Neural Termination Analysis

Mirco Giacobbe [1] **Daniel Kroening** [2] **Julian Parsert** [1]

Abstract

We introduce a novel approach to the automated termination analysis of computer programs: we train neural networks to act as ranking functions. Ranking functions map program states to values that are bounded from below and decrease as the program runs. The existence of a valid ranking function proves that the program terminates. While in the past ranking functions were usually constructed using static analysis, our method learns them from sampled executions. We train a neural network so that its output decreases along execution traces as a ranking function would; then, we use formal reasoning to verify whether it generalises to all possible executions. We present a custom loss function for learning lexicographic ranking functions and use satisfiability modulo theories for verification. Thanks to the ability of neural networks to generalise well, our method succeeds over a wide variety of programs. This includes programs that use data structures from standard libraries. We built a prototype analyser for Java bytecode and show the efficacy of our method over a standard dataset of benchmarks.

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, but also may prevent a program from responding at all. Termination analysis addresses the question of whether, for every possible input, a program halts. This is unsolvable in general, yet tools that work in practice have been developed by industry and academia (Cook et al., 2006a; Heizmann et al., 2013; Brockschmidt et al., 2016; Giesl et al., 2017; Le et al., 2020). Previous tools exclusively rely on formal reasoning, and the use of machine learning for termination analysis is rare. In this paper, we introduce a novel technique that trains neural networks to act as formal proofs of termination.

All authors contributed equally [1]Department of Computer Science, University of Oxford [2]Amazon, Inc. This work was done prior to joining Amazon.

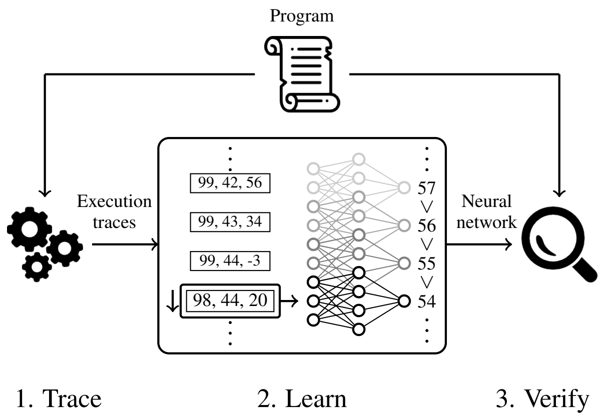


Figure 1: Schema of our procedure.

The standard way to argue that a program terminates is to present a *ranking function* (Floyd, 1967). Ranking functions map program states to values that (i) decrease by a discrete amount after every loop iteration and (ii) are bounded from below. They are certificates of termination: if a ranking function exists, then the program terminates for every possible input. Previous methods construct termination proofs by relying on the soundness of the underlying algorithm (Arts & Giesl, 2000; Bradley et al., 2005a; Cook et al., 2006b). However, finding a proof is much harder than checking that a given candidate proof is valid. When attempting to argue that a given program terminates, humans would look for decreasing patterns among the variables over fictional runs of the program. Then, they would guess a ranking function that fits these patterns.

Our procedure imitates the human approach. We take a program as input and proceed in three steps (see Fig. 1). First, we sample multiple inputs for the program and execute it with each of these inputs. As the program runs, we trace the values of its variables and collect the respective sequences of valuations. Second, using a custom loss function (see Sect. 4), we train a neural network so that its output decreases along these sequences. For programs with nested loops, we train it so that it descends a lexicographic order. We use a neural architecture that is naturally bounded from below, thus we obtain networks that act as ranking functions with respect to the sampled traces. Finally, we verify whether this candidate *neural ranking function* also

```
1  Iterator<Integer> i = list.iterator();
2  while (i.hasNext()) {
3    Integer e = i.next();
4    if (e < 0)
5      i.remove();
6  }
```

Figure 2: Program removing the negative elements of a list.

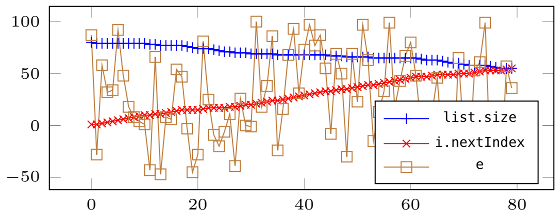


Figure 3: Execution trace for the program in Fig. 2.

decreases along every possible execution. To this end, we SMT solve over a symbolic encoding of the program. Upon an affirmative verification result, our procedure concludes that the program terminates for every possible input.

Learning candidate ranking functions from execution traces is agnostic with respect to the programming language. Our method requires no information about the program other than the traces to form the loss function. It thus applies without modifications to a broad variety of programs, ranging from toy examples that manipulate numerical counters to software that uses data structures. We implemented a prototype analyser for Java bytecode. Using a neural architecture with one hidden layer and a straight-forward training routine, our method discovered ranking functions for 47% of the benchmarks in a standard problem set for termination analysis (Giesl et al., 2019; Beyer, 2020). Moreover, it is able to discover ranking functions for programs that use heap-allocated data structures such as linked lists.

Methods for learning termination arguments from traces have been studied before using automata and template expressions as learning models (Lee et al., 2012; Nori & Sharma, 2013; Heizmann et al., 2014; Urban et al., 2016). Recently, neural networks have been used for discovering loop invariants (Si et al., 2018). Also, they have been applied to the stability analysis of dynamical systems (Chang et al., 2019; Abate et al., 2021). Our method builds upon all these ideas and, for the first time, neural networks are used for termination analysis.

2. Illustrative Example

The reason why a program terminates can be subtle, and the Java program in Fig. 2 is an exemplar. This program operates on a list, which implies that the termination analysis requires reasoning about the heap. However, there is a termination argument that only uses integer sequences. This is hidden within the implementation but can be spotted by examining the traces we obtain when running the program.

We create a variety of lists with random length and random elements and, using each of them as input, perform multiple runs of the program. We record the values of all integer variables that appear in the program, including object members, every time the program reaches line 2 (the loop header). We obtain multiple sequences of n-dimensional vectors of values, where n is the number of traced variables. For presentation, we select three key variables along one trace, `list.size`, `i.nextIndex`, and `e`, and plot them in Fig. 3. Note that this is a real trace, and these member variables appear in the current implementation of `java.util.LinkedList`'s iterator in OpenJDK[1].

To learn a neural ranking function for this example, we employ the simple neural network realising the function

$$f_\theta(x) = \mathrm{ReLU}(w_1 \cdot \texttt{list.size} + \cdots + w_n \cdot \texttt{e}), \qquad (1)$$

where the input $x = (\texttt{list.size}, \dots, \texttt{e}) \in \mathbb{R}^n$ is a vector of variables valuations, and the parameter $\theta = (w_1, \dots, w_n) \in \mathbb{R}^n$ is a vector of learnable weights. This is a neural network with n input neurons, one output neuron with ReLU activation function, and no hidden layers. The input neurons are fully connected to the output neuron. This function has a lower bound (zero) and is thus bounded from below. The final step to obtain a candidate ranking function is to learn a parameter θ so that f_θ decreases along all sampled execution traces.

We train the parameters of this neural network so that its output value decreases by $\delta > 0$ after every consecutive pair of program states. For this purpose, we collect all pairs $(\boldsymbol{x}, \boldsymbol{y}) \in \mathbb{R}^n \times \mathbb{R}^n$ such that $\boldsymbol{y}$ appears immediately after $\boldsymbol{x}$ in some trace. This results in the dataset of pairs $\{(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)})\}_{i=1}^k$. To train our neural network, we solve the minimisation problem

$$\operatorname*{arg\,min}_{\theta \in \mathbb{R}^n} \frac{1}{k} \sum_{i=1}^{k} \max\{f_\theta(\boldsymbol{y}^{(i)}) - f_\theta(\boldsymbol{x}^{(i)}) + \delta, 0\} \qquad (2)$$

using standard optimisation algorithms. Provided that the traces are well sampled, the optimal parameter $\theta^\star$ yields the function

$$f_{\theta^\star}(x) = \mathrm{ReLU}(\texttt{list.size} - \texttt{i.nextIndex}), \qquad (3)$$

after an appropriate filtering of the numerical noise in $\theta^\star$. The formal verifier confirms that $f_{\theta^\star}$ is a valid ranking function for the program in Fig. 2. Thus this program terminates for every possible input list.

[1] https://github.com/openjdk/jdk

```
while (i < k)
  i++;
```
(a)

```
while (i > k)
  i++;
```
(b)

Figure 4: Terminating and non-terminating programs.

By examining the numerical data in the execution traces our method can discover termination arguments for programs that use data structures in the same manner as for numerical programs. For many practical problems, this implies that we do not require any heap-specific reasoning machinery.

3. Termination Analysis

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 P is a pair (S, T) where S is a countable (possibly infinite) set of states and $T \subset S \times S$ is a transition relation. A *run* of P is any sequence of states

$$s_0, s_1, s_2, \ldots \tag{4}$$

such that $(s_i, s_{i+1}) \in T$ for all $i \geq 0$. A state without any successors is a terminating state. We say that a program terminates if all its runs are finite.

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

$$f(t) < f(s) \quad \text{for all } (s, t) \in T \tag{5}$$

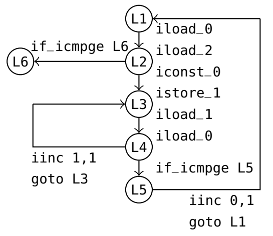
and the relation $(W, <)$ is well founded (Floyd, 1967). The standard example is a function that maps the traces to sequences of numbers that (i) decrease a discrete amount and (ii) are bounded from below.

The existence of a ranking function proves that the program terminates for every possible input. For instance, consider the programs of Fig. 4. The function $f(\mathtt{i}, \mathtt{k}) = -\mathtt{i}$ strictly decreases in both programs. However, in neither case this function has a bound from below, and f is therefore not a ranking function for either of the programs. On the other hand, $g(\mathtt{i}, \mathtt{k}) = \mathtt{k} - \mathtt{i}$ is a ranking function for Fig. 4a, because it decreases with every iteration and is bounded from below by zero. By contrast, the program in Fig. 4b does not admit a ranking function. For some inputs, this loop never stops, assuming that the program variables do not wrap around (termination analysis that considers wraparound is discussed in related literature (Cook et al., 2013a)).

Ranking functions are not limited to maps to integers or rational numbers with a minimum, but can also be maps to tuples with lower-bounded lexicographic orders.

Our program model can be seen as the operational semantics of an imperative programming language, where every state is associated to a control location (by the program counter) over a control flow graph.

```
while (i < k) {
  j = 0;
  while (j < i) {
    j++;
  }
  i++;
}
```
(a)



(b)

Figure 5: Program with nested counting and its CFG.

Control Flow Graph A control flow graph (CFG) for program P consists of 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. Let $\lambda \colon S \to L$ map every state to a control location. A CFG satisfies that $(\lambda(s), \lambda(t)) \in E$ for all $(s, t) \in T$.

For a Java program, a CFG can be directly obtained from its bytecode. An example is given in Fig. 5b, where control locations correspond to the source and target positions of jump instructions. We increase the efficiency of the termination analysis by determining the loop headers in the CFG. The loop headers are the dominators (entry locations) of the strongly connected components in the graph (Allen, 1970); for instance, the loop headers for Fig. 5b are L1 and L3. We reduce the computational cost of the analysis by learning ranking functions that decrease every time a loop header is encountered, rather than between every state transition. Discovering a function that decreases at loop headers is sufficient for proving termination.

4. Training Neural Ranking Functions

Our method proceeds in three stages: tracing, learning, and verification, as depicted in Fig. 1. At tracing stage, we sample multiple runs of the program and produce a dataset of transition observations. These are pairs of consecutive (possibly partial) state observations of the program state, collected during execution. Then, at learning stage, we train a neural networks so that its output descends a strict order with respect to all transition observation samples. The network's output may simply decrease by a discrete amount $\delta > 0$ or, using a network with multiple outputs, may decrease lexicographically. Finally, verification checks against the original program whether the neural network is a rank-

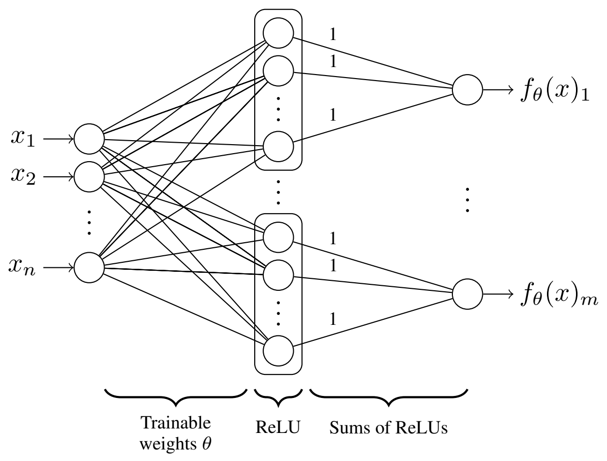


Figure 6: “Sum of ReLUs” architecture for lexicographic neural ranking functions.

ing function for every possible input. To this end, we use SMT solving over a symbolic encoding of the program and the network.

Transition Observations and Traces Our dataset is a collection of observations from the transitive closure of the transition relation T. The state of a program is large and complex and contains internal information that a neural network cannot handle directly. The standard approach to address this issue is to define an *embedding*. To this end, we define an observation function $\omega\colon S \to \mathbb{R}^n$, which extracts appropriate input to the neural network from a program state. States are recorded at key locations during a run, specifically, at loop headers; then ω is possibly partial. The observation function converts a run (see Eq. 4) into a *trace*

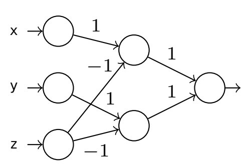
$$x_0, x_1, x_2, \dots \tag{6}$$

where, for all $i > 0$, the relation between $x_{i-1} \in \mathbb{R}^n$ and $x_i \in \mathbb{R}^n$ is that the earlier has been observed before the former. In other words, for all $i > 0$, we have that $x_{i-1} = \omega(s_{i-1})$ and $x_i = \omega(s_i)$ for some $(s_{i-1}, s_i) \in T^+$. Every pair of consecutive observations from a trace is a transition observation. Our dataset $\{(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)})\}_{i=1}^k$ is a finite set of transition observations.

We train a ranking function model $f_\theta\colon \mathbb{R}^n \to \mathbb{R}^m$ (parameterised by θ) with n inputs and m outputs so that it acts as a ranking function along every trace. First, we require that f_θ is bounded from below a priori. We enforce this condition over the neural architecture by choosing appropriate output activation functions and architecture. We then train the network so that its output strictly decreases with every transition. We discuss two loss functions and a neural network architecture for this purpose.

```
while (x > z || y > z) {
  if (x > z)
    x--;
  else if (y > z)
    y--;
}
```

(a)



(b)

Figure 7: Program with SOR ranking function.

Monolithic Ranking Loss A *monolithic neural ranking function* is the special case for which $m = 1$. In this case, our objective is obtaining a parameter θ such that the output decreases by a discrete amount $\delta > 0$ with respect to every transition observation $(\boldsymbol{x}, \boldsymbol{y})$:

$$f_\theta(\boldsymbol{y}) \le f_\theta(\boldsymbol{x}) - \delta. \tag{7}$$

To this end, we solve the optimisation problem in Eq. (2).

Lexicographic Ranking Loss We train general models ($m > 1$) so their output descends a lexicographic order along the traces. More specifically, we train them so that every transition may strictly decrease any output neuron by δ, as long as all other output neurons with smaller index do not increase. That is, for every transition observation $(\boldsymbol{x}, \boldsymbol{y})$ there exists an index $\boldsymbol{j} \in \{1, \dots, m\}$ such that

$$f_\theta(\boldsymbol{y})_{\boldsymbol{j}} \le f_\theta(\boldsymbol{x})_{\boldsymbol{j}} - \delta \qquad \text{and} \tag{8}$$

$$f_\theta(\boldsymbol{y})_i \le f_\theta(\boldsymbol{x})_i \qquad \text{for all } i < \boldsymbol{j}. \tag{9}$$

To train a *lexicographic neural ranking function*, we augment the dataset so that every transition observation $(\boldsymbol{x}, \boldsymbol{y})$ is associated with a desired lexicographic index $\boldsymbol{j}$. Then, we optimise the average value of the following loss function:

$$\mathcal{L}_\theta(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{j}) = \max\{f_\theta(\boldsymbol{y})_{\boldsymbol{j}} - f_\theta(\boldsymbol{x})_{\boldsymbol{j}} + \delta, 0\} + \sum_{i=1}^{\boldsymbol{j}-1} \max\{f_\theta(\boldsymbol{y})_i - f_\theta(\boldsymbol{x})_i, 0\}. \tag{10}$$

Lexicographic indices are added to the dataset before training. It is a good heuristic to associate indices to the nesting of the corresponding loop header, by increasing index from outermost to innermost nesting.

We minimise the loss function until it attains a value that is sufficiently close to zero. Then, we round all parameters to the highest order of magnitude and pass the model to the formal verifier.

Sum of ReLUs The architecture depicted in Fig. 6 is the most general architecture we consider, and we call it *the sum of ReLUs* (SOR). The SOR architecture features n inputs, m outputs, and one hidden layer. The hidden neurons are partitioned into m groups. Input and hidden layer are fully connected with trainable weights, whereas as each group is independently connected to an output neuron with non-trainable weights fixed to 1. Our experiments (see Sect. 6) show that this architecture, together with an appropriate sampling strategy, succeeds on a large range of programs in practice. The program in Fig. 7a is an exemplar. The program is challenging because the decreasing variable may change during execution, and it is difficult to identify an appropriate ranking function. As it turns out, the simple neural network in Fig. 7b is a correct neural ranking function for this program, and was efficiently learned by our prototype.

5. Implementation

In this section we give an overview of the implementation of ν`Term`. A prototype implementation of the ideas presented in the previous sections for termination analysis of Java bytecode (JBC) programs. Note that this is merely an engineering choice, motivated by the existence of libraries and the widespread usage of the JVM. Furthermore, as noted above, the separation of tracing, learning, and verification makes this approach very modular. Hence, to support a different input language one only has to adjust tracing and verifiication while the implementation of the primary contribution of this paper (the learning) remains unchanged.

Tracing and Sampling Tracing is the first step of our method (Fig. 1). This requires two parts. First, any program requires an input that needs to be generated. Second, the program in question has to be executed with this input. During the execution of the program we take snapshots of the memory. The sequence of all snapshots is what we refer to as the *trace* of a program. By repeating this process with different inputs we collect a large number of samples. Ideally, this sampled data exhibits every possible behaviour of the program.

We limit ourselves to deterministic programs, that is, programs that always exhibit the same behaviour given the same input. We generate the inputs using a pseudorandom generator with one of the following three distribution strategies:

- uniform,
- Gaussian, and
- pairwise anticorrelated sampling (PAS).

The first two simply denote the strategy where for each program input parameter we sample random values with either a uniform or a Gaussian distribution, respectively. The PAS strategy uses a multivariate normal distribution where we create a dependence between two randomly chosen values. Hence, this is a standard Gaussian distribution for all variables except for two where we create an indirect correlation. If enough samples are traced we can expect a sample for each possible pair in the input data. This has the advantage that it keeps the traces short while simultaneously creating diversity in the traces.

Once the input data is generated we can execute the programs with these arguments. Note that the runs of the program are independent, which allows us to parallelise the sampling process. The tracing itself is done by taking snapshots of all local variables and their contents throughout the execution of the target program. All of this functionality is provided by the Java Virtual Machine Tool Interface (JVM TI), which allows us to control and inspect the state of a JVM during the execution of a program.

Learning Once we have obtained the traces of the program we can learn ranking functions. As we are only interested in ranking the strongly connected components (i.e., loops) we analyse the CFG to identify the loop headers. With these we can extract the parts of the trace that are relevant. The memory snapshots at the loop header are joined together and subsequently turned into ranking pairs in a sliding window fashion. These pairs constitute our training data to which we apply a *pairwise learning to rank* algorithm (cf. Sect. 4). All of this is implemented using PyTorch (Paszke et al., 2019).

Verification There are numerous ways to check that a termination certificate (i.e., a ranking function) is correct. In our case, we translate the program into single static assignment (SSA) form, which we then translate into a satisfiability modulo theories (SMT) formula. This SMT formula encodes the requirement that the ranking function is indeed a decreasing measure for every iteration of every loop. Furthermore, it ensures that the relation is well founded. Finally, we use the SMT solver Z3 to prove these assertions (de Moura & Bjørner, 2008).

6. Experiments

In this section we present an evaluation of ν`Term` and discuss results, limitations, and possible improvements.

6.1. Benchmarks and Setup

Our evaluation is conducted on benchmarks gathered from the TermComp Termination Competition (Giesl et al., 2019) and the SV-COMP Software Verification Competition (Beyer, 2020). The problem sets of these competitions cover a wide variety of termination and non-termination problems as well as software verification in general. The TermComp data set focuses on term rewriting rather than program termination. We therefore restrict the evaluation to the two problem sets `Aprove_09` from TermComp and `term-crafted-lit` from SV-COMP. Furthermore, due to the limitations of the prototype implementation we removed problems from both data sets that:

- are non-terminating,
- have multiple function calls thus also recursive ones,
- are non-deterministic.

This left us with 75 problems from `term-crafted-lit` (originally 159) and 38 problems (originally 76) from `Aprove_09`. In addition, we made purely syntactic changes to the `Aprove_09` problems that do not change the difficulty of determining the termination of the program. Since `term-crafted-lit` is comprised of problems from literature on termination analysis translated to C the changes we had to make in this set are profound. Namely, in addition to filtering out problems for the reasons listed above, we also had to translate the problems from C into Java. Again, the underlying difficulty of determining termination remains the same.

The experiments were conducted on Ubuntu 18.04.5 (kernel version 4.15) running on an Intel(R) Xeon(R) Silver 4114 CPU @ 2.20 GHz (with 20 physical cores). As the algorithm is non-deterministic, we ran the benchmarks 10 times.

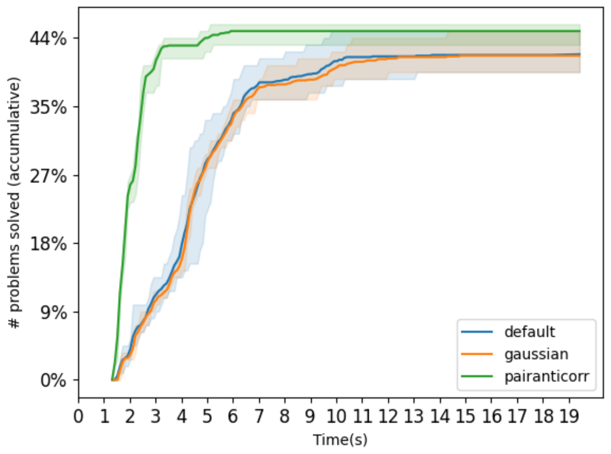


Figure 8: Percentage of problems solved with given time budget (in seconds)

6.2. Results

We ran ν`Term` on the aforementioned benchmarks with three different sampling strategies. The results are presented in Tab. 1. *Pairwise anticorrelated sampling* (PAS) produced the best results, proving termination of 47% of all problems followed by uniform sampling with 41% and Gaussian sampling with 40%. When looking at the wall clock time of running both benchmarks (this includes solved as well as unsolved problems) there is no notable difference between uniform sampling and Gaussian sampling. However, PAS is almost 45% faster on the `Aprove_09` data set and almost 30% faster on `term-crafted-lit`.

This is also evident in Fig. 8, where we plot the percentage of solved problems as a function of time. In particular, PAS solves 80% of all *solved* problems within 2.5 seconds while Gaussian and uniform sampling take over 7 seconds for the same percentile. To evaluate the performance of training alone we also plot the number of training iterations required to find a correct ranking function in Fig. 9.

6.3. Discussion

Neural termination analysis is an approach for synthesising ranking functions using neural networks trained on execution traces. Hence, it is no surprise that the sampling method used which has a large impact on the training data affects the overall performance. In our benchmarks this made a difference of 7% in the number of problems solved. The impact is even larger when looking at the time and the number of iterations required to solve a problem (cf. Fig. 8). To solve 80% of all solved problems PAS required one third of the time that the other sampling strategies required. This large difference in time is unsurprising as PAS generates smaller variable values (cf. Sect. 5) and therefore shorter traces. In most cases, tracing has the largest impact on the overall runtime. However, PAS does not only improve the tracing time but also the quality of the tracing data. This can clearly be seen in Fig. 9, where the number of problems solved (as a percentage of all problems) with respect to the number of training iterations is plotted. It shows that PAS reduces the number of training iterations required to train a correct ranking function. In particular, PAS requires fewer than 50 training iterations to solve 80% of all solved problems, while Gaussian and uniform sampling both solve around 57% of all solved problems with the same number of iterations. We expect the number of iterations to increase as ranking functions and models become more complex. This would also entail more tracing to increase the amount of training data.

Overall, ν`Term` proves termination for 47% of all problems (37% on `term-crafted-lit` and 66% on `Aprove_09`). This result is obtained with one sampling method as well as only one, simple neural network architecture (Sect. 4). In the last TermComp, Aprove (Giesl et al., 2017) was able to solve all problems in the `Aprove_09` set. This is due to the fact that Aprove is a mature tool that has seen a lot of development and contributions from multiple researchers over many years. On the other hand, ν`Term` is the first tool that utilises neural termination analysis and is still under development.

Benchmark	#	νTerm PAS #solved	νTerm PAS time	νTerm uniform #solved	νTerm uniform time	νTerm gaussian	
`term-crafted-lit`	75	28	171.2	23	243.3	24	240.9
`Aprove_09`	38	25	48.5	23	88.3	21	93.4
Total	113	53		46		45	

Table 1: ν`Term` with pairwise anticorrelated sampling, standard uniform distribution and normal distribution

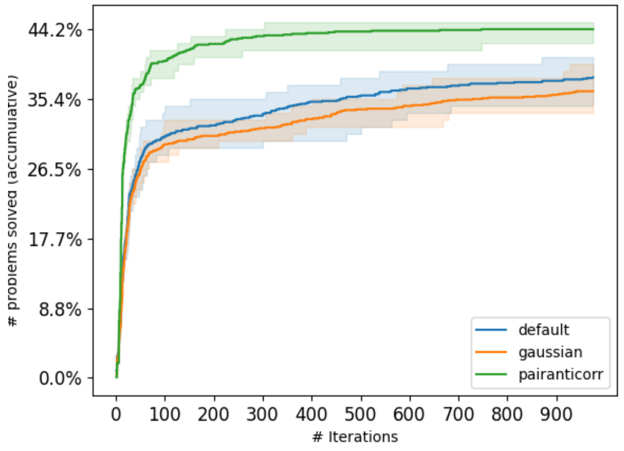


Figure 9: Percentage of problems solved over number of iterations required (accumulative).

6.4. Limitations

Owing to the inherent difficulty of the problem at hand (termination), methods for solving it are necessarily incomplete. Neural termination analysis is no exception. It is easy to construct programs that other methods can prove terminating but where neural termination analysis fails. We give a few examples and discuss possible causes.

Lack of Data Neural termination analysis learns termination arguments from trace data. Hence, any program feature that limits the data that can be gathered is a problem for our approach. An artificial instance of such a behaviour is the following loop:

```
for(int i = 0; i < 1; i++) {
  doSomething();
}
```

Regardless of the sampling strategy, this loop will only do one iteration and we will therefore never have enough training data to learn a ranking function. A realistic instance of this problem that can be found among the benchmarks is a function that computes a logarithm. For example, calculating $\texttt{log}_{50}10000$ only requires 3 iterations. In some simple instances this problem can be solved by judicious sampling. For instance, inputs with small numbers as the first argument and large numbers as the second argument might yield a sufficient amount of training data for the logarithm function. This will not help in other instances, say in the case of a loop with a single iteration.

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

(a)

```
while (i < 100) {
  j = i;
  i = doSomething(i);
  i = j;
  i--;
}
```

(b)

Figure 10: Loops with a multiple apparent decreasing function.

Multiple Apparent Ranking Functions Another limitation is that there might be multiple candidates for the ranking function. Choosing the "wrong" candidate may lead to problems in the verification phase. Consider the loop from Fig. 10a as example. As most machine learning algorithms are non-deterministic, the method we propose might devise the ranking function $f(i,j) := -i$ in some instances and $f(i,j) := j$ in others. However, only the former is correct. One solution for this would be taking control of the initialisation of i and j, sampling them appropriately, and to avoid overfitting j.

Similar problems can be caused by sharing (variables that refer to the same values through assignment), as in Fig. 10b. Considering the memory footprint of that program we find that both i and j could make adequate ranking arguments, as both decrease after every loop. However, verifying the ranking function $f(i,j) := j$ is considerably harder as it requires the auxiliary invariant $i = j$. One way to solve this problem could be to integrate existing methods that discover such invariants (Si et al., 2018; Ernst et al., 2007).

These limitations boil down to two challenges, one is appropriate sampling of traces and the other is identifying auxiliary loop invariants. Without these, our method may fail even on simple programs. Moreover, a static analysis of the program before learning should be done to avoid degenerate cases. Hence, it is apparent that in order to get the best results our method should be used in combination with coventional program analysis algorithms.

7. Related Work

Termination analysis is a large and mature area with a long history. Hence, we will only discuss previous work that is directly related to this paper. State-of-the-art methods for termination analysis can be categorized as either *direct* or *indirect*.

Direct Tools that follow the direct approach to termination analysis construct termination arguments that apply directly to the input program without translating it to a different representation. Such arguments are often found by combining static analysis and constraint solving to synthesise ranking functions. Depending on the class of input programs these constraints can be linear programs (Podelski & Rybalchenko, 2004), semi-definite programs (Cousot, 2005), semi-algebraic systems (Chen et al., 2007), or formulae over SMT theories (Cook et al., 2013a). To deal with complex programs lexicographic ranking functions were introduced. To synthesise such ranking arguments constraint solving has been combined with combinatorial search or abstract interpretation to synthesise lexicographic (Bradley et al., 2005a;b; Leike & Heizmann, 2014), piecewise (Urban, 2013), and disjoint well-founded relations (Cook et al., 2006b; 2013b).

Indirect Indirect termination describes methods that translate the input into a different model of computation and prove that the translated version terminates. This requires a guarantee that termination of the translation implies the termination of the original program. These alternate models of computation can be term rewrite systems (Giesl et al., 2017; Brockschmidt et al., 2010; Otto et al., 2010), constraint logic programs (Spoto, 2016), recurrence relations (Albert et al., 2007), and Büchi automata (Heizmann et al., 2014; Chen et al., 2018).

Termination from Execution Traces To our knowledge there are only two tools that utilise execution traces to synthesise termination arguments: one generates loop bounds and the other uses SMT to fit ranking functions (Nori & Sharma, 2013; Le et al., 2020). The earlier utilises execution to create a quadratic program with linear constraints so that the solution describes a loop bound. The latter, which is implemented in the tool DynamiTe, considers traces by themselves together with their transitive closure. Using SMT solvers this data is fitted to ranking function templates to produce ranking function candidates. If the SMT query is successful, an off-the-shelf reachability analysis tool is used to validate the candidate ranking function.

Ranking Measures The concept of learning rankings is not foreign to the machine learning community (Chen et al., 2009; Burges et al., 2005; Burges, 2010). This is commonly referred to as *machine-learned ranking* (MLR) and is for example used for information retrieval (Liu, 2011) and recommender systems (Karatzoglou et al., 2013), and search engines. However, machine-learned ranking has never been applied to termination analysis.

8. Conclusion

We introduced the first termination analysis method that takes advantage of neural networks. Inspired by how a human would find a termination argument, our approach guesses a ranking function candidate from sampled execution traces. We built a prototype analyser for Java bytecode. Compared to approaches based on formal reasoning, our approach is simple to implement: together with infrastructure for running programs and checking ranking functions, we implemented our prototype using a simple neural architecture with one hidden layer and a straight-forward training script. We evaluated our prototype on a standard set of benchmarks for termination analysis and obtained ranking functions for 47% of the problems.

Our results are the basis for future research in machine learning and formal verification. Learning proofs from examples could be applied not only to imperative programs, but also to functional programming and logic. Furthermore, we discuss how separating learning from verifying allows us to discover termination proofs without having to sacrifice correctness. Our method does not directly (without modification) apply to non-termination (Gupta et al., 2008). To this end, alternative proof techniques and machine learning models should be explored.

Acknowledgments

We thank Mohammanhosein Hasanbeig and Hadrien Pouget for revising parts of this manuscript. This work was in part supported by the HICLASS project (113213), a partnership between the Aerospace Technology Institute (ATI), Department for Business, Energy & Industrial Strategy (BEIS), Innovate UK. The research was also in part funded by the Oxford-DeepMind Graduate Scholarship as well as the Engineering and Physical Sciences Research Council (EPSRC) Doctoral Training Partnership.

References

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

Albert, E., Arenas, P., Genaim, S., Puebla, G., and Zanardini, D. COSTA: Design and implementation of a cost and termination analyzer for Java bytecode. In *FMCO*, volume 5382 of *Lecture Notes in Computer Science*, pp. 113–132. Springer, 2007.

Allen, F. E. Control flow analysis. In *Proceedings of a Symposium on Compiler Optimization*, pp. 1–19. ACM, 1970.

Arts, T. and Giesl, J. Termination of term rewriting using dependency pairs. *Theor. Comput. Sci.*, 236(1-2):133–178, 2000.

Beyer, D. Advances in automatic software verification: SV-COMP 2020. In *TACAS (2)*, volume 12079 of *Lecture Notes in Computer Science*, pp. 347–367. Springer, 2020.

Bradley, A. R., Manna, Z., and Sipma, H. B. Linear ranking with reachability. In *CAV*, volume 3576 of *Lecture Notes in Computer Science*, pp. 491–504. Springer, 2005a.

Bradley, A. R., Manna, Z., and Sipma, H. B. The polyranking principle. In *ICALP*, volume 3580 of *Lecture Notes in Computer Science*, pp. 1349–1361. Springer, 2005b.

Brockschmidt, M., Otto, C., von Essen, C., and Giesl, J. Termination graphs for Java bytecode. In *Verification, Induction, Termination Analysis*, volume 6463 of *Lecture Notes in Computer Science*, pp. 17–37. Springer, 2010.

Brockschmidt, M., Cook, B., Ishtiaq, S., Khlaaf, H., and Piterman, N. T2: Temporal property verification. In *TACAS*, volume 9636 of *Lecture Notes in Computer Science*, pp. 387–393. Springer, 2016.

Burges, C. J. From RankNet to LambdaRank to LambdaMART: An overview. Technical Report MSR-TR-2010-82, June 2010.

Burges, C. J. C., Shaked, T., Renshaw, E., Lazier, A., Deeds, M., Hamilton, N., and Hullender, G. N. Learning to rank using gradient descent. In *ICML*, volume 119 of *ACM International Conference Proceeding Series*, pp. 89–96. ACM, 2005.

Chang, Y., Roohi, N., and Gao, S. Neural Lyapunov control. In *NeurIPS*, pp. 3240–3249, 2019.

Chen, W., Liu, T., Lan, Y., Ma, Z., and Li, H. Ranking measures and loss functions in learning to rank. In *NIPS*, pp. 315–323. Curran Associates, Inc., 2009.

Chen, Y., Xia, B., Yang, L., Zhan, N., and Zhou, C. Discovering non-linear ranking functions by solving semi-algebraic systems. In *ICTAC*, volume 4711 of *Lecture Notes in Computer Science*, pp. 34–49. Springer, 2007.

Chen, Y., Heizmann, M., Lengál, O., Li, Y., Tsai, M., Turrini, A., and Zhang, L. Advanced automata-based algorithms for program termination checking. In *PLDI*, pp. 135–150. ACM, 2018.

Cook, B., Podelski, A., and Rybalchenko, A. Terminator: Beyond safety. In *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pp. 415–418. Springer, 2006a.

Cook, B., Podelski, A., and Rybalchenko, A. Termination proofs for systems code. In *PLDI*, pp. 415–426. ACM, 2006b.

Cook, B., Kroening, D., Rümmer, P., and Wintersteiger, C. M. Ranking function synthesis for bit-vector relations. *Formal Methods Syst. Des.*, 43(1):93–120, 2013a.

Cook, B., See, A., and Zuleger, F. Ramsey vs. lexicographic termination proving. In *TACAS*, volume 7795 of *Lecture Notes in Computer Science*, pp. 47–61. Springer, 2013b.

Cousot, P. Proving program invariance and termination by parametric abstraction, Lagrangian relaxation and semidefinite programming. In *VMCAI*, volume 3385 of *Lecture Notes in Computer Science*, pp. 1–24. Springer, 2005.

de Moura, L. M. and Bjørner, N. Z3: An efficient SMT solver. In *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pp. 337–340. Springer, 2008.

Ernst, M. D., Perkins, J. H., Guo, P. J., McCamant, S., Pacheco, C., Tschantz, M. S., and Xiao, C. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, 2007.

Floyd, R. W. Assigning meanings to programs. In *Proceedings of Symposium in Applied llathematics*, 1967.

Giesl, J., Aschermann, C., Brockschmidt, M., Emmes, F., Frohn, F., Fuhs, C., Hensel, J., Otto, C., Plücker, M., Schneider-Kamp, P., Ströder, T., Swiderski, S., and Thiemann, R. Analyzing program termination and complexity automatically with AProVE. *J. Autom. Reason.*, 58(1): 3–31, 2017.

Giesl, J., Rubio, A., Sternagel, C., Waldmann, J., and Yamada, A. The termination and complexity competition. In *TACAS (3)*, volume 11429 of *Lecture Notes in Computer Science*, pp. 156–166. Springer, 2019.

Gupta, A., Henzinger, T. A., Majumdar, R., Rybalchenko, A., and Xu, R. Proving non-termination. In *POPL*, pp. 147–158. ACM, 2008.

Heizmann, M., Christ, J., Dietsch, D., Ermis, E., Hoenicke, J., Lindenmann, M., Nutz, A., Schilling, C., and Podelski, A. Ultimate automizer with SMTInterpol (competition contribution). In *TACAS*, volume 7795 of *Lecture Notes in Computer Science*, pp. 641–643. Springer, 2013.

Heizmann, M., Hoenicke, J., and Podelski, A. Termination analysis by learning terminating programs. In *CAV*, volume 8559 of *Lecture Notes in Computer Science*, pp. 797–813. Springer, 2014.

Karatzoglou, A., Baltrunas, L., and Shi, Y. Learning to rank for recommender systems. In *RecSys*, pp. 493–494. ACM, 2013.

Le, T. C., Antonopoulos, T., Fathololumi, P., Koskinen, E., and Nguyen, T. DynamiTe: Dynamic termination and non-termination proofs. *Proc. ACM Program. Lang.*, 4 (OOPSLA):189:1–189:30, 2020.

Lee, W., Wang, B., and Yi, K. Termination analysis with algorithmic learning. In *CAV*, volume 7358 of *Lecture Notes in Computer Science*, pp. 88–104. Springer, 2012.

Leike, J. and Heizmann, M. Ranking templates for linear loops. In Ábrahám, E. and Havelund, K. (eds.), *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 172–186. Springer, 2014. ISBN 978-3-642-54862-8.

Liu, T. *Learning to Rank for Information Retrieval*. Springer, 2011.

Nori, A. V. and Sharma, R. Termination proofs from tests. In *ESEC/SIGSOFT FSE*, pp. 246–256. ACM, 2013.

Otto, C., Brockschmidt, M., von Essen, C., and Giesl, J. Automated termination analysis of Java bytecode by term rewriting. In *RTA*, volume 6 of *LIPIcs*, pp. 259–276. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2010.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. PyTorch: An imperative style, high-performance deep learning library. In *NeurIPS*, pp. 8024–8035. 2019.

Podelski, A. and Rybalchenko, A. A complete method for the synthesis of linear ranking functions. In *VMCAI*, volume 2937 of *Lecture Notes in Computer Science*, pp. 239–251. Springer, 2004.

Si, X., Dai, H., Raghothaman, M., Naik, M., and Song, L. Learning loop invariants for program verification. In *NeurIPS*, pp. 7762–7773, 2018.

Spoto, F. The Julia static analyzer for Java. In *SAS*, volume 9837 of *Lecture Notes in Computer Science*, pp. 39–57. Springer, 2016.

Urban, C. The abstract domain of segmented ranking functions. In *SAS*, volume 7935 of *Lecture Notes in Computer Science*, pp. 43–62. Springer, 2013.

Urban, C., Gurfinkel, A., and Kahsai, T. Synthesizing ranking functions from bits and pieces. In *TACAS*, volume 9636 of *Lecture Notes in Computer Science*, pp. 54–70. Springer, 2016.