathbf{x}, \mathbf{y})$ is satisfied as well. More formally,

Definition 1. *A ReLU neural network N is* certified *for a property $P := F_{pre}(\mathbf{x}) \Rightarrow F_{post}(\mathbf{x}, \mathbf{y})$ if the following formula is valid:*

$$F_{N,P}(\mathbf{x}, \mathbf{y}) := (F_N(\mathbf{x}, \mathbf{y}) \wedge F_{pre}(\mathbf{x})) \Rightarrow F_{post}(\mathbf{x}, \mathbf{y}).$$

Since $F_{pre}(\mathbf{x})$ and $F_{post}(\mathbf{x}, \mathbf{y})$ are arbitrary quantifier-free formulas, limiting properties to the form $F_{pre}(\mathbf{x}) \Rightarrow F_{post}(\mathbf{x}, \mathbf{y})$ does not restrict the expressiveness of our framework, as any arbitrary LRA formula on $(\mathbf{x}, \mathbf{y})$ can be rewritten to respect the implicative form. We say that a property P is *realizable* if there exists a function $G : \mathcal{X} \mapsto \mathcal{Y}$ such that $F_{pre}(\mathbf{x}) \Rightarrow F_{post}(\mathbf{x}, G(\mathbf{x}))$ is true for all $\mathbf{x} \in \mathcal{X}$. From now on, we consider only realizable properties.

Problem statement. Given:

- a dataset D of input-output pairs disjointly partitioned in training, validation, and test, and
- a realizable property P defined as above,

synthesize a ReLU neural network N such that:

- N is certified for P, and
- the accuracy of N on the test set is preserved as much as possible with respect to the accuracy of a (non necessarily certified) network resulting from the training on the initial D only.

An LRA formula with free variables is valid if and only if its negation is unsatisfiable. Hence, to check if N is certified for a property P, we can equivalently check whether $\neg F_{N,P}(\mathbf{x}, \mathbf{y})$ is unsatisfiable. Written more explicitly:

$$\begin{aligned} \neg F_{N,P}(\mathbf{x}, \mathbf{y}) &:= \neg ((F_N(\mathbf{x}, \mathbf{y}) \wedge F_{pre}(\mathbf{x})) \Rightarrow F_{post}(\mathbf{x}, \mathbf{y})) \\ &\equiv F_N(\mathbf{x}, \mathbf{y}) \wedge F_{pre}(\mathbf{x}) \wedge \neg F_{post}(\mathbf{x}, \mathbf{y}) \end{aligned}$$

is an LRA formula with free variables $\mathbf{x}$ and $\mathbf{y}$, whose set of solutions matches the set of counterexamples proving that N falsifies P.

To give a simple example, consider the following property P on N of Figure 2 in natural language: *an input less or equal than* 0 *always implies an output less or equal than* 0. More formally,

$$F_{N,P}(x, y) := (F_N(x, y) \wedge \overbrace{x \leq 0}^{F_{pre}(x)}) \Rightarrow \overbrace{y \leq 0}^{F_{post}(x,y)}$$

N is not certified for P, since the formula

$$\neg F_{N,P}(x, y) := F_N(x, y) \wedge x \leq 0 \wedge y > 0$$

is satisfiable. By letting α be an assignment that associates a real value to each of the variables in $\mathbf{x}$ and $\mathbf{y}$ to make $\neg F_{N,P}(\mathbf{x}, \mathbf{y})$ true, one possible solution is $\alpha(x) = -3, \alpha(y) = 10$ since $N(-3) = 10$.

5 Counterexample repair for data generation

Let N be a ReLU neural network and P be a property on N. Suppose that $\neg F_{N,P}(\mathbf{x}, \mathbf{y})$ is satisfiable and let α be a satisfying assignment. To correct the behavior of the network, we want to build another assignment α' such that:

1. the value assignments to the input variables do not change: for each $i = 1, \ldots, n$, it holds that $\alpha'(x_i) = \alpha(x_i)$;
2. the new assignment α' makes $F_{post}(\mathbf{x}, \mathbf{y})$ true.

To build α', we proceed as follows. Let $\mathbf{s} = (s_1, \ldots, s_m)$ be a vector of fresh real variables not appearing in $F_{post}(\mathbf{x}, \mathbf{y})$, and let $F'_{post}(\mathbf{s})$ be the quantifier-free LRA formula defined as follows:

$$F'_{post}(\mathbf{s}) := F_{post}(\mathbf{x}, \mathbf{y})[\alpha(x_1)/x_1, \ldots, \alpha(x_n)/x_n, \\ \alpha(y_1) + s_1/y_1, \ldots, \alpha(y_m) + s_m/y_m]$$

where $F[t_1/x_1, \ldots, t_k/x_k]$ is the simultaneous multiple substitution that replaces every occurrence of the free variable x_i with the term t_i. In our example,

$$F'_{post}(s) := F_{post}[\alpha(x)/x, \alpha(y) + s/y] = 10 + s \leq 0$$

Now, any assignment β to the variables $\mathbf{s}$ that makes $F'_{post}(\mathbf{s})$ true defines a corresponding assignment α' to the variables $\mathbf{x}$ and $\mathbf{y}$ as follows:

1. For each $i = 1, \ldots, n$, let $\alpha'(x_i) = \alpha(x_i)$;
2. For each $i = 1, \ldots, m$, let $\alpha'(y_i) = \alpha(y_i) + \beta(s_i)$.

In our example, any $\beta(s) \leq -10$ is fine as $10 + \beta(s) \leq 0$. Clearly, α' makes the original postcondition formula F_{post} true. Thus, we can add the new input-output pair:

$$(\alpha'(x_1), \ldots, \alpha'(x_n), \alpha'(y_1), \ldots, \alpha'(y_m))$$

to the current dataset D, and start over the training phase with this new information. If $\beta(x) = -10$, then we will add the pair $(\alpha(x), \alpha(y) + \beta(s)) = (-3, 0)$ to D. However, any β assigning a value less or equal to -10 to s still produces a new pair that respects the postcondition.

We would like to generate an α' such that the loss of the certified N on the test set is preserved as much as possible with respect to the loss that a non certified network, trained on D, would have on the same test set. To this end, we use as new data point an α' that minimize the distance from the counterexample α. Such an α' can be computed by solving the following quadratic programming problem (QP):

$$\min \sum_{i=1,\ldots,m} s_i^2 \qquad \text{subject to} \qquad F'_{post}(\mathbf{s})$$

In this way we generate a new data point $(\mathbf{x}, \mathbf{y}')$ by selecting a $\mathbf{y}'$ that is at minimal distance from the original $\mathbf{y}$ and such that the property is respected. In fact, the purpose of the minimization problem is to generate an example as close as possible to the original data distribution while respecting property P, that is, to minimize the changes needed to certify the network.

In our example, we solve

$$\min s^2 \qquad \text{subject to} \qquad 10 + s \leq 0$$

The optimal solution is $\beta(s) = -10$, leading to:

$$\alpha'(x) = \alpha(x) = -3$$
$$\alpha'(y) = \alpha(y) + \beta(s) = 0$$

and thus $(-3, 0)$. Finally, to speed up the next training phase we generate further points to add to D by sampling around α' and adding a sample to the dataset D only if it satisfies $F_{post}(\mathbf{x}, \mathbf{y})$. This way we avoid solving multiple QP problems to generate additional data.

Termination. The overall goal of the CEGIS loop is to synthesize a function that respects a given property. Since there are uncountably many possible inputs for the function, the CEGIS loop may go on for a very long time (or even forever), producing more and more counterexamples, even when a function respecting the property exists. Since the property is a quantifier-free LRA formula, we know we can restrict the search to piecewise-linear functions. We know that ReLU neural networks compute piecewise linear functions, and thus, from recent universal approximation results for ReLU neural networks [27], we can conclude that a network big enough to respect the property exists. Well established results from probably approximately correct (PAC) learning prove that in our setting the sample complexity of the learning problem is finite [4, Theorem 21.5], and thus that the CEGIS loop terminate with high probability. We believe that for some subclasses of constraints the approach is guaranteed to converge with a number of iterations that depends on the expressiveness of the network and on the size of the property. We are currently carrying out an investigation on this theoretical part, that is left for future publications.

6 Mind your manners, robot!

We applied our framework to the MANNERS-DB example [32]. The case study involves a robot moving in a room where humans and animals are present. The controller is implemented through a neural network. The description of the inputs and outputs are given in Table 1. In particular, the outputs of the neural network provide numbers representing the adequacy of the possible actions executable by the robot. Each adequacy is a score of 1 to 5, where the bigger the number, the more adequate the action. The adequacy scores have been computed as the average across human judgments obtained via crowdsourcing (see [32]). The original dataset comprises 11050 input-output pairs.

We normalized each input and output to make it fall in $[0, 1]$. When some inputs are not applicable, such as distance to the closest animal when no animal is present or group distance when no group exists, we normalized their value to 1. We tested our workflow on 5 different ReLU, where each network has 28 inputs and 8 outputs and a different combination of number of hidden layers and nodes per layer. We considered the following networks (we highlight in bold the identifiers that we also use in the rest of the paper):

6x9: 6 hidden layers with 9 nodes each;
7x8: 7 hidden layers with 8 nodes each;
8x7: 8 hidden layers with 7 nodes each;
9x6: 9 hidden layers with 6 nodes each;
10x10: 10 hidden layers with 10 nodes each.

We randomly partitioned the dataset in train, validation, and test so that they contain 8840 (i.e., 80%), 1105 (i.e., 10%), and 1105 (i.e., 10%) pairs, respectively. We used the Adam optimizer [22] with learning rates of 0.01, 0.001, and 0.0001 and mean square error (MSE) loss function to train the networks on the normalized data for 100 epochs and by dividing the data in batches of size 100 each. We keep the network with minimal loss on the validation set among the epochs, by applying an early stopping criterion of 20 epochs and we evaluate its accuracy (at the end) on the test set. Every CEGIS

Input	Description	Type	Range
x_1	operating mode	INT	$[0, 1]$
x_2	number of people	INT	$[0, 9]$
x_3	number of people in group	INT	$[0, 5]$
x_4	group radius	FLOAT	$[0.5, 1]$
x_5	distance to group	FLOAT	$[0, 6]$
x_6	robot within group	INT	$[0, 1]$
x_7	robot facing group	INT	$[0, 1]$
x_8	robot work radius	FLOAT	$[0, 3]$
x_9	distance to closest human	FLOAT	$[0.3, 5]$
x_{10}	distance to 2nd closest human	FLOAT	$[0.3, 5]$
x_{11}	distance to 3rd closest human	FLOAT	$[0.3, 5]$
x_{12}	direction to closest human	FLOAT	$[0, 360]$
x_{13}	direction to 2nd closest human	FLOAT	$[0, 360]$
x_{14}	direction to 3rd closest human	FLOAT	$[0, 360]$
x_{15}	direction from closest human to robot	FLOAT	$[0, 360]$
x_{16}	robot facing closest human	INT	$[0, 1]$
x_{17}	robot facing 2nd closest human	INT	$[0, 1]$
x_{18}	robot facing 3rd closest human	INT	$[0, 1]$
x_{19}	closest human facing robot	INT	$[0, 1]$
x_{20}	2nd closest human facing robot	INT	$[0, 1]$
x_{21}	3d closest human facing robot	INT	$[0, 1]$
x_{22}	number of children	INT	$[0, 2]$
x_{23}	distance to closest child	FLOAT	$[0.4, 6]$
x_{24}	number of animals	INT	$[0, 1]$
x_{25}	distance to closest animal	FLOAT	$[0.4, 6]$
x_{26}	number of people on sofa	INT	$[0, 2]$
x_{27}	music playing	INT	$[0, 1]$
x_{28}	number of agents in scene	INT	$[1, 11]$
Output	**Description**	**Type**	**Range**
y_1	vacuum cleaning	INT	$[1, 5]$
y_2	mopping the floor	INT	$[1, 5]$
y_3	carry warm food	INT	$[1, 5]$
y_4	carry cold food	INT	$[1, 5]$
y_5	carry drinks	INT	$[1, 5]$
y_6	carry small objects	INT	$[1, 5]$
y_7	carry big objects	INT	$[1, 5]$
y_8	cleaning or starting conversation	INT	$[1, 5]$

Table 1. Input and outputs of the MANNERS-DB neural network.

iteration starts over with a new network whose weights are randomly initialized.

We considered two properties that are examples of behavioral constraints on the manners of the robot.

P_1**:** "*if the distance to the closest child is within 0.6 (meters), then mopping the floor must be the less adequate action*";

P_2**:** "*if the distance to the closest human is within 0.6 (meters), then the adequateness of mopping the floor must be less than or equal to the adequateness of mopping the floor in all possible scenarios in which the distance to the closest animal is within 0.6 (meters)*".

In this section we discuss the first property only. To formalize the second property we need to extend our property language as we will show in the next section.

The first property is formalized by the following formula:

$$P_1(\mathbf{x}, \mathbf{y}) := \overbrace{\Big(\Big(\bigwedge_{i=1}^{28} 0 \leq x_i \leq 1\Big) \wedge x_{23} \leq 0.1\Big)}^{F_{pre}(\mathbf{x})} \Rightarrow \overbrace{\bigwedge_{i=1}^{8} y_2 \leq y_i}^{F_{post}(\mathbf{x},\mathbf{y})}$$

The formula F_{pre} restricts the inputs of the network to be inside the normalized interval $[0, 1]$, and formalizes the constraint "distance to the closest child within 0.6 meters" by forcing the corresponding input variable x_{23} to be less or equal to 0.1 (i.e., 0.6 normalized). We set up the verification phase to produce up to 50 counterexamples if N does not respect P_1. For each counterexample α we compute α' by solving the quadratic problem:

$$\min \sum_{i=1}^{8} s_i^2 \quad \text{subject to} \quad \bigwedge_{i=1}^{8} \alpha(y_2) + s_2 \leq \alpha(y_i) + s_i$$

and then sample 20 additional points by adding a noise vector to the repaired counterexample with mean of 0 and a variance and standard deviation of 1. Notice that among all the sampled points, we keep only those which satisfy the constraint. Thus, that hyperparameter to obtain the noise vector has a (usually small) impact on the ratio of counterexamples actually added for each iteration, but it does not affect the convergence of the method. Thus, at each iteration of the CEGIS workflow we can add up to 1050 new pairs to the dataset.

We implemented our approach in Python 3, using PyTorch 2.1.1 as the deep learning framework and Gurobi 11 [17] to solve the various constraint satisfaction problems (see [37] for the source code). Gurobi can also produce multiple solutions to constraint satisfaction problems, a functionality that we exploited to generate multiple counterexamples. We run our software on a High Performance Computing (HPC) center using used a node equipped 2 Intel(R) Xeon(R) CPU E5-2650 v3 2.30GHz, 1 GPU Nvidia T4 (cudadrv495), 160GB of RAM, and running Ubuntu 20.04.3 LTS. The HPC is handled by the SLURM workload manager [19]. We assigned the corresponding job 20 CPU cores, 150GB of RAM, and the whole GPU of the node.

Table 2 provides a summary of the experimental results. For each combination of property, network structure and learning rate, the table lists the number of iterations needed to certify the network, the total computation time, and the time spent on training, verification and sampling (in seconds). It also provides some statistics on the quality of the final network by comparing the loss value of the initial network with the loss value of the final network, and by showing the total number of new pairs added to the dataset. Each experiment is intended with respect to one property, one network, one learning rate.

The data is aligned with property P_1 and P_2. Since the properties are implications, most of the data satisfies them trivially by falsifying the precondition. 98% of points satisfies P_1 trivially, 0.5% satisfies P_1 by respecting the post-condition, and 1.5% violates the property. For property P_2 we have that 94.5% of points satisfies the property trivially, 2.7% satisfies the post-condition and 2.7% violates P_2.

In the case of property P_1, we were able to certify all combinations of network, for every learning rate, with a relatively small number of iterations and computation time. The MSE loss of the initial network is between 0.059 and 0.066, while the MSE loss of the final network ranges between 0.061 and 0.067, showing that certifying P_1 has little impact on the MSE loss on the test set. Moreover, the experiments shows that changing the number of hidden layers does not seem to affect the final loss value, nor the computation time.

7 Extended constraint language

As we already pointed out, the second property is not expressible with the language proposed in Section 4. To formalize P_2 we need to consider the output of the network on two different input vectors: one that represents a scenario where the closest human is within 0.6 meters and one for a scenario where the closest animal is within 0.6 meters, and then check that the adequateness of mopping the floor in the first scenario is less or equal to the adequateness in the second scenario. This can be formalized by extending the formula used

Property	Network	Learning Rate	CEGIS Iterations	Total Time (s)	Training Time (s)	Verification Time (s)	Repair + Sampling Time (s)	Initial Loss (test)	Final Loss (test)	New Pairs
P_1	6x9	0.01	30	1334	1273	11.222	0.868	0.06	0.067	8178
		0.001	65	5821	5683	33.163	1.89	0.059	0.067	20826
		0.0001	34	2827	2729	17.602	1.031	0.062	0.062	10366
	7x8	0.01	35	1290	1229	9.691	1.052	0.066	0.067	8839
		0.001	16	858	825	9.313	0.443	0.059	0.066	4763
		0.0001	21	1571	1520	8.227	0.603	0.066	0.062	6229
	8x7	0.01	25	1053	1017	4.821	0.709	0.06	0.066	5542
		0.001	22	1360	1317	9.1	0.634	0.06	0.066	6419
		0.0001	19	1461	1424	6.848	0.533	0.062	0.067	6314
	9x6	0.01	23	1085	1049	3.416	0.664	0.061	0.066	5422
		0.001	34	1972	1922	7.684	1.029	0.06	0.067	6793
		0.0001	20	1617	1581	4.733	0.564	0.064	0.067	5986
	10x10	0.01	21	855	805	22.499	0.636	0.066	0.066	4791
		0.001	41	8863	3015	5771.98	1.243	0.06	0.061	12345
		0.0001	17	1517	1449	38.539	0.492	0.062	0.067	5468
P_2	6x9	0.01	9	348	329	3.404	0.332	0.059	0.066	6634
		0.001	11	753	721	8.15	0.416	0.059	0.078	9948
		0.0001	136	51371	51005	91.485	5.586	0.06	0.161	118480
	7x8	0.01	2	43	41	0.13	0.04	0.061	0.066	74
		0.001	97	23922	23599	108.857	3.817	0.06	0.113	97192
		0.0001	8	617	590	5.639	0.285	0.063	0.077	6670
	8x7	0.01	7	258	245	1.822	0.241	0.06	0.066	4456
		0.001	7	454	438	2.717	0.242	0.06	0.068	5448
		0.0001	4	277	266	1.799	0.121	0.063	0.07	3242
	9x6	0.01	2	75	72	0.119	0.041	0.061	0.066	126
		0.001	6	375	361	2.428	0.202	0.061	0.068	5666
		0.0001	5	370	360	1.46	0.165	0.061	0.07	3306
	10x10	0.01	2	88	86	0.276	0.043	0.061	0.066	14
		0.001	8	574	531	28.639	0.291	0.06	0.068	5606
		0.0001	62	9354	9035	247.04	2.889	0.062	0.105	29342

Table 2. Summary of the experimental results. **Network** shows the size of the hidden part given in (LAYERS)x(NODES_PER_LAYER).

in Definition 1 with two copies of the network (one for each input-output pair), and by allowing F_{pre} to be defined on the two inputs and F_{post} on the two input-output pairs:

$$F_{N,P_2}(\mathbf{x},\mathbf{y},\mathbf{x}',\mathbf{y}') := \Big(F_N(\mathbf{x},\mathbf{y}) \wedge F_N(\mathbf{x}',\mathbf{y}') \wedge \underbrace{\Big(\bigwedge_{i=1}^{28} 0 \le x_i \le 1 \wedge 0 \le x_i' \le 1\Big) \wedge x_9 \le 0.1 \wedge x_{25}' \le 0.1\Big)}_{F_{pre}(\mathbf{x},\mathbf{x}')} \Rightarrow \underbrace{y_2 \le y_2'}_{F_{post}(\mathbf{x},\mathbf{y},\mathbf{x}',\mathbf{y}')}$$

where $(\mathbf{x},\mathbf{y})$ is the input-output pair of the first copy of the network and $(\mathbf{x}',\mathbf{y}')$ is the input-output pair of the second copy. With the new encoding, every counterexample is a pair of data points (one for each copy of the network). When we repair the counterexample by solving the quadratic problem:

$$\begin{aligned} &\min \left(\sum_{i=1}^{8} s_i^2 + \sum_{i=1}^{8} (s_i')^2\right) \\ &\text{subject to} \quad \alpha(y_2) + s_2 \le \alpha(y_2') + s_2' \end{aligned}$$

we obtain two new data points as well. As for the first property, we sample around the new data points to add up to 2100 new points to the dataset.

Table 2 pictures a different situation for P_2 compared to P_1. In this case certifying the property requires very few iterations, with little impact on the loss, except for some combinations of network structure and learning rate where certifying the property may require up to 136 iterations and more than 14 hours of computation. Also, the MSE loss of the final certified network increases significantly in these cases (it more than doubles in the worst case).

Figure 3 shows a detailed analysis of the experiments for the 8x7 network and learning rate 0.001. Figures 3(a) and 3(b) compare the training time with the verification time for each iteration of the CEGIS workflow, for property P_1 and P_2. The graphs are in y-logarithmic scale, and show that the training time is currently the boottleneck of the approach, since it is at least one order of magnitude greater than the verification time.

Figures 3(c) and 3(d) show the number of counterexamples generated by the verification phase and the number of new pairs added to the dataset after the sampling phase. The number of counterexamples is equal to the maximum of 50 in almost all iterations, except for one iteration for property P_1, one iteration for property P_2, and the final iterations (where no counterexample exist since the network now respect the property). For property P_1, the number of new pairs ranges from 14 to 386, while for property P_2 it ranges from 22 to 1132.

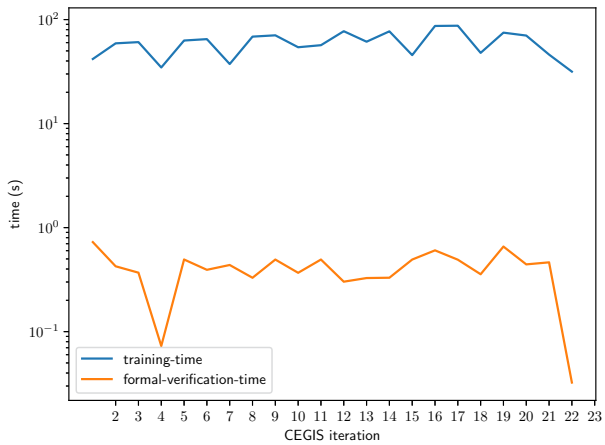
8 Conclusions and future work

We proposed a method based on the CEGIS workflow to automatically synthesize certified neural networks. The certification part is based on formal verification, whereas the overall synthesis exploits information on counterexamples to repair them and build new input-output pairs to add to the current dataset. An important point of our approach is that it is modular, meaning that our approach can be applied off-the-shelf to other kinds of neural networks and properties provided there exist a formal verification and a counterexample repair part able to handle the corresponding constraints.

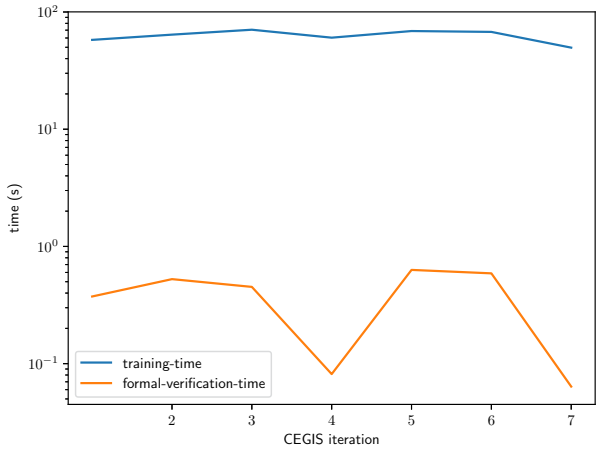
As future work, we will consider more complex tasks, e.g., reinforcement learning settings and, consequently, larger neural networks and more complex properties. We will also study if it is possible to extend the applicability of the proposed framework to enforce *fairness* constraints, and we will explore to what extent the examples generated to certify a specific network architecture can be reused in other architectures. Finally, we will exploit methods to inject soft constraints in the loss function to reduce the number of iterations required to generate a certified network, and to improve the scalability of the framework.

Acknowledgements

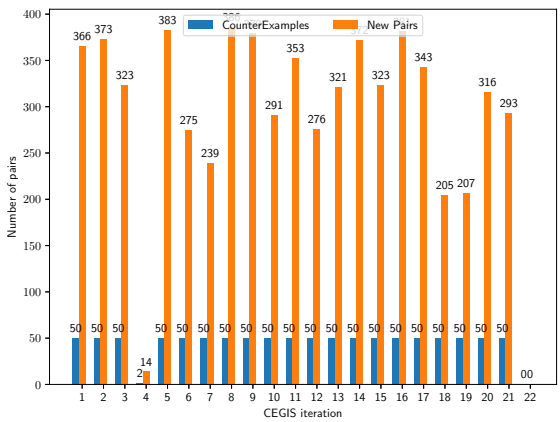
This work was supported by the National Recovery and Resilience Plan (NRRP), Mission 4 Component 2 Investment 1.5 - Call for tender No. 3277 of 30 December 2021 of Italian Ministry of University and Research funded by the European Union – NextGenerationEU; project code: ECS00000043, Concession Decree No. 1058 of June



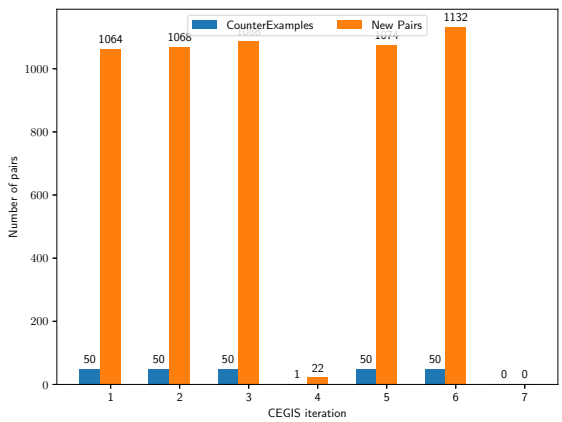
(a) Training and formal verification time.



(b) Training and formal verification time.



(c) New added pairs.



(d) New added pairs.

Figure 3. Certification process with for P_1 (left) and P_2 (right). Network 8x7, learning rate 0.001. Plots (a) and (b) are in y-log scale.

23, 2022, CUP C43C22000340006, project title "iNEST: Interconnected Nord-Est Innovation Ecosystem", and by Mission 4 Component 2 Investment 1.3 - Call for tender No. 341 of March 15, 2022 of Italian Ministry of University and Research – NextGenerationEU; project code PE0000013, Concession Decree No. 1555 of October 11, 2022, CUP C63C22000770006, project title "Future AI Research (FAIR) - Spoke 2 Integrative AI - Symbolic conditioning of Graph Generative Models (SymboliG)". This work was also supported by the Projects iNEST Young Researchers "Neuro-Symbolic AI in digital twins", and by the INdAM-GNCS 2024 Project "Certificazione, monitoraggio, ed interpretabilità in sistemi di intelligenza artificiale" CUP_E53C23001670001.

References

[1] A. Abate, D. Ahmed, M. Giacobbe, and A. Peruffo. Formal synthesis of lyapunov neural networks. *IEEE Control. Syst. Lett.*, 5(3):773–778, 2021.

[2] A. Albarghouthi. Introduction to neural network verification. *Found. Trends Program. Lang.*, 7(1-2):1–157, 2021.

[3] M. Alshiekh, R. Bloem, R. Ehlers, B. Könighofer, S. Niekum, and U. Topcu. Safe reinforcement learning via shielding. In *AAAI 2018*, pages 2669–2678. AAAI Press, 2018.

[4] M. Anthony and P. L. Bartlett. *Neural Network Learning: Theoretical Foundations*. Cambridge University Press, 1999.

[5] C. Brix, M. N. Müller, S. Bak, T. T. Johnson, and C. Liu. First three years of the international verification of neural networks competition (VNN-COMP). *Int. J. Softw. Tools Technol. Transf.*, 25(3):329–339, 2023.

[6] Y. Chang, N. Roohi, and S. Gao. Neural lyapunov control. In *NeurIPS*, pages 3240–3249, 2019.

[7] K. Chatterjee, T. A. Henzinger, M. Lechner, and D. Zikelic. A learner-verifier framework for neural network controllers and certificates of stochastic systems. In *TACAS (1)*, volume 13993 of *Lecture Notes in Computer Science*, pages 3–25. Springer, 2023.

[8] A. Church. Logic, arithmetic, and automata. *Journal of Symbolic Logic*, 29(4):210–210, 1964. doi: 10.2307/2270398.

[9] C. Cornelio, J. Stuehmer, S. X. Hu, and T. M. Hospedales. Learning where and when to reason in neuro-symbolic inference. In *ICLR 2023*. OpenReview.net, 2023.

[10] A. Daniele and L. Serafini. Knowledge enhanced neural networks for relational domains. In *AIxIA 2022*, volume 13796 of *LNCS*, pages 91–109. Springer, 2022.

[11] J. Deng, N. Ding, Y. Jia, A. Frome, K. Murphy, S. Bengio, Y. Li, H. Neven, and H. Adam. Large-scale object classification using label relation graphs. In *ECCV 2014*, volume 8689 of *LNCS*, pages 48–64. Springer, 2014.

[12] P. Dragone, S. Teso, and A. Passerini. Neuro-symbolic constraint programming for structured prediction. In *(IJCLR 2021)*, volume 2986 of *CEUR Workshop Proceedings*, pages 6–14. CEUR-WS.org, 2021.

[13] EASA and Collins Aerospace. Formal methods use for learning assurance (ForMuLA). Technical report, Apr. 2023. URL https://www.easa.europa.eu/en/downloads/137878/en.

[14] European Parliament. Artificial intelligence act, 2024. URL

https://eur-lex.europa.eu/legal-content/EN/TXT/HTML/?uri=CELEX:52021PC0206.
[15] E. Giunchiglia and T. Lukasiewicz. Multi-label classification neural networks with hard logical constraints. *JAIR*, 72:759–818, 2021.
[16] E. Giunchiglia, M. C. Stoian, and T. Lukasiewicz. Deep learning with logical constraints. In *IJCAI 2022*, pages 5478–5485. ijcai.org, 2022.
[17] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2024. URL https://www.gurobi.com.
[18] N. Hoernle, R. Karampatsis, V. Belle, and K. Gal. Multiplexnet: Towards fully satisfied logical constraints in neural networks. In *AAAI 2022*, pages 5700–5709. AAAI Press, 2022.
[19] M. A. Jette and T. Wickberg. Architecture of the slurm workload manager. In *Job Scheduling Strategies for Parallel Processing*, pages 3–23, Cham, 2023. Springer Nature Switzerland.
[20] G. Katz, C. W. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer. Reluplex: An efficient SMT solver for verifying deep neural networks. In *CAV 2017*, volume 10426 of *LNCS*, pages 97–117. Springer, 2017.
[21] G. Katz, D. A. Huang, D. Ibeling, K. Julian, C. Lazarus, R. Lim, P. Shah, S. Thakoor, H. Wu, A. Zeljic, D. L. Dill, M. J. Kochenderfer, and C. W. Barrett. The marabou framework for verification and analysis of deep neural networks. In *CAV 2019*, volume 11561 of *LNCS*, pages 443–452. Springer, 2019.
[22] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. In Y. Bengio and Y. LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL http://arxiv.org/abs/1412.6980.
[23] T. Li and V. Srikumar. Augmenting neural networks with first-order logic. In *ACL 2019*, pages 292–302. Association for Computational Linguistics, 2019.
[24] T. Li, V. Gupta, M. Mehta, and V. Srikumar. A logic-driven framework for consistency of neural models. In *EMNLP-IJCNLP 2019*, pages 3922–3933. Association for Computational Linguistics, 2019.
[25] X. Liu, X. Han, N. Zhang, and Q. Liu. Certified monotonic neural networks. In *NeurIPS 2020*, 2020.
[26] P. Minervini and S. Riedel. Adversarially regularising neural NLI models to integrate logical background knowledge. In *CoNLL 2018*, pages 65–74. Association for Computational Linguistics, 2018.
[27] S. Park, C. Yun, J. Lee, and J. Shin. Minimum width for universal approximation. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=O-XJwyoIF-k.
[28] A. Peruffo, D. Ahmed, and A. Abate. Automated and formal synthesis of neural barrier certificates for dynamical models. In *TACAS (1)*, volume 12651 of *Lecture Notes in Computer Science*, pages 370–388. Springer, 2021.
[29] A. Sivaraman, G. Farnadi, T. D. Millstein, and G. V. den Broeck. Counterexample-guided learning of monotonic neural networks. In *NeurIPS 2020*, 2020.
[30] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat. Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS*, pages 404–415. ACM, 2006.
[31] A. Tacchella, L. Pulina, D. Guidotti, and S. Demarchi. VNN-LIB, https://www.vnnlib.org, 2022.
[32] J. Tjomsland, S. Kalkan, and H. Gunes. Mind your manners! A dataset and a continual learning approach for assessing social appropriateness of robot actions. *Frontiers Robotics AI*, 9:669420, 2022.
[33] W. Wang and S. J. Pan. Integrating deep learning with logic fusion for information extraction. In *AAAI 2020*, pages 9225–9232. AAAI Press, 2020.
[34] G. Wu, B. Say, and S. Sanner. Scalable planning with deep neural network learned transition models. *J. Artif. Intell. Res.*, 68:571–606, 2020.
[35] Y. Xie, Z. Xu, K. S. Meel, M. S. Kankanhalli, and H. Soh. Embedding symbolic knowledge into deep networks. In *NeurIPS 2019*, pages 4235–4245, 2019.
[36] J. Xu, Z. Zhang, T. Friedman, Y. Liang, and G. V. den Broeck. A semantic loss function for deep learning with symbolic knowledge. In *ICML 2018*, volume 80, pages 5498–5507. PMLR, 2018.
[37] M. Zavatteri, D. Bresolin, and N. Navarin. Code for "automated synthesis of certified neural networks", 2024. URL https://github.com/matteozavatteri/certified-nn-ecai24.
[38] H. Zhu, Z. Xiong, S. Magill, and S. Jagannathan. An inductive synthesis framework for verifiable reinforcement learning. In *ACM 2019*, pages 686–701. ACM, 2019.