Neural Termination Analysis

Mirco Giacobbe*
University of Birmingham
UK

Daniel Kroening†*
Amazon, Inc.
USA

Julian Parsert*
University of Oxford
UK

ABSTRACT

We introduce a novel approach to the automated termination analysis of computer programs: we use neural networks to represent ranking functions. Ranking functions map program states to values that are bounded from below and decrease as a program runs; the existence of a ranking function proves that the program terminates. We train a neural network from sampled execution traces of a program so that the network's output decreases along the traces; then, we use symbolic reasoning to formally verify that it generalises to all possible executions. Upon the affirmative answer we obtain a formal certificate of termination for the program, which we call a neural ranking function. We demonstrate that, thanks to the ability of neural networks to represent nonlinear functions, our method succeeds over programs that are beyond the reach of state-of-the-art tools. This includes programs that use disjunctions in their loop conditions and programs that include nonlinear expressions.

CCS CONCEPTS

• **Software and its engineering** → **Software verification and validation**; **Formal software verification**; • **Theory of computation** → **Program analysis**; **Program verification**; • **Computing methodologies** → **Machine learning**; **Neural networks**.

KEYWORDS

Artificial Intelligence and Machine Learning for Software Engineering, Automated Reasoning, Computer-aided Verification, Formal Methods, Ranking Function Synthesis, Termination Analysis

ACM Reference Format:
Mirco Giacobbe, Daniel Kroening, and Julian Parsert. 2022. Neural Termination Analysis. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22), November 14–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3540250.3549120

1 INTRODUCTION

Software is a complex artefact. Programming is prone to error and some bugs are hard to find even after extensive testing. Bugs may cause crashes, undesirable outputs, and can prevent a program from responding at all which causes poor performance and can be a vulnerability [1]. Termination analysis addresses the question of whether, for every possible input, a program halts. This is undecidable in general, yet tools that work in practice have been developed by industry and academia [21, 37, 51, 55, 69]. In this paper, we introduce a novel technique that effectively trains neural networks to act as formal proofs of termination and, thanks to the expressivity of neural networks, significantly extends the set of programs that can be proven to terminate automatically.

```
int x, y, z;
...
while (x < y || x < z) {
  x++;
}
```

Figure 1: A simple program with disjunctive loop guard.

To argue that a program terminates one usually presents a *ranking function* for each loop in the program. Ranking functions map program states to values that (i) decrease by a discrete amount after every loop iteration and (ii) are bounded from below [48]. They are certificates of termination: if a ranking function exists, then the program terminates for every possible input.

Many existing methods find ranking functions by relying on symbolic reasoning [5, 9, 18, 19, 22, 31, 32, 35, 36, 38, 40, 51, 56, 71, 79, 85, 94, 101]. Moreover, they typically focus on the case of linear ranking functions for programs that can be represented as conjunctions of linear constraints. For this particular case, Farkas' lemma offers a means to compute the ranking function efficiently [85]. However, finding proofs for programs that use disjunctive (e.g., Fig. 1) or nonlinear loop guards is much more difficult.

Our approach is based on the principle that *finding a proof is much harder than checking that a given candidate proof is valid.* We use machine learning to guess a proof followed by symbolic reasoning to verify it: first, we learn a candidate ranking function by training a neural network that decreases along sampled runs of the program; then, we use satisfiability modulo theories (SMT) solving to check whether this candidate *neural ranking function* (NRF) decreases along every possible run of the program. We use networks that are always bounded from below, thus upon success we have a proof of termination.

Our experiments with established termination benchmarks demonstrate that the idea is effective: most loops can be proven to terminate using very tiny neural networks (fewer than 10 neurons) and at most a thousand sample runs. We give exemplars of neural architectures and loss functions for training monolithic and lexicographic ranking functions. Using a neural network with just one hidden layer and a straight-forward training routine, our method discovers neural ranking functions for over 75% of the benchmarks in a standard problem set for termination analysis [15, 52]. Our

*The authors are listed in alphabetical order regardless of individual contributions or seniority.
†This work was done prior to joining Amazon.

ESEC/FSE '22, November 14–18, 2022, Singapore, Singapore

ACM ISBN 978-1-4503-9413-0/22/11…$15.00
https://doi.org/10.1145/3540250.3549120

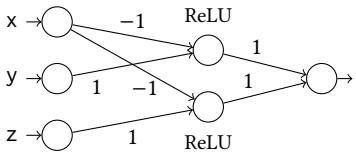

Figure 2: Neural ranking function for the program in Fig. 1.

method subsumes a broad range of existing termination analysis strategies: not only do we discover linear ranking functions, but also ranking functions for problems that require piecewise linear or lexicographic termination arguments [56, 103].

Furthermore, we observe that the ability of neural networks to represent nonlinear functions enables termination proofs for programs that go beyond what the state of the art can handle. Programs that use disjunctive or nonlinear loop guards as well as programs that require piecewise linear ranking functions are proven terminating just as easily by our new technique.

While we perform our experiments with Java programs, our training procedure is agnostic with respect to the programming language and requires no information about the program other than execution traces. It applies without modifications to software that constructs data structures as long as a procedure for verifying the candidate ranking functions is available. The complexity of formal reasoning about the program is entirely delegated to the verification procedure, which only has to solve the task of checking the validity of a given ranking function.

2 ILLUSTRATIVE EXAMPLE

We prove that programs terminate by showing that their loops admit ranking functions. Constructing ranking functions for loops that involve disjunctive loop guards or involve nonlinear constraints is hard for existing technologies. As an exemplar, consider the loop in Fig. 1. This loop terminates for every arbitrary initialisation of the variables x, y, and z and only involves linear constraints. Yet, a linear ranking argument is insufficient to prove that this loop terminates. Naively one might believe that $-x$ is a ranking argument because it decreases with every loop iteration; however, it is not bounded from below by a given constant coefficient. Notably, the same holds for expression $y + z - x$, which can be always assigned to a value that is smaller than any given constant coefficient with an adversary initialisation of y and z, if we assume that x, y, and z can take any unbounded integer. In fact, under this assumption, no linear combination of x, y, and z is a valid ranking function for this loop, which requires a nonlinear ranking function.

A valid ranking function for the program in Fig. 1 is

$$f(x, y, z) = \max\{y - x, 0\} + \max\{z - x, 0\}. \tag{1}$$

This function not only decreases in every iteration, but is also non-negative for every valuation of x, y, and z and it is bounded from below by zero. This function corresponds to the simple neural network with ReLU activation functions in Fig. 2, which is effectively learned and verified by our method.

We argue that (1) neural networks are a powerful model to represent ranking functions of non-trivial programs. Loops that include disjunctions in their loop guards are not rare. Similar behaviour can

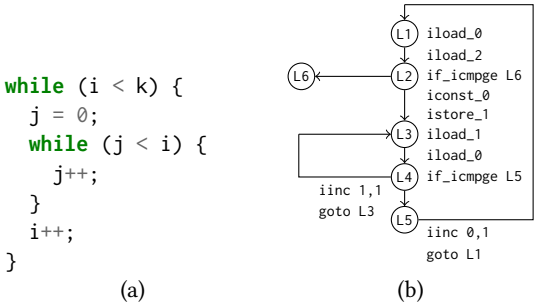

Figure 3: A Java program and the respective CFG.

be induced by conditional control flow, early breaks or by throwing exceptions. Suffice it to say that the following loop is semantically equivalent to the example in Fig. 1:

```
while (true) {
  if (x >= y && x >= z)
    break;
  x++;
}
```

Moreover, we also argue that (2) decoupling the process of guessing the ranking function from that of checking it enables us to effectively discover termination arguments for programs that involve nonlinear constraints. As it turns out, as of today, neither AProVE nor Ultimate can determine (within a time budget of 60 s) that the following loop terminates:

```
while (x*x*x < y) {
  x++;
}
```

By contrast, our method learns and verifies the ranking function $\max\{y - x, 0\}$ in less than a second.

3 BACKGROUND

Programs and Transition Systems. A computer program is a list of instructions that, together with the machine interpreting them, defines a state transition system over the state space of the machine. A state transition system is a pair $P = (S, T)$ where S is a countable (possibly infinite) set of states and $T \subset S \times S$ is a transition relation. A state contains all information that is necessary to determine its successor state(s). For example, this can include the value of variables that are explicitly declared in the source code, or that exist as registers in the interpreter (e.g., the program counter). The transition relation determines the successors of a state. A state without any successors is a terminating state. A state may have multiple successors because of operations that are external to the program (e.g., non-deterministic input assignments) or are underspecified and may therefore have multiple outcomes. A run of P is any sequence of states $s^{(0)}, s^{(1)}, s^{(2)}, \ldots$ such that $(s^{(i)}, s^{(i+1)}) \in T$ for all $i \geq 0$. We say that a program terminates if all its runs are finite.

Control Flow Graphs and Loop Headers. A control flow graph (CFG) for program P is a finite directed graph $G = (L, E)$ where L is a finite set of control locations and $E \subseteq L \times L$ is a set of control edges. For Java programs, a CFG can be obtained for its bytecode,

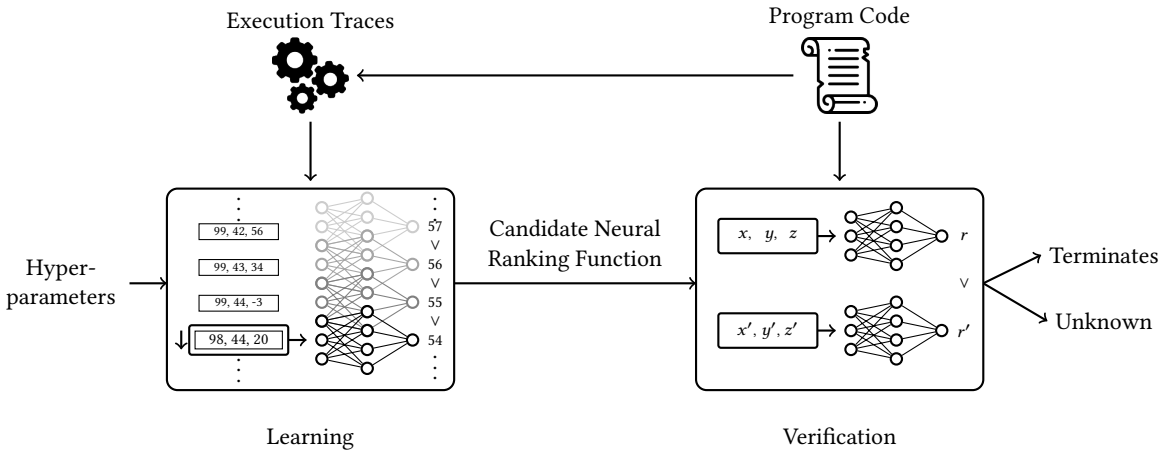

Figure 4: Schema of our framework.

as illustrated in Fig. 3. Control locations correspond to source and target addresses of jump instructions or entry or exit points of a procedure. Control edges indicate whether there exists a sequence of instructions or a jump that lead from the respective source to the respective destination location. A state is on a control location if the next instruction to be executed is on a control location; in the Java Virtual Machine, this is determined by the program counter. Notably, we have that for every finite run $s^{(0)}, \ldots, s^{(k)}$ such that $s^{(0)}$ and $s^{(k)}$ are respectively on control locations l and l', the control graph must admit a path from l to l'. Then, let $L' \subseteq L$ be any subset of control locations. We define $S_{L'} \subseteq S$ as the set of states on any location in L' and $T_{L'} \subseteq S_{L'} \times S_{L'}$ as the the maximal relation of states on control locations in L' such that $(s^{(0)}, s^{(k)}) \in T_{L'}$ only if there exist a finite run $s^{(0)}, s^{(1)}, \ldots, s^{(k-1)}, s^{(k)}$ that does not encounter any control location in L' in between, i.e., $s^{(1)}, \ldots, s^{(k-1)} \notin S_{L'}$. Special control locations are loop headers, that is, the dominators (entry locations) of the strongly connected components in the graph [6]; for instance, the loop headers for Fig. 3b are L1 and L3. They are important in termination analysis because every run along a loop (i.e., for and while statements) necessarily enters and iterates over at least one of them. We denote the set of loop headers as $H \subseteq L$.

Ranking Functions. To determine whether a program terminates, our method attempts to find a ranking function for it. A function $f \colon S_H \to R$ is a ranking function for the program P if

$$f(s') \prec f(s) \quad \text{for all } (s, s') \in T_H \tag{2}$$

and the relation $(R, \prec)$ is well founded [48]. The existence of a ranking function proves that the program terminates, and every program that does not terminate necessarily lacks a ranking function. A standard way of giving a ranking function is to identify a function that maps states at loop headers to sequences of numbers that (i) decrease by a discrete amount and (ii) are bounded from below. Another way is to define tuples of functions that decrease lexicographically at loop headers, which is particularly useful for nested loops.

4 OVERVIEW OF THE METHOD

We propose a framework (Fig. 4) for termination analysis using neural networks as ranking functions. These neural ranking functions are first trained over execution traces of a program and subsequently verified in combination with the program code. Thus, the three parameters of *neural termination analysis* are

(1) the tracing/sampling strategy,
(2) the neural network architecture, and
(3) the verification procedure.

These steps are independent from each other. Hence, changing the learning setup (e.g. considering different models, other learning frameworks, etc.) only requires a change in the second step. Similarly, considering different input languages or alternative verification procedures only requires a change in the verification and (potentially) tracing procedure.

As illustrated in Fig. 4, in the first step we collect execution traces for a program P that we want to show terminating. These execution traces are used subsequently as training data to train the neural ranking functions. These traces are obtained by first generating test input data for P and then tracing the execution of P with the test input data. Since the training phase exclusively works with these execution traces it is important that they adequately represent the behaviour of the program P. More details can be found in Appendix A.

In the second step, we use the execution traces to train neural networks to become neural ranking functions. We discuss suitable choices for neural network architectures in Sect. 5. We always use neural architectures that guarantee that the neural network's output is bounded from below. The training procedure minimises a loss function that punishes neural networks that do not decrease over the sampled observations. As a result, we obtain a neural network that behaves like a ranking function over the sampled traces.

Finally, to assert that a trained neural ranking function generalises to all possible executions, we pass it to a formal verification procedure. The formal verification procedure encodes both the program and the neural network symbolically (more details about our encoding are shown in Appendix B). Then, it uses SMT solving

to formally decide whether the neural network is a valid ranking function for the program. Upon an affirmative result, we conclude that the program terminates; upon a negative result, we return an inconclusive answer, that is, the program may or may not terminate. The verification procedure guarantees that our method is sound.

The combination of a particular sampling strategy, neural architecture, and verification procedure is an instance of our approach, which offers a flexible and extensible termination analysis framework that combines testing methods, deep learning, and symbolic reasoning.

5 NEURAL RANKING FUNCTIONS

Our method collects execution traces for a program and then proceeds in two phases: first it trains and then it formally verifies. The first phase thus takes a set of execution traces as input and returns a neural network as output. The set of execution traces forms the dataset which is used to train a neural network that mimics a ranking function along these traces. The neural network is trained by minimising a loss function that ensures that the neural network decreases by a discrete amount after every pair of subsequent observations in the traces. We analyse every loop in the program and provide a ranking function for each of them.

We provide two strategies for training these candidate neural ranking function. The first strategy trains a monolithic ranking argument, that is, a neural ranking function that outputs one value that decreases along the traces. The second strategy generalises the first and trains a lexicographic ranking argument, that is, a neural ranking function that outputs many values that decrease lexicographically. Whether to use one or the other strategy is heuristic; lexicographic arguments are normally suitable for nested loops.

Observation Functions and Traces. We train our candidate neural ranking function from states collected along program runs. However, this is made difficult by the fact that the states of a program are large and complex and contain internal information that is not directly amenable to deep learning. The standard approach to address this issue is to construct an *embedding*, defined by means of an observation function. An observation function $\omega\colon S_{L'} \to \mathbb{R}^n$ extracts vectors of numerical values that can be taken as input by a neural network, from states that are on a specific set of control locations $L' \subseteq L$. These numerical values can be, for instance, the values of numerical variables in memory. With an observation function ω we convert any run into a trace $o^{(0)}, o^{(1)}, o^{(2)}, \ldots$, which is the sequence of observations in $\mathbb{R}^n$ recorded every time a location in L' is encountered. In other words, every such trace corresponds to a sequence $s^{(0)}, s^{(1)}, s^{(2)}, \ldots$ of states in $S_{L'}$ such that $o^{(i)} = \omega(s^{(i)})$ and $(s^{(i)}, s^{(i+1)}) \in T_{L'}$, for all $i \geq 0$.

General Architecture for Neural Ranking Functions. We use feed-forward neural networks as models of ranking functions. Generally, this is a function $f\colon \Theta \times \mathbb{R}^n \to \mathbb{R}^m$ with n inputs, m outputs, and parameterised by $\theta \in \Theta$, defined in terms of interconnected neurons partitioned into one input, one output, and k intermediate hidden layers. The intermediate hidden layers are defined as a parameterised function

$$\sigma(W, b; x) = \mathrm{ReLU}(Wx + b) \tag{3}$$

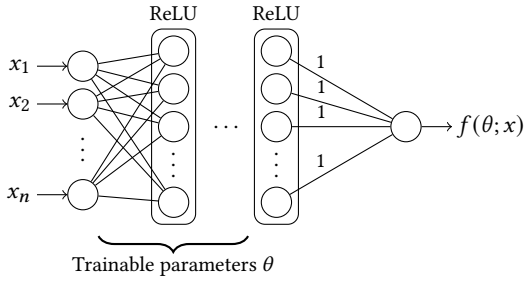

Figure 5: Architecture for monolithic NRFs.

where the parameters are a matrix of weights W and a vector of biases b. These define an affine transformation of the input, while ReLU applies a nonlinear transformation $\max\{\cdot, 0\}$ element-wise to each of the h neurons in the respective hidden layer, that is

$$\mathrm{ReLU}(x_1, \ldots, x_h) = (\max\{x_1, 0\}, \ldots, \max\{x_h, 0\}). \tag{4}$$

The parameters of the network are thus a sequence of weight matrices and bias vectors $\theta = (W^{(1)}, b^{(1)}, \ldots, W^{(k)}, b^{(k)})$, each of which corresponds to a hidden layer. We impose that the output layer has no bias, its weights $W^{(k+1)}$ are not trainable—and their coefficients are non-negative. The neural network thus defines the function

$$f(\theta; \cdot) = W^{(k+1)}\sigma(W^{(k)}, b^{(k)}; \cdot) \circ \cdots \circ \sigma(W^{(1)}, b^{(1)}; \cdot), \tag{5}$$

whose output is in turn guaranteed to be non-negative for every valuation of inputs and parameters. This ensures that the function output is always bounded from below by zero. To train this neural network so as it behaves as a raking function it remains to ensure that it decreases after every loop iteration.

Monolithic Ranking Loss. A monolithic neural ranking function is the special case where $m = 1$; one example is depicted in Fig. 5. Our goal is to train it in such a way its output decreases by a discrete amount $\delta > 0$ along a set of sample traces of observations collected every time a loop header is encountered. For this purpose, we define an embedding using an observation function $\omega\colon S_H \to \mathbb{R}^n$. With this embedding, we record multiple traces from the program under analysis and store a dataset of observation pairs $D \subset \mathbb{R}^n \times \mathbb{R}^n$ such that for every pair $(\boldsymbol{o}, \boldsymbol{o'})$ we have that $\boldsymbol{o}$ is immediately followed by $\boldsymbol{o'}$ in the trace. In other words, D constitutes a sliding window of size two over the trace. Note that D possibly contains the pairs of multiple execution traces. We train our network so as to decrease between every pair $(\boldsymbol{o}, \boldsymbol{o'}) \in D$ of sampled observations:

$$f(\theta; \boldsymbol{o'}) \leq f(\theta; \boldsymbol{o}) - \delta. \tag{6}$$

To this end, we solve the following optimisation problem:

$$\arg\min_{\theta} \frac{1}{|D|} \sum_{(\boldsymbol{o}, \boldsymbol{o'}) \in D} \underbrace{\max\{f(\theta; \boldsymbol{o'}) - f(\theta; \boldsymbol{o}) + \delta, 0\}}_{\mathcal{L}(\boldsymbol{o}, \boldsymbol{o'}, \theta)} \tag{7}$$

Function $\mathcal{L}(\boldsymbol{o}, \boldsymbol{o'}, \theta)$ is the loss of the neural network over a given pair. The higher the value of $\mathcal{L}$ the more the network increases over the pair, whereas for pairs that decrease by at least δ the value is always 0 (i.e., negative values do not affect the sum). As a result of the optimisation problem, we obtain parameters that ensure the network decreases along all sampled traces.

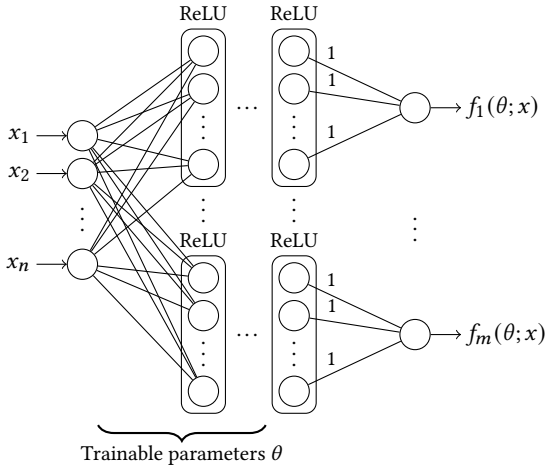

Figure 6: Architecture for lexicographic NRFs.

Example 5.1. Consider the program in Fig. 1. For this program, we define an embedding $\omega\colon S_H \to \mathbb{Z}^3$ that observes the values of x, y, and z every time a run hits the entry location of the loop. We reason about the loop in isolation and sample random initial values for the variables. Two example traces are

$$(5,-3,10),(6,-3,10),(7,-3,10),(8,-3,10),(9,-3,10),(10,-3,10)$$

and

$$(-2,3,-5),(-1,3,-5),(0,3,-5),(1,3,-5),(2,3,-5),(3,3,-5).$$

The dataset corresponding to exactly these traces thus contains 10 pairs of consecutive observations. We use the monolithic architecture in Fig. 2, which has exactly one hidden layer made of two neurons. We set $\delta = 1$ and observe that once the parameters in Fig. 2 are attained, the loss function $\mathcal{L}$ measures zero over all pairs, which is its minimum. The corresponding function f in Eq. 1 maps both traces to the sequence

$$5, 4, 3, 2, 1, 0.$$

The SMT solver successfully verifies that the following formula evaluates to true for every possible assignment to $\mathsf{x}, \mathsf{y}, \mathsf{z}, \mathsf{x}', \mathsf{y}', \mathsf{z}'$:

$$[(\mathsf{x} < \mathsf{y} \vee \mathsf{x} < \mathsf{z}) \wedge \mathsf{x}' = \mathsf{x} + 1 \wedge \mathsf{y}' = \mathsf{y} \wedge \mathsf{z}' = \mathsf{z}] \implies f(\mathsf{x},\mathsf{y},\mathsf{z}) - \delta \geq f(\mathsf{x}',\mathsf{y}',\mathsf{z}') \quad (8)$$

Unprimed variables represent an observation before an iteration and primed variables after an iteration. The validity of this formula confirms that f decreases by δ for every possible assignment of the variables and is thus a valid ranking function. Note that f is bounded from below by construction owing to the use of ReLUs.

Lexicographic Ranking Loss. Neural ranking functions with $m \geq 2$ can learn lexicographic ranking arguments. Figure 6 gives an example of an architecture for this purpose. Our goal is training the neural network to ensure that every pair of observations decreases some output neuron by δ, as long as all other outputs with smaller index do not increase. Lexicographic arguments are suitable to programs with multiple loop headers. In this case, we associate to each loop header an index $i \in \{1,\dots,m\}$ for the output neuron that we expect to decrease every time the header is visited. For every header we define an embedding that records an observation every time that header is visited. We thus obtain multiple datasets $D_1,\dots,D_m$ from our sample runs, one for each header. We train our network to obtain that, for each pair $(\boldsymbol{o},\boldsymbol{o}') \in D_i$ at the i-th header, the output neurons decrease lexicographically as follows:

$$f_i(\theta;\boldsymbol{o}') \leq f_i(\theta;\boldsymbol{o}) - \delta \qquad \text{and} \quad (9)$$

$$f_j(\theta;\boldsymbol{o}') \leq f_j(\theta;\boldsymbol{o}) \qquad \text{for all } j < i. \quad (10)$$

To train the network we solve the following problem:

$$\arg\min_{\theta} \frac{1}{|D_1| + \cdots + |D_m|} \sum_{i=1}^{m} \sum_{(\boldsymbol{o},\boldsymbol{o}') \in D_i} \mathcal{L}_i(\boldsymbol{o},\boldsymbol{o}',\theta) \quad (11)$$

The loss of a pair of observations is determined by the dataset it belongs to:

$$\mathcal{L}_i(\boldsymbol{o},\boldsymbol{o}',\theta) = \max\{f_i(\theta;\boldsymbol{o}') - f_i(\theta;\boldsymbol{o}) + \delta, 0\} + \sum_{j=1}^{i-1} \max\{f_j(\theta;\boldsymbol{o}') - f_j(\theta;\boldsymbol{o}), 0\}. \quad (12)$$

Function $\mathcal{L}_i(\boldsymbol{o},\boldsymbol{o}',\theta)$ takes its minimal value 0 when both conditions (13) and (10) are satisfied. If (11) also attains value 0, then all samples satisfy these conditions and the network constitutes a lexicographic neural raking function over the sampled traces.

Example 5.2. Consider the program in Fig. 3, which has two nested loops. The outer loop has its header at location L1 and the inner at location L3. We associate L1 with index 1 and L3 with index 2 and learn a lexicographic argument accordingly. We define two embeddings $\omega_1\colon S_{\{\mathsf{L1}\}} \to \mathbb{Z}^3$ and $\omega_2\colon S_{\{\mathsf{L3}\}} \to \mathbb{Z}^3$ that map states at control locations L1, resp. L3, to the values of i, j, and k. We take, for this example, one sample run with initial values 0,0,3 and obtain the following two datasets of pairs induced by ω_1 and ω_2 respectively:

$$D_1 = \{((0,0,3),(1,0,3)),((1,0,3),(2,1,3)),((2,1,3),(3,2,3))\}$$

$$D_2 = \{((1,0,3),(1,1,3)),((2,0,3),(2,1,3)),((2,1,3),(2,2,3))\}$$

We use a neural network as in Fig. 6 with one hidden layer and one hidden neuron in each of the two blocks. The first output must converge to the following function:

$$f_1(\mathsf{i},\mathsf{j},\mathsf{k}) = \max\{\mathsf{k} - \mathsf{i}, 0\}. \quad (13)$$

The second output may converge to either of the following functions, which are both valid:

$$f_2(\mathsf{i},\mathsf{j},\mathsf{k}) = \max\{\mathsf{i} - \mathsf{j}, 0\} \quad (14)$$

$$f_2(\mathsf{i},\mathsf{j},\mathsf{k}) = \max\{\mathsf{k} - \mathsf{j}, 0\} \quad (15)$$

Proving that these functions are a valid termination lexicographic argument relies on auxiliary invariants which, as discussed in the appendix, we extract using a syntactic heuristic. For instance, checking that (15) decreases along the inner loop requires the auxiliary invariant $i < k$; checking that (13) decreases along the outer loop requires an argument that the inner loop leaves k and i unchanged.

Table 1: Results of running *v*Term (with 7 neurons), AProVE, Ultimate, and DynamiTe on term-crafted, Aprove_09, and nuTerm-advantage. In the case of *v*Term we report the average results rounded to the first decimal. The last two columns show the union of the first two problem sets and all three problem sets, respectively.

	Aprove_09(1)		term-crafted(2)		nuTerm_advantage(3)		combined{1, 2}		combined{1, 2, 3}	
#	38		72		14		110		124	
*v*Term	34.6	91%	49.6	68.9%	13	92.9%	84.2	76.6%	97.2	78.4%
Aprove	34	89.5%	64	88.9%	3	21.4%	98	89.1%	102	81.5%
Ultimate	34	89.5%	61	84.5%	1	7.8%	95	86.4%	96	77.4%
DynamiTe	31	81.6%	46	63.9%	7	50%	77	70%	84	67.7%

6 EXPERIMENTS

We present an experimental evaluation to answer the following research questions:

RQ1 Can neural ranking functions be used to formally prove the termination of programs?

RQ2 Do neural ranking functions advance the state of the art in termination analysis?

RQ3 How do neural ranking functions scale in terms of the complexity of the program?

To answer these questions, we developed a prototype implementation of the methods discussed in the previous sections for proving termination of Java programs, which we name *v*Term. Our implementation strictly separates tracing, learning, and verification as discussed in Sect. 4.

To obtain program traces we first generate test data for the program in question using a multivariate normal distribution. Subsequently, we execute the program using this test data while maintaining control of the execution and collecting memory snapshots using the Java Virtual Machine Tool Interface (JVMTI). For more details on sampling and tracing we refer to Appendix A.

Once tracing is completed, this data is used to train the neural ranking function where PyTorch [83] is used as the machine learning framework. Finally, we encode the problem of certifying the neural ranking function into an SMT formula and use Z3 [44] to solve it. More details about the implementation of this step can be found in Appendix B.

6.1 Benchmarks and Setup

We consider three sets of programs for our experimental evaluation. The first and second set are comprised of problems from the TermComp Termination Competition [52] and the the SV-COMP Software Verification Competition [15]. Both problem sets are publicly available and cover a wide variety of termination and non-termination problems as well as software verification in general.

From these sets, we discard the non-terminating programs as they do not have ranking functions. Furthermore, for this evaluation we consider deterministic programs with a maximum of two nested loops without function calls. Hence, we focus on the two problem sets Aprove_09 from TermComp and term-crafted from SV-COMP. After dropping problems due to the aforementioned constraints we are left with 72 problems from term-crafted (originally 159) and 38 problems (originally 76) from Aprove_09. The problem set term-crafted comprises problems from literature on termination analysis, which are given as C code. We therefore translate these problems into Java by hand. In addition, we also split the main functions' bodies into an initialisation and loop part. Note that this purely syntactic change does not alter the difficulty of determining termination of a program. Finally, we present an additional problem set – nuTerm-advantage – consisting of problems created by us for showcasing notable strengths and weaknesses of the tested tools.

To answer *RQ2*, we compare *v*Term to Ultimate [56], Aprove [51], and DynamiTe [69], which, collectively, represent the state of the art in termination analysis.

Setup. The experiments were conducted on Linux Kernel 5.15 running on an Intel Core i7 5820K at 3.3 GHz with 16 GB RAM and an NVIDIA GTX 980 graphics card. For learning we use the Adam optimiser provided by PyTorch and a learning rate of 0.05. We run the benchmarks 5 times with random seeds that were fixed a priori for reproducibility. Full instructions on how to reproduce the results (including the seeds) are part of the supplementary material. We ran all tools with a timeout of 60 seconds for each problem.

6.2 Experimental Results

Can neural ranking functions be used to formally prove the termination of software programs? To answer this question we ran *v*Term on the three benchmark sets mentioned above. The results are given in Tab. 1. We observe the best performance of *v*Term when using a neural network consisting of 7 neurons with 1000 sample traces with a maximum length of 1000. The exact strategy used is described in the supplementary material. *v*Term proves termination for 97.2 out of 124 problems on average (100 in the best out of the five runs), which accounts for 78.4% of the problems in the problem set. When considering the different problem sets separately we solve 91.0% (Aprove_09), 68.9% (term-crafted), and 92.9% (nuTerm_advantage) of the problems. Note that even when disregarding the nuTerm_advantage set, *v*Term solves 76.6% of the problems. Given that very simple neural networks suffice to prove termination of a substantial subset of the standard benchmarks, we answer *RQ1* in the affirmative.

Do neural ranking functions advance the state of the art in termination analysis? To compare with the state of the art, we also ran Ultimate [56], AProVE [51], and DynamiTe [69] on the same

benchmarks. Since Ultimate [56] and DynamiTe [69] do not support Java code as input we ran the experiments on the C versions of the problems. The results are presented in Tab. 1. Overall, the strongest tool is AProVE, which solves 81.4% of all problems, followed by *v*Term with 78.4% and Ultimate with 77.4% and finally DynamiTe with 67.7%. By considering the preexisting data sets separately, we see that *v*Term comes in first on the Aprove_09 set and third on term-crafted. On these two sets combined, *v*Term solves 76.6% of the problems with AProVE and Ultimate solving 89.1% and 86.4% respectively and DynamiTe trailing with 70% of problems solved. We conclude that on the existing benchmarks, *v*Term performs comparably to the state of the art.

Our hypothesis is that *v*Term advances the state of the art when applied to programs that have either disjunctive loop conditions or programs that are nonlinear. However, the existing benchmark sets suffer from confirmation bias, and focus on programs that avoid these features. To show that *v*Term indeed advances the state of the art, we have compiled the nuTerm-advantage data set, with programs that feature

(1) non linear conditions, and
(2) disjunctions in conditions.

The following code snippet is from DynamiteExampleX4.java from the nuTerm-advantage problem set:

```
int a = 0;
while ( a*a*4 <= n ) {
  a = a + 1;
}
```

This loop only uses two variables, a and n, where n remains constant throughout the execution while a is incremented by 1 in every iteration. Despite the fact that the loop guard $a^2 \cdot 4 \leq n$ is nonlinear, there is a linear ranking function. Furthermore, the execution traces of the loop only show the incrementing of a, which is also linear. Our tool *v*Term can solve this problem with a tiny neural network, consisting of a single neuron, and reports the ranking function $\text{ReLU}(n - a + 1)$. Neither Aprove_09 nor Ultimate are able to prove termination of this problem, but DynamiTe, a tool which also utilises execution traces, can solve it.

Disjunctions increase the complexity of formal reasoning significantly. This is illustrated by the following code snippet from Square2VarsDisj.java in the nuTerm-advantage set:

```
int a = 0, b = 0;
while ( a*a <= m || b*b <= n ) {
  a = a + 1;
  b = b + 1;
}
```

None of the tools we compare with can show termination of this loop, while *v*Term proves termination by learning the ranking function $\text{ReLU}(m - a + 2) + \text{ReLU}(n - b + 2)$. Note that DynamiTe is able to show termination if the loop head was either `while` $a^2 \leq$ m or `while` $b^2 \leq$ n. However, once both conditions are connected with a disjunction, DynamiTe fails to show termination.

The nuTerm-advantage problem set comprises problems that exhibit either nonlinear conditions, disjunctions, or a combination of both. For this set, *v*Term solves on average 92.9% of the problems

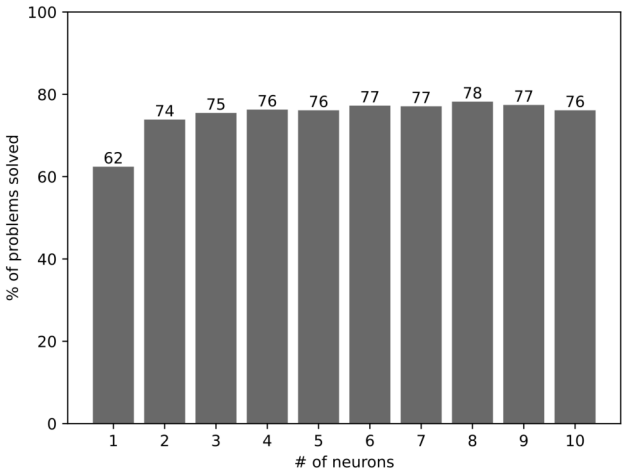

Figure 7: Percentage of problems solved for neural networks with 1 to 10 hidden neurons.

while DynamiTe comes second, solving 50%, followed by AProVE (21.4%) and Ultimate (7.8%).

In conclusion, the experiments conducted and presented in Tab. 1 show that on existing benchmarks, *v*Term either performs either comparable to or stronger than (e.g., on Aprove_09) the state of the art. Furthermore, we identified weaknesses in the existing tools when considering a broader range of programs and show that neural termination analysis can solve these problems by providing a set of programs on which *v*Term outperforms all existing tools by a large margin. Thus, we can answer *RQ2* in the affirmative.

How do neural ranking functions scale in terms of the complexity of the program? Our experiments have only required tiny neural networks, consisting of no more than 10 neurons. The results presented in Tab. 1 were obtained by a neural network consisting of 7 neurons. Increasing the number of neurons further does not yield significant gains on the existing problem sets as shown in Fig. 7. We hypothesise that programmers avoid writing loops that require termination arguments that depend on a very large number of variables. To evaluate how our technique scales in the number of variables that are required for the ranking argument, we use the following program template, designed to require at least k neurons.

```
int a = 0;
while ( a*a*4 <= n_1 || ... || a*a*4 <= n_k ) {
  a++;
}
```

We include the instances of this template for values of k up to 4 in the nuTerm-advantage problem set. Note that neither AProVE nor Ultimate is able to solve this problem for any value of k; DynamiTe is able to solve these problems for k up to 2 (with a noticeable increase in runtime from 8 s to 30 s), but times out for any larger k. Note that *v*Term solves these problems in the problem set within 2 seconds. Trying programs with k up to 10 it becomes clear that the learning procedure continues to scale well while the verification starts to become the bottleneck.

It is worth emphasising that neural networks with 7 neurons, for example, which would be able to prove such a loop for $k = 7$ terminating, are laughably small compared to state-of-the-art neural networks used in other areas such as natural language processing. Hence, it is likely that by increasing the size of the neural network one would sooner run into issues verifying the neural networks and generating meaningful traces than training the neural networks. Furthermore, we hypothesise that the vast majority of loops in real world programs do not have termination conditions that involve hundreds of disjuncts. In conclusion, we can tentatively answer *RQ3* by suggesting that neural termination analysis scales well in the complexity of the program, noting that other parts such as verification and tracing might not scale as well.

6.3 Discussion

Owing to the inherent difficulty of the problem at hand (termination), methods for solving it are necessarily incomplete. Neural termination analysis is no exception. Under this constraint we presented an experimental evaluation to answer three research questions regarding the efficacy of neural ranking functions (*RQ1*), their advantages over other approaches (*RQ2*), and their scalability (*RQ3*). Despite the favourable answers for each of the questions it is important to point out weaknesses of our approach. For example, it is easy construct programs that other methods can prove terminating but where neural termination analysis fails. Usually, these fall into one of the following three cases.

Insufficient Data. Neural termination analysis learns termination arguments from execution traces. Hence, any program feature that limits the data that can be collected is a problem for our approach. Several benchmarks in the dataset exhibit behaviour such as the following:

```
while (x > 0) {
  x = -2*x + 10;
}
```

This loop has more than one iteration if and only if x is one of $1, 2, 3, 4$ and even then the trace is extremely short. Similar issues can occur when offsets or certain program branches only occur in rare cases. Such instances may lead to overfitting of the networks to the sampled traces. As a result, a learned neural ranking function may satisfy all required properties over the sampled traces but not when verifying it with respect to all possible inputs in the verification procedure. Solver-driven test input generation may be a means to ensure that traces for these behaviours are included in the training data set. Another issue related to insufficient data is the existence of large constants in the problems. For instance, considering a problem such as the following:

```
while (x < 10000000) {
  x++;
}
```

This would require complete traces (letting x go all the way to 10000000), which take a long time to gather. Furthermore, learning a bias that large can also pose problems.

Model Expressivity. As ranking functions become more complex, we need neural architectures that are able to express them. One instance from the dataset where this problem manifests is a benchmark where the ranking function depends on whether an input variable is even or odd. None of the neural architectures discussed in Sec. 5 is expressive enough to capture the concept of "even" and "odd". One way of solving this problem is by considering further neural architectures, which would require a more sophisticated data collection. The key limiting factor when deploying such architectures will likely be the increased complexity of the verification process, rather than the learning.

Verification. When there are multiple correct ranking functions the verification procedure may not be able to prove all of them correct. The following loop can exhibit such behaviour:

```
int j = i;
while (i < 100) {
  i++;
  j = i;
}
```

When purely looking at the execution traces, which is what the learning procedure does, i and j have the exact same values at the loop head. Hence, if the learning process comes up with the ranking function $100 - j$ the verifier would not be able to prove it correct unless it is supplied with the auxiliary invariant that $i = j$ at the loop head. One way to solve this problem could be to integrate existing methods that discover such invariants [46, 92].

7 THREATS TO VALIDITY

We discuss threats to the validity of our experimental claims.

Benchmark Bias. Our claims depend on the choice of benchmark programs. We focus on sequential Java programs, and programs in other programming languages, or programs that use concurrency, may require ranking functions that our method cannot find. While we use standard benchmarks from literature introduced by others to enable a comparison of different termination tools, these benchmarks may not be representative of software written by developers. Moreover, a source of bias may be introduced by our collection of programs with either disjunctive conditions or nonlinear behaviour. While we believe that both features are important in commodity software, it remains to be quantified how large the benefit of supporting these features is on a larger repository of software.

Test input generation. We require that test inputs can be generated that exercise the programs to yield traces to train the neural network. While our simple sampling method is successful on our benchmarks, it may not be possible in general to obtain sufficiently diverse test inputs. More sophisticated means to generate test inputs are known, and can be used to mitigate this threat.

Neural architecture complexity. While our experiments suggest that tiny neural networks are able to prove termination of most program loops, there may exist programs that require a large number of neurons, increasing training and verification complexity.

Auxiliary Invariants. We use a standard verification step for checking the validity of the neural ranking function, which, in some cases, requires an auxiliary invariant. The results of an end-to-end termination analysis are dependent on the quality of these invariants, and the tools we compare with use a variety of different

approaches to solve this problem. Our own tool uses a simplistic heuristic for guessing these invariants (Appendix B), which may be exceptionally successful. The documentation on the algorithms for generating these invariants is limited, and a proper comparison of alternative methods for generating ranking functions requires an implementation in a single framework.

8 RELATED WORK

8.1 Termination Analysis

Many methods for automatically proving termination have been developed and implemented. Owing to the undecidability of the problem in general, most techniques restrict the scope of the analysis in some way, e.g., to linear ranking functions and programs [14, 53, 85], semi-definite programs [40], semi-algebraic systems [32], or formulae drawn from specific SMT theories [35]. To deal with complex program loops, lexicographic ranking functions [18, 19, 71], piecewise ranking functions [101, 102], disjunctively well-founded transition invariants [36, 38, 67, 86], and implicit ranking functions have been used [27]. Ranking functions have been synthesised symbolically using, e.g., Farkas' lemma and template-based guess-and-check strategies [47, 103]. To prove conditional termination, also abstract interpretation (by underapproximation) and loop summarisation methods have been used [26, 34, 41, 100, 111].

Alternative methods perform termination analysis by translating programs to alternative models of computation and show that the resulting model is terminating. This requires a guarantee that termination of the translated program implies the termination of the original program. Models used for this purpose include term rewriting systems [22, 51, 79], constraint logic programs [94], recurrence relations [5], and Büchi automata [31, 56].

More recently, SMT solving has been used to discover ranking functions from execution traces [69], similarly to methods based on machine learning.

8.2 Machine Learning for Termination Analysis

In the last years, several termination analysis approaches that incorporate machine learning technologies have been presented. Early methods learn linear ranking functions from execution traces by constructing a linear regression problem whose solutions describe a loop bound [77]. Recently, machine learning models such as Support Vector Machines (SVM) have been used as representation for ranking functions in [107]. Methods based on SVM have been applied to single or nested loop programs defined using conjunctions of continuous functions for the guard, and deterministic assignments defined as continuous functions as well [72].

Another deep learning approach for termination analysis has recently been introduced [97]. This method uses neural networks with sigmoidal activation functions, which are shown to be an appropriate ranking function representation for programs defined using continuous functions, without disjunctions and conditional choices. While this is suitable to describe deterministic dynamical systems in discrete time, this language restriction makes the method inapplicable to software, including the majority of our simple termination analysis benchmarks. We estimate that 46 out of 110 (cf. combined $\{1, 2\}$ in Tab. 1) programs in the preexisting benchmark sets are in the scope of (but not necessarily solved by) their method and remark that our method solves 39 out of these 46 problems. In the 14 benchmarks in `nuTerm_advantage` ($\{3\}$) only 5 are in their scope. Unfortunately, we cannot directly evaluate the effectiveness of their method on our benchmark set (neither $\{1, 2\}$ nor $\{3\}$), because an implementation is unavailable. Moreover, their method cannot be easily implemented in our infrastructure. In fact, neural ranking functions with sigmoidal activation lack efficient—and complete—decision procedures for checking their validity. Notably, their approach required the development of bespoke decision procedures for this purpose. Conversely, our method uses ReLU activation functions, which can be encoded into expressions in decidable theories, for which efficient SMT solvers are available. Our work goes a step further by showing that neural networks with ReLU activation functions are sufficient to obtain results that are comparable to state-of-the-art tools and even enable the effective termination analysis of programs that are beyond their reach.

Recently, a data-driven method has taken a similar approach and employed efficiently checkable templates to learn loop bounds [106]. They propose a portfolio of methods and templates for this purpose. By contrast, our method employs neural networks whose expressive power subsumes a wide variety of ranking function templates. Our learning phase only relies on optimising a loss function, and can thus be implemented using generic optimisation algorithms that are readily available in machine learning frameworks.

A heuristic approach to termination analysis based on deep learning has been proposed by Alon and David [7]. This approach uses graph neural networks on a program's abstract syntax tree to estimate the likelihood of termination. Specifically, this results in attention networks that propose locations in the program that might be a cause for non-termination. Importantly however, the proposed method trains neural networks over a large dataset of terminating and non-terminating programs in a supervised learning fashion. The approach is envisioned to be part of a debugging workflow where potential issues are highlighted for consumption by a programmer or another analyser, and it is not aimed at providing formal proofs of termination; therefore, it does not give correctness guarantees. By contrast, our approach has a distinct training phase for each program and does not rely on any a priori knowledge. Moreover, it provides a formal certificate of termination—the neural ranking function—whose validity we check using SMT solving. Our method is thus fully unsupervised and provides formal guarantees of termination when a valid neural ranking function is found.

8.3 Deep Learning for Automated Reasoning

Neural networks have been used in other areas of automated reasoning and verification. In automated and interactive theorem proving, neural networks have been used for premise selection [60] and proof search in first and higher-order logic [78, 81]. Recent approaches have made use of deep learning in program and control synthesis [43, 82, 88, 95]. In software verification, deep learning has been used to find loop invariants [20, 89, 92].

Our method falls within the realm of approaches that use neural networks *to represent*, rather than *to output*, formal certificates of correctness with soundness guarantees. Exemplars are Lyapunov neural networks and their generalisation into neural barrier certificates, which have been used for the formal stability and safety

analysis of dynamical systems [2, 3, 25, 28, 42, 84, 110]. More recently, the neural ranking supermartingale model has been introduced for the termination analysis of probabilistic programs, and subsequently applied to the stability analysis of stochastic control systems [4, 70]. In a similar fashion, also decision tree learning has been applied to the verification of probabilistic programs [13].

8.4 Formal Verification of Neural Networks

Automated reasoning methods that use neural networks as representation of proof certificates, including our approach, rely on formal verification technologies to check the validity of these neural certificates. Various methods for the formal verification of neural networks have been developed in the last few years, driven by the quest for formal guarantees against adversarial attacks in computer vision [96]. Significant effort has been made towards this goal by using out-of-the-box SMT solvers [87] and, subsequently, developing tailored methods to reason about neural networks. This effort led to the development of many effective tools and algorithms [23, 45, 57, 59, 62–64, 73, 91, 93, 98, 109].

Methods for adversarial attacks reason about neural networks in isolation, while reasoning about neural ranking functions requires reasoning about neural networks together with (1) constraints arising from the encoding of a program which (2) are usually in theories other than real arithmetic such as integer or bit-vector arithmetic. Reasoning about neural networks in combination with other systems has been treated in the context of safety analysis of neural network controllers for dynamical systems [10, 61, 90, 99, 104]. Methods of this kind apply abstract interpretation for a bounded number of steps or compute invariants. Verification of neural networks under different theories has been considered for binarized and quantized neural networks, in isolation from other systems, specifically for adversarial attack problems [8, 12, 49, 58, 74].

In our work, we use off-the-shelf SMT solving to check neural ranking functions because, in our experiments, we rely on relatively small networks. While we observe that for a wide variety of problems small networks are sufficient, we do not preclude that our method may benefit from using larger networks. Using larger networks may pose limits to the scalability of off-the-shelf SMT solvers. Our work adds a novel problem to the spectrum of formal verification questions for neural networks, contributing to their relevance to software engineering applications beyond robustness to adversarial attacks.

9 CONCLUSION

We introduced a termination analysis method that takes advantage of neural networks by learning a ranking function candidate from sampled execution traces. This neural ranking function is subsequently verified using formal methods. We provide a prototype implementation of this method called *v*Term. When using tiny neural networks with one hidden layer and a straight-forward training script we solved 76.6% of the problems in a standard set of benchmarks for termination analysis performing comparably to state-of-the-art tools. Furthermore, we identify problems with disjunctive or non-linear loop guards where competing tools are unable to prove termination. We show, experimentally, that *v*Term is able to solve these types of problems by creating a separate set of benchmarks that feature such loop guards. On this set, *v*Term can solve 94.3% of the problems, outperforming competing tools.

Our result suggests future research both in machine learning and formal verification. Learning proof certificates from examples applies not only to imperative programs, but also to functional programming and logic. Moreover, separating proof learning from formal proof checking may also apply to further verification tasks such as non-termination [54], which is difficult for conventional formal approaches.

DATA AVAILABILITY STATEMENT

The problem sets and the code of our prototype tool *v*Term used in this experimental evaluation are available at [50].

ACKNOWLEDGMENTS

We are grateful to all reviewers for their helpful suggestions. This work was in part supported by the Oxford-DeepMind Graduate Scholarship, the Engineering and Physical Sciences Research Council (EPSRC) and the HICLASS project, a partnership between the Aerospace Technology Institute (ATI), Department for Business, Energy & Industrial Strategy (BEIS), and Innovate UK (project 113213).

A TRACING

A trace is a sequence of snapshots (observations) of the program's state as the program runs. Traces are thus generated dynamically, by running the program. The process of tracing takes two inputs: the program that is to be traced and a list of program locations in the program where a snapshot of the state is to be taken. In our case these locations are the loop heads in the program. Our approach is conceptually simple and independent of the platform and programming language. As we consider Java, we give implementation details specific to the Java environment and the Java Virtual Machine (JVM), but our approach could also be applied to more abstract models of computation.

Tracing consists of three steps: *input sampling*, *execution*, and *snapshot*. We describe each of these steps below.

Input sampling. Termination analysis is commonly applied to program fragments that contain some initialisation and a (possibly nested) loop. Therefore, we work with programs that are not closed, but require inputs. We only consider deterministic programs, i.e., two traces that are generated with the same sequence of inputs are identical. We use two sampling strategies based on a Gaussian distribution: *pairwise anticorrelated sampling* (PAS) and *Gaussian sampling*. PAS uses a multivariate normal distribution where we enforce the same variance for all inputs except for two randomly chosen inputs. For these two chosen variables we create a covariance. Hence, this is a standard Gaussian distribution for all variables except for two where we enforce a covariance. Gaussian sampling, on the other hand, is a sampling strategy where each variable is sampled independently from another from a Gaussian distribution with a variance of 1000 and no covariance.

Execution. We start executing the program with the sampled arguments. We maintain control over the JVM during the execution using the Java Virtual Machine Tool Interface (JVMTI). Once we hit a loop head location, we halt the execution and take a snapshot.

Snapshot. Using the JVMTI we have access to the Local Variable Table (LVT). The LVT contains all local variables of the function. We create a memory snapshot by iterating through the LVT and reading the values of every variable that is in scope at the given location. For variables that are out of scope, we record a placeholder default value (which depends on the type of the variable). Since the number of local variables does not change, the size of the snapshots is always the same. Once the snapshot is collected, we append it to the trace of the current program. If the maximum length of a trace is reached we force a termination of the virtual machine, otherwise we resume the execution. The resulting list of snapshots constitutes an execution trace. The goal is to sample the input data in such a way that we achieve a high coverage of the function and the data therefore best represent a possible ranking argument while keeping the required number of program runs low. Our experiments with different sampling strategies show that PAS exhibits better performance than multivariate gaussian sampling. When using the same neural network with 10 neurons we solve 79% of problem using traces obtained with PAS and 74.8% when using multivariate gaussian sampling. It should be noted that both sampling techniques are extremely simplistic. The results may be further improved by utilising more sophisticated test input generation or fuzzing [11, 24, 29, 30, 80, 105, 108].

B VERIFICATION

We verify that a candidate (monolithic) neural raking function $f\colon \Theta \times \mathbb{R}^n \to \mathbb{R}$ is a valid neural ranking function for program $P = (S, T)$ by verifying that it decreases every time a loop header is encountered. For this purpose, we construct a symbolic encoding of the transition relation between every two loop headers T_H. Thus the verification question corresponds to that of determining whether, for any two states between loop headers that also satisfy an auxiliary invariant, the trained neural ranking function decreases by $\delta > 0$. This corresponds to checking the following validity question:

$$\forall s, s'\colon (s, s') \in T_H \wedge s \in A \implies f(\theta; \omega(s)) \geq f(\theta; \omega(s')) + \delta \quad (16)$$

Note that θ is constant is this formula. Then, if this formula is valid, we have that $f(\theta; \cdot)$ is a valid ranking function. We verify this by checking the dual satisfiability question using an SMT solver. The dual question is that of finding a counterexample where the candidate does not decrease by δ, that is, the following formula:

$$\exists s, s'\colon \underbrace{(s, s') \in T_H \wedge s \in A \wedge f(\theta; \omega(s)) < f(\theta; \omega(s')) + \delta}_{\varphi} \quad (17)$$

If the quantifier-free formula φ is determined unsatisfiable by the SMT solver, then the ranking function is valid.

Encoding φ involves encoding constraints for the program T_H and A, constraints for the neural ranking function f, and the interface between them which is the observation function ω. We encode the transition relation between loop headers T_H using a single static assignment encoding, which introduces intermediate variables after each assignment and encodes operations using appropriate arithmetic expressions. This is similar to a bounded model checking encoding [33, 39], which is possible because every run between adjacent loop headers has necessarily fixed length. According to the semantics one wants to consider, the program can be encoded in the theory of integers or the theory of bit-vectors (in our experiments, we use the theory of integers). Also f and ω can be seen as bounded programs (note that f is a feed-forward network) and therefore can be encoded similarly. Our neural networks are compositions of linear layers and ReLU activation functions, which result in first-order logic formulae in the theory of reals with linear arithmetic. Notably, we use much smaller networks compared to common machine learning applications, and we argue that these are sufficient to solve a broad variety of termination problems. Also our program encoding is smaller than those generated by bounded model checkers, as it involves at most one loop unrolling. For this reason, our overall encoding ultimately results in formulae that are efficiently solvable by modern SMT solvers.

Identifying auxiliary invariants A that are strong enough for termination analysis is difficult (encoding them is straightforward). Our verification procedure uses a heuristic that syntactically extracts constraints from the program (e.g. from conditional statements) and checks whether these are valid loop invariants for A using the SMT solver. As it turns out, this naive heuristic was sufficient to obtain the results presented in this paper. Using more sophisticated loop invariant generation methods can only improve the effectiveness of our tool and is subject of future investigation. Methods for generating loop invariants include procedure based on theorem provers [65], constraint based invariant synthesis [16, 17, 68]. Invariants for Java Programs have been constructed using symbolic execution [75]. Tools for the discovery of invariants from trace data include Daikon [46] and DIG [76].

Similarly, for lexicographic neural ranking functions we verify the validity of the conditions in Eq. (13) and (10) over each loop header and respective output component of the neural ranking function. Let $H = \{h_1, \ldots, h_m\}$ be the set of loop headers, then for every $i = 1, \ldots, m$ we verify the validity of the following conditions:

$$\begin{aligned}
&\forall s, s'\colon (s, s') \in T_{\{h_i\}} \wedge s \in A_i \implies f_i(\theta; \omega(s)) \geq f_i(\theta; \omega(s')) + \delta \\
&\forall s, s'\colon (s, s') \in T_{\{h_i\}} \wedge s \in A_i \implies f_{i-1}(\theta; \omega(s)) \geq f_{i-1}(\theta; \omega(s')) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\
&\forall s, s'\colon (s, s') \in T_{\{h_i\}} \wedge s \in A_i \implies f_1(\theta; \omega(s)) \geq f_1(\theta; \omega(s'))
\end{aligned} \quad (18)$$

Note that each condition can be checked independently: the lexicographic argument is violated if any of the conditions is violated, otherwise it is valid. We remark that unlike the monolithic case, $T_{\{h_i\}}$ may represent runs of arbitrarily length when the loop with header h_i has nested loops. To encode $T_{\{h_i\}}$ as a bounded problem we substitute every inner loop with a summary. Several methods have been developed for this purpose [66, 100, 111]. As a heuristic, we construct a loop summary that encodes the invariance of all variables that are never assigned within the loop, together with a transition invariant that encodes the respective (and previously verified) lexicographic component in the neural ranking function and the respective auxiliary invariant. Using or developing more sophisticated summarisation techniques is matter of future research.

REFERENCES

[1] 2022. *OpenSSL Security Advisory [15 March 2022].* https://www.openssl.org/news/secadv/20220315.txt

[2] Alessandro Abate, Daniele Ahmed, Alec Edwards, Mirco Giacobbe, and Andrea Peruffo. 2021. FOSSIL: a software tool for the formal synthesis of Lyapunov functions and barrier certificates using neural networks. In *HSCC*. ACM, 24:1–24:11.

[3] Alessandro Abate, Daniele Ahmed, Mirco Giacobbe, and Andrea Peruffo. 2021. Formal Synthesis of Lyapunov Neural Networks. *IEEE Control. Syst. Lett.* 5, 3 (2021), 773–778.

[4] Alessandro Abate, Mirco Giacobbe, and Diptarko Roy. 2021. Learning Probabilistic Termination Proofs. In *CAV (2) (LNCS, Vol. 12760)*. Springer, 3–26.

[5] Elvira Albert, Puri Arenas, Samir Genaim, German Puebla, and Damiano Zanardini. 2007. COSTA: Design and Implementation of a Cost and Termination Analyzer for Java Bytecode. In *FMCO (LNCS, Vol. 5382)*. Springer, 113–132.

[6] Frances E. Allen. 1970. Control Flow Analysis. In *Proceedings of a Symposium on Compiler Optimization*. ACM, 1–19.

[7] Yoav Alon and Cristina David. 2022. Using Graph Neural Networks for Program Termination. In *ESEC/SIGSOFT FSE*. ACM. To appear.

[8] Guy Amir, Haoze Wu, Clark W. Barrett, and Guy Katz. 2021. An SMT-Based Approach for Verifying Binarized Neural Networks. In *TACAS (2) (LNCS, Vol. 12652)*. Springer, 203–222.

[9] Thomas Arts and Jürgen Giesl. 2000. Termination of term rewriting using dependency pairs. *Theor. Comput. Sci.* 236, 1-2 (2000), 133–178.

[10] Edoardo Bacci, Mirco Giacobbe, and David Parker. 2021. Verifying Reinforcement Learning up to Infinity. In *IJCAI*. ijcai.org, 2154–2160.

[11] Mauro Baluda, Giovanni Denaro, and Mauro Pezzè. 2016. Bidirectional Symbolic Analysis for Effective Branch Testing. *IEEE Trans. Software Eng.* 42, 5 (2016), 403–426.

[12] Teodora Baluta, Shiqi Shen, Shweta Shinde, Kuldeep S. Meel, and Prateek Saxena. 2019. Quantitative Verification of Neural Networks and Its Security Applications. In *CCS*. ACM, 1249–1264.

[13] Jialu Bao, Nitesh Trivedi, Drashti Pathak, Justin Hsu, and Subhajit Roy. 2022. Data-Driven Invariant Learning for Probabilistic Programs. In *CAV (1) (LNCS, Vol. 13371)*. Springer, 33–54.

[14] Amir M. Ben-Amram and Samir Genaim. 2014. Ranking Functions for Linear-Constraint Loops. *J. ACM* 61, 4 (2014), 26:1–26:55.

[15] Dirk Beyer. 2020. Advances in Automatic Software Verification: SV-COMP 2020. In *TACAS (2) (LNCS, Vol. 12079)*. Springer, 347–367.

[16] Dirk Beyer, Thomas A. Henzinger, Rupak Majumdar, and Andrey Rybalchenko. 2007. Invariant Synthesis for Combined Theories. In *VMCAI (LNCS, Vol. 4349)*. Springer, 378–394.

[17] Dirk Beyer, Thomas A. Henzinger, Rupak Majumdar, and Andrey Rybalchenko. 2007. Path invariants. In *PLDI*. ACM, 300–309.

[18] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. 2005. Linear Ranking with Reachability. In *CAV (LNCS, Vol. 3576)*. Springer, 491–504.

[19] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. 2005. The Polyranking Principle. In *ICALP (LNCS, Vol. 3580)*. Springer, 1349–1361.

[20] Marc Brockschmidt, Yuxin Chen, Pushmeet Kohli, Siddharth Krishna, and Daniel Tarlow. 2017. Learning Shape Analysis. In *SAS (LNCS, Vol. 10422)*. Springer, 66–87.

[21] Marc Brockschmidt, Byron Cook, Samin Ishtiaq, Heidy Khlaaf, and Nir Piterman. 2016. T2: Temporal Property Verification. In *TACAS (LNCS, Vol. 9636)*. Springer, 387–393.

[22] Marc Brockschmidt, Carsten Otto, Christian von Essen, and Jürgen Giesl. 2010. Termination Graphs for Java Bytecode. In *Verification, Induction, Termination Analysis (LNCS, Vol. 6463)*. Springer, 17–37.

[23] Rudy Bunel, Ilker Turkaslan, Philip H. S. Torr, Pushmeet Kohli, and Pawan Kumar Mudigonda. 2018. A Unified View of Piecewise Linear Neural Network Verification. In *NeurIPS*. 4795–4804.

[24] Cristian Cadar and Koushik Sen. 2013. Symbolic execution for software testing: three decades later. *Commun. ACM* 56, 2 (2013), 82–90.

[25] Ya-Chien Chang, Nima Roohi, and Sicun Gao. 2019. Neural Lyapunov Control. In *NeurIPS*. 3240–3249.

[26] Hong-Yi Chen, Cristina David, Daniel Kroening, Peter Schrammel, and Björn Wachter. 2018. Bit-Precise Procedure-Modular Termination Analysis. *ACM Trans. Program. Lang. Syst.* 40, 1 (2018), 1:1–1:38.

[27] Jianhui Chen and Fei He. 2020. Proving Termination by k-Induction. In *ASE*. IEEE, 1239–1243.

[28] Shaoru Chen, Mahyar Fazlyab, Manfred Morari, George J. Pappas, and Victor M. Preciado. 2021. Learning Lyapunov functions for hybrid systems. In *HSCC*. ACM, 13:1–13:11.

[29] Tsong Yueh Chen, Fei-Ching Kuo, Robert G. Merkel, and T. H. Tse. 2010. Adaptive Random Testing: The ART of test case diversity. *J. Syst. Softw.* 83, 1 (2010), 60–66.

[30] Tsong Yueh Chen, Hing Leung, and I. K. Mak. 2004. Adaptive Random Testing. In *ASIAN (LNCS, Vol. 3321)*. Springer, 320–329.

[31] Yu-Fang Chen, Matthias Heizmann, Ondrej Lengál, Yong Li, Ming-Hsien Tsai, Andrea Turrini, and Lijun Zhang. 2018. Advanced automata-based algorithms for program termination checking. In *PLDI*. ACM, 135–150.

[32] Yinghua Chen, Bican Xia, Lu Yang, Naijun Zhan, and Chaochen Zhou. 2007. Discovering Non-linear Ranking Functions by Solving Semi-algebraic Systems. In *ICTAC (LNCS, Vol. 4711)*. Springer, 34–49.

[33] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. 2004. A Tool for Checking ANSI-C Programs. In *TACAS (LNCS, Vol. 2988)*. Springer, 168–176.

[34] Byron Cook, Sumit Gulwani, Tal Lev-Ami, Andrey Rybalchenko, and Mooly Sagiv. 2008. Proving Conditional Termination. In *CAV (LNCS, Vol. 5123)*. Springer, 328–340.

[35] Byron Cook, Daniel Kroening, Philipp Rümmer, and Christoph M. Wintersteiger. 2013. Ranking function synthesis for bit-vector relations. *Formal Methods Syst. Des.* 43, 1 (2013), 93–120.

[36] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. 2006. Termination proofs for systems code. In *PLDI*. ACM, 415–426.

[37] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. 2006. Terminator: Beyond Safety. In *CAV (LNCS, Vol. 4144)*. Springer, 415–418.

[38] Byron Cook, Abigail See, and Florian Zuleger. 2013. Ramsey vs. Lexicographic Termination Proving. In *TACAS (LNCS, Vol. 7795)*. Springer, 47–61.

[39] Lucas C. Cordeiro, Pascal Kesseli, Daniel Kroening, Peter Schrammel, and Marek Trtík. 2018. JBMC: A Bounded Model Checking Tool for Verifying Java Bytecode. In *CAV (1) (LNCS, Vol. 10981)*. Springer, 183–190.

[40] Patrick Cousot. 2005. Proving Program Invariance and Termination by Parametric Abstraction, Lagrangian Relaxation and Semidefinite Programming. In *VMCAI (LNCS, Vol. 3385)*. Springer, 1–24.

[41] Patrick Cousot and Radhia Cousot. 2012. An abstract interpretation framework for termination. In *POPL*. ACM, 245–258.

[42] Hongkai Dai, Benoit Landry, Lujie Yang, Marco Pavone, and Russ Tedrake. 2021. Lyapunov-stable Neural-Network Control. In *Robotics: Science and Systems*.

[43] Charles Dawson, Zengyi Qin, Sicun Gao, and Chuchu Fan. 2021. Safe Nonlinear Control Using Robust Neural Lyapunov-Barrier Functions. In *CoRL (PMLR, Vol. 164)*. PMLR, 1724–1735.

[44] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS (LNCS, Vol. 4963)*. Springer, 337–340.

[45] Rüdiger Ehlers. 2017. Formal Verification of Piece-Wise Linear Feed-Forward Neural Networks. In *ATVA (LNCS, Vol. 10482)*. Springer, 269–286.

[46] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon System for Dynamic Detection of Likely Invariants. *Sci. Comput. Program.* 69, 1-3 (2007), 35–45.

[47] Grigory Fedyukovich, Yueling Zhang, and Aarti Gupta. 2018. Syntax-Guided Termination Analysis. In *CAV (1) (LNCS, Vol. 10981)*. Springer, 124–143.

[48] Robert W. Floyd. 1967. Assigning Meanings to Programs. In *Proceedings of Symposium in Applied Mathematics*, Vol. 19. 19–32.

[49] Mirco Giacobbe, Thomas A. Henzinger, and Mathias Lechner. 2020. How Many Bits Does it Take to Quantize Your Neural Network?. In *TACAS (2) (LNCS, Vol. 12079)*. Springer, 79–97.

[50] Mirco Giacobbe, Daniel Kroening, and Julian Parsert. 2022. Code and problem sets for 'Neural Termination Analysis'. https://doi.org/10.1145/3554332

[51] Jürgen Giesl, Cornelius Aschermann, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Jera Hensel, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, Stephanie Swiderski, and René Thiemann. 2017. Analyzing Program Termination and Complexity Automatically with AProVE. *J. Autom. Reason.* 58, 1 (2017), 3–31.

[52] Jürgen Giesl, Albert Rubio, Christian Sternagel, Johannes Waldmann, and Akihisa Yamada. 2019. The Termination and Complexity Competition. In *TACAS (3) (LNCS, Vol. 11429)*. Springer, 156–166.

[53] Laure Gonnord, David Monniaux, and Gabriel Radanne. 2015. Synthesis of ranking functions using extremal counterexamples. In *PLDI*. ACM, 608–618.

[54] Ashutosh Gupta, Thomas A. Henzinger, Rupak Majumdar, Andrey Rybalchenko, and Ru-Gang Xu. 2008. Proving non-termination. In *POPL*. ACM, 147–158.

[55] Matthias Heizmann, Jürgen Christ, Daniel Dietsch, Evren Ermis, Jochen Hoenicke, Markus Lindenmann, Alexander Nutz, Christian Schilling, and Andreas Podelski. 2013. Ultimate Automizer with SMTInterpol (Competition Contribution). In *TACAS (LNCS, Vol. 7795)*. Springer, 641–643.

[56] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. 2014. Termination Analysis by Learning Terminating Programs. In *CAV (LNCS, Vol. 8559)*. Springer, 797–813.

[57] Patrick Henriksen and Alessio R. Lomuscio. 2020. Efficient Neural Network Verification via Adaptive Refinement and Adversarial Search. In *ECAI (FAIA, Vol. 325)*. IOS Press, 2513–2520.

[58] Thomas A. Henzinger, Mathias Lechner, and Dorde Zikelic. 2021. Scalable Verification of Quantized Neural Networks. In *AAAI*. AAAI Press, 3787–3795.

[59] Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. 2017. Safety Verification of Deep Neural Networks. In *CAV (1) (LNCS, Vol. 10426)*. Springer, 3–29.

[60] Geoffrey Irving, Christian Szegedy, Alexander A. Alemi, Niklas Eén, François Chollet, and Josef Urban. 2016. DeepMath - Deep Sequence Models for Premise Selection. In *NIPS*. 2235–2243.

[61] Radoslav Ivanov, James Weimer, Rajeev Alur, George J. Pappas, and Insup Lee. 2019. Verisig: verifying safety properties of hybrid systems with neural network controllers. In *HSCC*. ACM, 169–178.

[62] Guy Katz, Clark W. Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. 2017. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In *CAV (1) (LNCS, Vol. 10426)*. Springer, 97–117.

[63] Guy Katz, Derek A. Huang, Duligur Ibeling, Kyle Julian, Christopher Lazarus, Rachel Lim, Parth Shah, Shantanu Thakoor, Haoze Wu, Aleksandar Zeljic, David L. Dill, Mykel J. Kochenderfer, and Clark W. Barrett. 2019. The Marabou Framework for Verification and Analysis of Deep Neural Networks. In *CAV (1) (LNCS, Vol. 11561)*. Springer, 443–452.

[64] Panagiotis Kouvaros and Alessio Lomuscio. 2021. Towards Scalable Complete Verification of ReLU Neural Networks via Dependency-based Branching. In *IJCAI*. ijcai.org, 2643–2650.

[65] Laura Kovács and Andrei Voronkov. 2009. Finding Loop Invariants for Programs over Arrays Using a Theorem Prover. In *FASE (LNCS, Vol. 5503)*. Springer, 470–485.

[66] Daniel Kroening, Natasha Sharygina, Stefano Tonetta, Aliaksei Tsitovich, and Christoph M. Wintersteiger. 2008. Loop Summarization Using Abstract Transformers. In *ATVA (LNCS, Vol. 5311)*. Springer, 111–125.

[67] Daniel Kroening, Natasha Sharygina, Aliaksei Tsitovich, and Christoph M. Wintersteiger. 2010. Termination Analysis with Compositional Transition Invariants. In *CAV (LNCS, Vol. 6174)*. Springer, 89–103.

[68] Shuvendu K. Lahiri and Randal E. Bryant. 2004. Indexed Predicate Discovery for Unbounded System Verification. In *CAV (LNCS, Vol. 3114)*. Springer, 135–147.

[69] Ton Chanh Le, Timos Antonopoulos, Parisa Fathololumi, Eric Koskinen, and ThanhVu Nguyen. 2020. DynamiTe: Dynamic Termination and Non-termination Proofs. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 189:1–189:30.

[70] Mathias Lechner, Dorde Zikelic, Krishnendu Chatterjee, and Thomas A. Henzinger. 2022. Stability Verification in Stochastic Control Systems via Neural Network Supermartingales. In *AAAI*. AAAI Press, 7326–7336.

[71] Jan Leike and Matthias Heizmann. 2014. Ranking Templates for Linear Loops. In *TACAS (LNCS, Vol. 8413)*. Springer, 172–186.

[72] Yi Li, Xuechao Sun, Yong Li, Andrea Turrini, and Lijun Zhang. 2019. Synthesizing Nested Ranking Functions for Loop Programs via SVM. In *ICFEM (LNCS, Vol. 11852)*. Springer, 438–454.

[73] Changliu Liu, Tomer Arnon, Christopher Lazarus, Christopher A. Strong, Clark W. Barrett, and Mykel J. Kochenderfer. 2021. Algorithms for Verifying Deep Neural Networks. *Found. Trends Optim.* 4, 3-4 (2021), 244–404.

[74] Nina Narodytska, Shiva Prasad Kasiviswanathan, Leonid Ryzhyk, Mooly Sagiv, and Toby Walsh. 2018. Verifying Properties of Binarized Deep Neural Networks. In *AAAI*. AAAI Press, 6615–6624.

[75] ThanhVu Nguyen, Matthew B. Dwyer, and Willem Visser. 2017. SymInfer: Inferring Program Invariants using Symbolic States. In *ASE*. IEEE Computer Society, 804–814.

[76] ThanhVu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. 2014. DIG: A Dynamic Invariant Generator for Polynomial and Array Invariants. *ACM Trans. Softw. Eng. Methodol.* 23, 4 (2014), 30:1–30:30.

[77] Aditya V. Nori and Rahul Sharma. 2013. Termination proofs from tests. In *ESEC/SIGSOFT FSE*. ACM, 246–256.

[78] Miroslav Olsák, Cezary Kaliszyk, and Josef Urban. 2020. Property Invariant Embedding for Automated Reasoning. In *ECAI (FAIA, Vol. 325)*. IOS Press, 1395–1402.

[79] Carsten Otto, Marc Brockschmidt, Christian von Essen, and Jürgen Giesl. 2010. Automated Termination Analysis of Java Bytecode by Term Rewriting. In *RTA (LIPIcs, Vol. 6)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 259–276.

[80] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-Directed Random Test Generation. In *ICSE*. IEEE Computer Society, 75–84.

[81] Aditya Paliwal, Sarah M. Loos, Markus N. Rabe, Kshitij Bansal, and Christian Szegedy. 2020. Graph Representations for Higher-Order Logic and Theorem Proving. In *AAAI*. AAAI Press, 2967–2974.

[82] Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. 2017. Neuro-Symbolic Program Synthesis. In *ICLR (Poster)*. OpenReview.net.

[83] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *NeurIPS*. 8024–8035.

[84] Andrea Peruffo, Daniele Ahmed, and Alessandro Abate. 2021. Automated and Formal Synthesis of Neural Barrier Certificates for Dynamical Models. In *TACAS (1) (LNCS, Vol. 12651)*. Springer, 370–388.

[85] Andreas Podelski and Andrey Rybalchenko. 2004. A Complete Method for the Synthesis of Linear Ranking Functions. In *VMCAI (LNCS, Vol. 2937)*. Springer, 239–251.

[86] Andreas Podelski and Andrey Rybalchenko. 2004. Transition Invariants. In *LICS*. IEEE Computer Society, 32–41.

[87] Luca Pulina and Armando Tacchella. 2012. Challenging SMT solvers to verify neural networks. *AI Commun.* 25, 2 (2012), 117–135.

[88] Zengyi Qin, Kaiqing Zhang, Yuxiao Chen, Jingkai Chen, and Chuchu Fan. 2021. Learning Safe Multi-agent Control with Decentralized Neural Barrier Certificates. In *ICLR*. OpenReview.net.

[89] Gabriel Ryan, Justin Wong, Jianan Yao, Ronghui Gu, and Suman Jana. 2020. CLN2INV: Learning Loop Invariants with Continuous Logic Networks. In *ICLR*. OpenReview.net.

[90] Christian Schilling, Marcelo Forets, and Sebastián Guadalupe. 2022. Verification of Neural-Network Control Systems by Integrating Taylor Models and Zonotopes. In *AAAI*. AAAI Press, 8169–8177.

[91] David Shriver, Sebastian G. Elbaum, and Matthew B. Dwyer. 2021. Reducing DNN Properties to Enable Falsification with Adversarial Attacks. In *ICSE*. IEEE, 275–287.

[92] Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. 2018. Learning Loop Invariants for Program Verification. In *NeurIPS*. 7762–7773.

[93] Gagandeep Singh, Timon Gehr, Matthew Mirman, Markus Püschel, and Martin T. Vechev. 2018. Fast and Effective Robustness Certification. In *NeurIPS*. 10825–10836.

[94] Fausto Spoto. 2016. The Julia Static Analyzer for Java. In *SAS (LNCS, Vol. 9837)*. Springer, 39–57.

[95] Dawei Sun, Susmit Jha, and Chuchu Fan. 2020. Learning Certified Control Using Contraction Metric. In *CoRL (PMLR, Vol. 155)*. PMLR, 1519–1539.

[96] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus. 2014. Intriguing properties of neural networks. In *ICLR (Poster)*.

[97] Wang Tan and Yi Li. 2021. Synthesis of ranking functions via DNN. *Neural Comput. Appl.* 33, 16 (2021), 9939–9959.

[98] Hoang-Dung Tran, Stanley Bak, Weiming Xiang, and Taylor T. Johnson. 2020. Verification of Deep Convolutional Neural Networks Using ImageStars. In *CAV (1) (LNCS, Vol. 12224)*. Springer, 18–42.

[99] Hoang-Dung Tran, Xiaodong Yang, Diego Manzanas Lopez, Patrick Musau, Luan Viet Nguyen, Weiming Xiang, Stanley Bak, and Taylor T. Johnson. 2020. NNV: The Neural Network Verification Tool for Deep Neural Networks and Learning-Enabled Cyber-Physical Systems. In *CAV (1) (LNCS, Vol. 12224)*. Springer, 3–17.

[100] Aliaksei Tsitovich, Natasha Sharygina, Christoph M. Wintersteiger, and Daniel Kroening. 2011. Loop Summarization and Termination Analysis. In *TACAS (LNCS, Vol. 6605)*. Springer, 81–95.

[101] Caterina Urban. 2013. The Abstract Domain of Segmented Ranking Functions. In *SAS (LNCS, Vol. 7935)*. Springer, 43–62.

[102] Caterina Urban. 2015. FuncTion: An Abstract Domain Functor for Termination - (Competition Contribution). In *TACAS (LNCS, Vol. 9035)*. Springer, 464–466.

[103] Caterina Urban, Arie Gurfinkel, and Temesghen Kahsai. 2016. Synthesizing Ranking Functions from Bits and Pieces. In *TACAS (LNCS, Vol. 9636)*. Springer, 54–70.

[104] Joseph A. Vincent and Mac Schwager. 2021. Reachable Polyhedral Marching (RPM): A Safety Verification Algorithm for Robotic Systems with Deep Neural Network Components. In *ICRA*. IEEE, 9029–9035.

[105] Willem Visser, Corina S. Pasareanu, and Sarfraz Khurshid. 2004. Test Input Generation with Java PathFinder. In *ISSTA*. ACM, 97–107.

[106] Rongchen Xu, Jianhui Chen, and Fei He. 2022. Data-Driven Loop Bound Learning for Termination Analysis. In *ICSE*. ACM, 499–510.

[107] Yue Yuan and Yi Li. 2019. Ranking Function Detection via SVM: A More General Method. *IEEE Access* 7 (2019), 9971–9979.

[108] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. 2021. *The Fuzzing Book*. CISPA Helmholtz Center for Information Security. https://www.fuzzingbook.org/ Retrieved 2021-10-26 15:30:20+02:00.

[109] Huan Zhang, Tsui-Wei Weng, Pin-Yu Chen, Cho-Jui Hsieh, and Luca Daniel. 2018. Efficient Neural Network Robustness Certification with General Activation Functions. In *NeurIPS*. 4944–4953.

[110] Hengjun Zhao, Xia Zeng, Taolue Chen, and Zhiming Liu. 2020. Synthesizing barrier certificates using neural networks. In *HSCC*. ACM, 25:1–25:11.

[111] Shaowei Zhu and Zachary Kincaid. 2021. Termination analysis without the tears. In *PLDI*. ACM, 1296–1311.