CLN2INV: LEARNING LOOP INVARIANTS WITH CONTINUOUS LOGIC NETWORKS

Anonmous authors
Paper under double-blind review

ABSTRACT

Program verification offers a framework for ensuring program correctness and therefore systematically eliminating different classes of bugs. Inferring loop invariants is one of the main challenges behind automated verification of real-world programs which often contain many loops. In this paper, we present Continuous Logic Network (CLN), a novel neural architecture for automatically learning loop invariants directly from program execution traces. Unlike existing neural networks, CLNs can learn precise and explicit representations of formulas in Satisfiability Modulo Theories (SMT) for loop invariants from program execution traces. We develop a new sound and complete semantic mapping for assigning SMT formulas to continuous truth values that allows CLNs to be trained efficiently. We use CLNs to implement a new inference system for loop invariants, CLN2INV, that significantly outperforms existing approaches on the popular Code2Inv dataset. CLN2INV is the first tool to solve all 124 theoretically solvable problems in the Code2Inv dataset. Moreover, CLN2INV takes only 1.1 second on average for each problem, which is $40\times$ faster than existing approaches. We further demonstrate that CLN2INV can even learn 12 significantly more complex loop invariants than the ones required for the Code2Inv dataset.

1 INTRODUCTION

Program verification offers a principled approach for systematically eliminating different classes of bugs and proving the correctness of programs. However, as programs have become increasingly complex, real-world program verification often requires prohibitively expensive manual effort (Wilcox et al., 2015; Gu et al., 2016; Chajed et al., 2019). Recent efforts have focused on automating the program verification process, but automated verification of general programs with unbounded loops remains an open problem (Nelson et al., 2017; 2019).

Verifying programs with loops requires determining loop invariants, which captures the effect of the loop on the program state irrespective of the actual number of loop iterations. Automatically inferring correct loop invariants is a challenging problem that is undecidable in general and difficult to solve in practice (Blass & Gurevich, 2001; Puria et al., 2014). Existing approaches use stochastic search (Sharma & Aiken, 2016), heurstics-based search (Galeotti et al., 2015), PAC learning based on counter examples (Padhi & Millstein, 2017), or reinforcement learning (Si et al., 2018). However, these approaches often struggle to learn complex, real-world loop invariants.

In this paper, we introduce a new approach to learning loop invariants by modeling the loop behavior from program execution traces using a new type of neural architecture. We note that inferring loop invariants can be posed as learning formulas in Satisfiability Modulo Theories (SMT) (Biere et al., 2009) over program variables collected from program execution traces (Nguyen et al., 2017). In principle, Neural networks seem well suited to this task because they can act as universal function approximators and have been successfully applied in various domains that require modeling of arbitrary functions (Hornik et al., 1989; Goodfellow et al., 2016). However, loop invariants must be represented as explicit SMT formulas to be usable for program verification. Unfortunately, existing methods for extracting logical rules from general neural architectures lack sufficient precision (Auggsta & Kathirvalavakumar, 2012), while inductive logic learning lacks sufficient expressiveness for use in verification (Evans & Grefenstette, 2018).
We address this issue by developing a novel neural architecture, Continuous Logic Network (CLN), which is able to efficiently learn explicit and precise representations of SMT formulas by using continuous truth values. Unlike existing neural architectures, CLNs can represent a learned SMT formula explicitly in its structure and thus allow us to precisely extract the exact formula from a trained model.

In order to train CLNs, we introduce a new semantic mapping for SMT formulas to continuous truth values. Our semantic mapping builds on BL, or basic fuzzy logic (Hájek, 2013), to support general SMT formulas in a continuous logic setting. We further prove that our semantic model is sound (i.e., truth assignments for the formulas are consistent with their discrete counterparts) and complete (i.e., all formulas can be represented) with regard to the discrete SMT formula space. These properties allow CLNs to represent any quantifier-free SMT formula operating on mixed integer-real arithmetic as an end-to-end differentiable series of operations.

We use CLNs to implement a new inference system for loop invariants, CLN2INV, that significantly outperforms state-of-the-art tools on the Code2Inv dataset by solving all 124 theoretically solvable problems in the dataset. This is 20 problems more than LoopInvGen, the winner of the SyGus 2018 competition loop invariant track (Padhi & Millstein, 2017). Moreover, CLN2INV finds invariants for each program in 1.1 second on average, more than 40 times faster than LoopInvGen. We also demonstrate that CLN2INV is able to learn complex, real-world loop invariants with combinations of conjunctions and disjunctions of multivariable constraints.

Our main contributions are:

- We introduce a new semantic mapping for assigning continuous truth values to SMT formulas that is theoretically grounded and enables learning formulas through backpropagation. We further prove that our semantic model is sound and complete.
- We develop a novel neural architecture, Continuous Logic Networks (CLNs), that to the best of our knowledge is the first to efficiently learn precise and explicit SMT formulas by construction.
- We use CLNs to implement a new loop invariant inference system, CLN2INV, that is the first to solve all 124 theoretically solvable problems in the Code2Inv dataset, 20 more than the existing methods. CLN2INV is able to find invariants for each problem in 1.1 second on average, 40× faster than existing systems.
- We further show CLN2INV is able to learn 12 more complex loop invariants than the ones present in the Code2Inv dataset with combinations of multivariable constraints.

Related Work. Traditionally, loop invariant learning relies on stochastic or heuristics-guided search (Sharma & Aiken, 2016; Galeotti et al., 2015). Other approaches like NumInv analyze traces and discover conjunctions of equalities by solving a system of linear equations (Sharma et al., 2013; Nguyen et al., 2017). LoopInvGen uses PAC learning of CNF using counter-examples (Padhi et al., 2016; Padhi & Millstein, 2017). By contrast, Code2Inv learns to guess loop invariants using reinforcement learning with recurrent and graph neural networks (Si et al., 2018). However, these approaches struggle to learn complex invariants. Unlike these works, CLN2INV efficiently learns complex invariants directly from execution traces.

There is an extensive work on PAC learning of boolean formulas, but learning precise formulas require a prohibitively large number of samples (Kearns et al., 1994). Several recent works use differentiable logic to learn boolean logic formulas from noisy data (Kimmig et al., 2012; Evans & Grefenstette, 2018; Payani & Fekri, 2019) or improving adversarial robustness by applying logical rules to training (Fischer et al., 2019). By contrast, our work learns precise SMT formulas directly by construction, allowing us to learn richer predicates with compact representation in a noiseless setting.

A variety of numerical relaxations have been applied to SAT and SMT solving. Application-specific approximations using methods such as interval overapproximation and slack variables have been developed for different classes of SMT (Eggers et al., 2008; Nuzzo et al., 2010). More recent work has applied recurrent and graph neural networks to Circuit SAT problems and unsat core detection (Amizadeh et al., 2019; Selsam et al., 2019; Selsam & Bjørner, 2019). FastSMT uses embeddings from natural language processing like skip-gram and bag-of-words to represent formulas for search strategy optimization (Balanovic et al., 2018). Unlike these approaches, we relax the SMT semantics directly to generate a differentiable representation of SMT.
2 Background

In this section, we introduce the problem of inferring loop invariants and provide a brief overview of Satisfiability Modulo Theories (SMT), which are used to represent loop invariants. We provide background into fuzzy logic, which we extend with our new continuous semantic mapping for SMT.

2.1 Loop Invariants

Loop invariants capture loop behavior irrespective of number of iterations, which is crucial for verifying programs with loops. Given a loop, while LC, a pre-condition P, and a post-condition Q, the verification task involves finding a loop invariant I that can be concluded from the pre-condition and implies the post-condition. Formally, it must satisfy the following three conditions, in which the second is a Hoare triple describing the loop:

\[ P \implies I \quad \{I \land LC\} \ C \quad \{I\} \quad \neg LC \land I \implies Q \]

Example of Loop Invariant. Consider the example loop in Fig. 1. For a loop invariant to be usable, it must be valid for the pre-condition \( t = 10 \land u = 0 \), the recursion step when \( t \neq 0 \), and the post condition \( u = 20 \) when the loop condition is no longer satisfied, i.e., \( t = 0 \). The correct and precise invariant \( I \) for the program is \( (2t + u = 20) \).

\[
\begin{align*}
//pre: \ & t=10 \ /\ u=0 \\
while (t \neq 0) \\
\quad t &= t - 1; \\
\quad u &= u + 2; \\
//post: \ & u=20
\end{align*}
\]

(a) Example loop

The desired loop invariant \( I \) for the left program is a boolean function over program variables \( t, u \) such that:

\[
\forall t, u, \left\{ \begin{array}{l}
\text{pre} \quad t = 10 \land u = 0 \implies I(t, u) \\
\text{inv} \quad I(t, u) \land (t \neq 0) \implies I(t - 1, u + 2) \\
\text{post} \quad I(t, u) \land (t = 0) \implies u = 20
\end{array} \right.
\]

(b) The desired and precise loop invariant \( I \) is \( (2t + u = 20) \).

Figure 1: Example Loop Invariant inference problem.

2.2 Satisfiability Modulo Theories

Satisfiability Modulo Theories (SMT) are an extension of Boolean Satisfiability that allow solvers to reason about complex problems efficiently. Loop invariants and other formulas in program verification are usually encoded with quantifier-free SMT. A formula \( F \) in quantifier-free SMT can be inductively defined as below:

\[
F := E_1 \lor E_2 \mid \neg F \mid F_1 \lor F_2 \mid F_1 \land F_2 \mid \exists \in \{=, \neq, <, >, \leq, \geq\}
\]

where \( E_1 \) and \( E_2 \) are expressions of terms. The loop invariant \((2t + u = 20)\) in Fig. 1 is an SMT formula. Nonlinear arithmetic theories admit higher-order terms such as \( t^2 \) and \( t \times u \), allowing them to express more complex constraints. For example, \((- (2 \geq t^2))\) is an SMT formula that is true when the value of the high-order term \( t^2 \) is larger than 2.

2.3 Basic Fuzzy Logic (BL)

Basic fuzzy logic (BL) is a class of logic that uses continuous truth values in the range \([0, 1]\) and is differentiable almost everywhere\(^1\) [Hájek 2013]. BL defines logical conjunction with functions called \( t \)-norms, which must satisfy specific conditions to ensure that the behavior of the logic is consistent with boolean First Order Logic. Formally, a \( t \)-norm (denoted \( \otimes \)) in BL is a binary operator over truth values in the interval \([0, 1]\) satisfying the following conditions: consistency \((1 \otimes x = x \land 0 \otimes x = 0)\), commutativity \((x \otimes y = y \otimes x)\), associativity \((x \otimes (y \otimes z) = (x \otimes y) \otimes z)\), and monotonicity \((x_1 \leq x_2) \implies x_1 \otimes y \leq x_2 \otimes y\). Besides these conditions, BL also requires that \( t \)-norms be continuous. Given a \( t \)-norm \( \otimes \), its associated \( t \)-conorm (denoted \( \oplus \)) is defined with DeMorgan’s law: \( t \oplus u = \neg(t \otimes \neg u)\), which can be considered as logical disjunction. A common \( t \)-norm is the product \( t \)-norm \( x \otimes y = x \cdot y \) with its associated \( t \)-conorm \( x \oplus y = x + y - x \cdot y \).

\(^1\)Almost everywhere indicates the function is differentiable everywhere except for a set of measure 0. For example, a Rectified Linear Unit is differentiable almost everywhere except at zero.
3 CONTINUOUS SATISFIABILITY MODULO THEORIES

We introduce a continuous semantic mapping, $S$, for SMT on BL that is end-to-end differentiable. The mapping $S$ associates SMT formulas with continuous truth values while preserving each formula’s semantics. In this paper, we only consider quantifier-free formulas. This process is analogous to constructing t-norms for BL, where a t-norm operates on continuous logical inputs.

We define three desirable properties for continuous semantic mapping $S$ that will preserve formula semantics while facilitating parameter training with gradient descent:

1. $S(F)$ should be consistent with BL. For any two formulas $F$ and $F'$, where $F(x)$ is satisfied and $F'(x)$ is unsatisfied with an assignment $x$ of formula terms, we should have $S(F'(x)) < S(F(x))$. This will ensure the semantics of SMT formulas are preserved.

2. $S(F)$ should be differentiable almost everywhere. This will facilitate training with gradient descent through backpropagation.

3. $S(F)$ should be increasing everywhere as the terms in the formula approach constraint satisfaction, and decreasing everywhere as the terms in the formula approach constraint violation. This ensures there is always a nonzero gradient for training.

Continuous semantic mapping. We first define the mapping for “$>$” (greater-than) and “$\geq$” (greater-than-or-equal-to) as well as adopting definitions for “$=$”, “$\wedge$”, and “$\lor$” from BL. All other operators can be derived from these. For example, “$<$” (less-than-or-equal-to) is derived using “$\geq$” and “$=$”, while “$=$” (equality) is then defined as the conjunction of formulas using “$<$” and “$\geq$.”

Given constants $B > 0$ and $\epsilon > 0$, we first define the the mapping $S$ on “$>$” and “$\geq$” using shifted and scaled sigmoid functions:

$$S(t > u) \triangleq \frac{1}{1 + e^{-B(t-u-\epsilon)}} \quad S(t \geq u) \triangleq \frac{1}{1 + e^{-B(t-u+\epsilon)}}$$

Illustrations of these functions are given in Appendix A. The validity of our semantic mapping lie in the following facts, which can be proven with basic algebra.

$$\lim_{\epsilon \to 0^+} \frac{1}{1 + e^{-B(t-u-\epsilon)}} = \begin{cases} 1 & t > u \\ 0 & t \leq u \end{cases} \quad \lim_{B \to +\infty} \frac{1}{1 + e^{-B(t-u+\epsilon)}} = \begin{cases} 1 & t \geq u \\ 0 & t < u \end{cases}$$

When $\epsilon$ goes to zero and $B \to \infty$, our continuous mapping of “$>$” and “$\geq$” will preserve their original semantics. Under these conditions, our mapping satisfies all three desirable properties. In practice, for small $\epsilon$ and large $B$, the properties are also satisfied if $|t - u| > \epsilon$.

Next we define the mapping $S$ for boolean operators “$\wedge$”, “$\lor$” and “$\neg$” using BL. Given a specific t-norm $\otimes$ and its corresponding t-conorm $\oplus$, it is straightforward to define mappings of “$\wedge$”, “$\lor$” and “$\neg$”:

$$S(F_1 \wedge F_2) \triangleq S(F_1) \otimes S(F_2) \quad S(F_1 \lor F_2) \triangleq S(F_1) \oplus S(F_2) \quad S(\neg F) \triangleq 1 - S(F)$$

Based on the above definitions, the mapping for other operators can be derived as follows:

$$S(t < u) = S((t \geq u)) = \frac{1}{1 + e^{B(t-u+\epsilon)}} \quad S(t \leq u) = S((t > u)) = \frac{1}{1 + e^{B(t-u-\epsilon)}}$$

$$S(t = u) = S((t \geq u) \land (t \leq u)) = \frac{1}{1 + e^{B(t-u+\epsilon)}} \otimes \frac{1}{1 + e^{B(t-u-\epsilon)}}$$

The mapping $S$ on “$=$” is valid since the following limit holds (see Appendix B for the proof).

$$\lim_{B \to +\infty} \frac{1}{1 + e^{B(t-u+\epsilon)}} = \begin{cases} 1 & t = u \\ 0 & t \neq u \end{cases}$$

The mapping for other operators shares similar behavior in the limit, and also fulfill our desired properties under the same conditions.

Using our semantic mapping $S$, most of the standard operations of integer and real arithmetic, including addition, subtraction, multiplication, division, and exponentiation, can be used normally and mapped to continuous truth values while keeping the entire formula differentiable. Moreover, any expression in SMT that has an integer or real-valued result can be mapped to continuous logical values via these formulas, although end-to-end differentiability may not be maintained in cases where specific operations are nondifferentiable.
4 CONTINUOUS LOGIC NETWORKS

In this section, we describe the construction of Continuous Logic Networks (CLNs) based on our continuous semantic mapping for SMT on BL.

**CLN Construction.** CLNs use our semantic mapping to provide a general neural architecture for learning SMT formulas. In a CLN, the learnable coefficients and smoothing parameters correspond to the learnable parameters in a standard feedforward network, and the continuous predicates, t-norms, and t-conorms operate as activation functions like ReLUs in a standard network. In this paper, we focus on shallow networks to address the loop invariant inference problem, but we envision deeper general purpose CLNs that can learn arbitrary SMT formulas. When constructing a CLN, we work from an SMT Formula Template, in which every value is marked as either an input term, a constant, or a learnable parameter. Given an SMT Formula Template, we dynamically construct a CLN as a computational graph. Figure 2 shows a simple formula template and the constructed CLN. We denote the CLN model constructed from the formula template \( S(F) \) as \( M_F \).

**CLN Training.** Once the CLN has been constructed based on a formula template, it is trained with the following optimization. Given a CLN model \( M \) constructed from an SMT template with learnable parameters \( W \), and a set \( X \) of valid assignments for the terms in the SMT template, the expected value of the CLN is maximized by minimizing a loss function \( L \) that penalizes model outputs that are less than one. A minimum scaling factor \( \beta \) is selected, and a hinge loss is applied to the scaling factors \( (B) \) to force the differentiable predicates to approach sharp cutoffs. The offset \( \epsilon \) is also regularized to ensure precision. The overall optimization is formulated as:

\[
\max_{\{W,B,\epsilon\}} \mathbb{E}[M(X; W, B, \epsilon)] = \min_{\{W,B,\epsilon\}} \sum_{x \in X} L(M(x; W, B, \epsilon)) + \lambda \sum_{B \in B} L_{hinge}(\beta, B) + \gamma ||\epsilon||_2
\]

where \( \lambda \) and \( \gamma \) are hyperparameters respectively governing the weight assigned to the scaling factor and offset regularization. \( L_{hinge}(\beta, B) \) is defined as \( \max(0, \beta - B) \), and \( L \) is any loss function strictly decreasing in domain \([0, 1]\). Given a CLN that has been trained to a loss approaching 0 on a given set of valid assignments, we show that the resulting continuous SMT formula learned by the CLN is consistent with an equivalent discrete SMT formula. In particular, we prove that such a formula is sound, (i.e., a CLN will learn a correct SMT formula with respect to the training data), and that our continuous mapping is complete, (i.e., CLNs can represent any SMT formula that can be represented in discrete logic). We further prove that CLNs are guaranteed to converge to a globally optimal solution for formulas, which can be expressed as the conjunction of linear equalities. We provide formal definitions and proofs for soundness and completeness in Appendix [C] and optimality in Appendix [D].

5 LOOP INVARIANT LEARNING

We use CLNs to implement a new inference system for loop invariants, CLN2INV, which learns invariants directly from execution traces. CLN2INV follows the same overall process as other loop invariant inference systems such as LoopInvGen and Code2Inv – it iterates through likely candidate invariants and checks its validity with an SMT solver. The key difference between our method and other systems is that it learns a loop invariant formula directly from trace data. Figure 2 provides an overview of the architecture.

**Preprocessing.** We first perform static analysis and instrument the program to prepare for training data generation. In addition to collecting the given precondition and post-condition, the static analysis extracts all constants in the program, along with the loop termination condition. We then instrument the program to record all program variables before each loop execution and after the loop termination. We also restrict the loop to terminate after a set number of iterations to prevent loops running indefinitely (for experiments in this paper, we set the max loop iterations to 50). We also strengthen the precondition to ensure loop execution (see Appendix [E]).

**Training Data Generation.** We generate training data by running the program repeatedly on a set of randomly initialized inputs that satisfy the preconditions. Unconstrained variables are initialized from a uniform distribution centered on 0 with width \( r \), where \( r \) is a hyper-parameter of the sampling process. Variables with either upper or lower bound precondition constraints are initialized from
Under review as a conference paper at ICLR 2020

Figure 2: System architecture and CLN construction from SMT templates.

a uniform distribution adjacent to their constraints with width $r$, while variables with both upper and lower bounds in the precondition are sampled uniformly within their bounds. For all of our experiments in this paper, we set $r$ to 10. When the number of uninitialized variables is small (i.e., less than 3), we perform this sampling exhaustively. An example of training data generation is provided in Appendix F.

**Template Generation.** We generate templates in three stages with increasing expressiveness:

1. We first generate templates directly from the pre-condition and post-condition.
2. We next extract the individual clauses from the pre- and post-condition, as well as the loop condition, and generate templates from conjunctions and disjunctions of each possible pair of clauses.
3. We finally generate more generic templates of increasing complexity with a combination of one or more equality constraints on all variables combined with conjunctions of inequality constraints, which are based on the loop condition and individual variables.

We describe the template generation in detail in Appendix F. To detect when higher order terms may be present in the invariant, we perform a log-log linear regression on each variable relative to the loop iteration, similarly to Sharma et al. (2013). If the loop contains one or more variables that grow superlinearly relative to the loop iteration, we add higher order polynomial terms to the equality constraints in the template, up to the highest degree detected among the loop variables.

**CLN Construction and Training.** Once a template formula has been generated, a CLN is constructed from the template using the formulation in §4. As an optimization, we represent equality constraints as Gaussian-like functions that retain a global maximum when the constraint is satisfied as discussed in Appendix G. We then train the model using the collected execution traces.

**Invariant Checking.** Invariant checking is performed using SMT solvers such as Z3 (De Moura & Bjørner, 2008). After the CLN for a formula template has been trained, the SMT formula for the loop invariant is recovered by normalizing the learned parameters. The invariant is checked against the pre, post, and recursion conditions as described in §2.1. If the correct invariant is not found, we return to the template generation phase to continue the search with a more expressive template.

6 E X P E R I M E N T S

We compare the performance of CLN2INV with two existing methods and demonstrate the efficacy of the method on several more difficult problems. Finally, we conduct two ablation studies to justify our design choices.

**Test Environment.** All experiments are performed on an Ubuntu 18.04 server with an Intel Xeon E5-2623 v4 2.60GHz CPU, 256Gb of memory, and an Nvidia GTX 1080Ti GPU.

**System Configuration.** We implement CLNs in PyTorch and use the Adam optimizer for training with learning rate 0.01 (Paszke et al., 2017; Kingma & Ba, 2014). Because the performance of CLN is dependent on weight initialization, the CLN training randomly restart if the model does not reach termination within 2,000 epochs. Learnable parameters are initialized from a uniform distribution in the range [-1, 1], which we found works well in practice.
**Test Dataset.** We use the same benchmark used in the evaluation of Code2Inv. We have removed nine invalid programs from Code2Inv’s benchmark and test on the remaining 124. The removed programs are invalid because there are inputs which satisfy the precondition but result in a violation of the post-condition. See Appendix [H] for details on the removed problems. The benchmark consists of loops expressed as C code and corresponding SMT files. Each loop can have nested if-then-else blocks (without nested loops). Programs in the benchmark may also have uninterpreted functions with boolean return values (emulating external function calls) in branches or loop termination conditions.

### 6.1 Comparison to Existing Solvers

![Figure 3: Performance evaluation.](image)

**Performance Comparison.** We compare CLN2INV to two state-of-the-art methods: Code2Inv (based on neural code representation and reinforcement learning) and LoopInvGen (PAC learning over synthesized CNF formulas) [Si et al., 2018; Padhi & Millstein, 2017]. We limit each method to one hour per problem in the same format as the SyGuS Competition [Alur et al., 2019]. CLN2INV is able to solve all 124 problems in the benchmark. LoopInvGen solves 104 problems while Code2inv solves 90.

Figure 3a shows the measured runtime on each evaluated system. CLN2INV solves problems in 1.1 second on average, which is over $40 \times$ faster than LoopInvGen, the second fastest system in the evaluation. It spends the most time on solver calls (0.6s avg.) and CLN training (0.5s avg.), with negligible time spent on preprocessing, data generation, and template generation on each problem (<20ms ea.). We provide a breakdown of runtimes in Appendix [I]. In general, CLN2INV has similar performance to LoopInvGen on simple problems but is able to scale efficiently to complex problems. Figure 3b shows the number of Z3 calls made by each method. For almost all problems, CLN2INV requires fewer Z3 calls than the other systems, although for some difficult problems it uses more Z3 calls than Code2Inv.

Table [1] summarizes results of the performance evaluation. Code2Inv require much more time on average per problem, but minimizes the number of calls made to an SMT solver. In contrast, LoopInvGen is efficient at generating a large volume of guessed candidate invariants, but is much less accurate for each individual invariant. CLN2INV can be seen as balance between the two approaches: it searches over candidate invariants more quickly than Code2Inv, but generates more accurate invariants than LoopInvGen, resulting in lower overall runtime.

### 6.2 More Difficult Loop Invariants

We consider two classes of more difficult loop invariant inference problems that are not present in the Code2Inv dataset. The first require conjunctions and disjunctions of multivariable constraints, and the second require polynomials with many higher order terms. Both of these classes of problems

---

2 The Code2Inv authors originally reported solving 92 problems using the same one hour timeout. We believe that the difference may be caused by changes in the testing environment or randomized model initialization.

3 At most 6171 data points are generated for a given program in the benchmark, and 1041 data points per program on average, using the sampling strategy in §5.
Table 1: Results and summary statistics for performance evaluation.

<table>
<thead>
<tr>
<th>Method</th>
<th>Number Solved</th>
<th>Avg Time (s)</th>
<th>Avg Z3 Calls</th>
<th>Time/Z3 Call (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code2Inv</td>
<td>90</td>
<td>266.71</td>
<td>16.62</td>
<td>50.89</td>
</tr>
<tr>
<td>LoopInvGen</td>
<td>104</td>
<td>45.11</td>
<td>3,605.43</td>
<td>0.08</td>
</tr>
<tr>
<td>CLN2INV</td>
<td>124</td>
<td>1.07</td>
<td>31.77</td>
<td>0.17</td>
</tr>
</tbody>
</table>

are significantly more challenging because they are more complex and cause the space of possible invariants to grow much more quickly.

To evaluate on problems that require invariants with conjunctions and disjunctions of multivariable constraints, we construct 12 additional problems. For these problems, we only consider loops with invariants that cannot be easily inferred with pre- and post-condition based heuristics. Appendix J describes these problems in more detail and provides examples. CLN2INV is able to find correct invariants for all 12 problems in less than 20 seconds, while Code2Inv and LoopInvGen time out after an hour.

To evaluate on problems with higher order polynomial invariants, we test CLN2INV on the power summation problems in the form \( u = \sum_{t=0}^{t_d} t^d \) for a given degree \( d \), which have been used in evaluation for polynomial loop invariant inference (Sharma et al., 2013; Nguyen et al., 2017). We discuss these problems in more detail in Appendix J. CLN2INV can correctly learn the invariant for 1st and 2nd order power summations, but cannot learn correct invariants for 3rd, 4th or 5th order summations, which have many more higher order terms. We do not evaluate the other methods on these problems because they are not configured for nonlinear arithmetic by default.

6.3 Ablation Studies

Effect of CLN Training on Performance. CLN2INV relies on a combination of heuristics using static analysis and learning formulas from execution traces to correctly infer loop invariants. In this ablation we disable model training and limit CLN2INV to static models with no learnable parameters. Static CLN2INV solves 91 problems in the dataset. Figure 4 shows a comparison of full CLN2INV with one limited to static models. CLN2INV’s performance with training disabled shows that a large number of problems in the dataset are relatively simple and can be inferred from basic heuristics. However, for more difficult problems, CLN learning is key to inferring correct invariants.

7 Conclusion

We develop a novel neural architecture that explicitly and precisely learns SMT formulas by construction. We achieve this by introducing a new sound and complete semantic mapping for SMT that enables learning formulas through backpropagation. We use CLNs to implement a loop invariant inference system, CLN2INV, that is the first to solve all theoretically solvable problems in the Code2Inv benchmark and takes only 1.1 second on average. We believe that the CLN architecture will also be beneficial for other domains that require learning SMT formulas.
REFERENCES


A CONTINUOUS PREDICATES

Figure 5 shows examples of shifted sigmoids for $S(>)$, $S(\geq)$, and $S(=)$.

![Plot of $S(x \geq 0)$, $S(x > 0)$ with sigmoid](image)

(a) Plot of $S(x \geq 0)$, $S(x > 0)$ with sigmoid

![Plot of $S(x = 0)$ with product t-norm](image)

(b) Plot of $S(x = 0)$ with product t-norm

Figure 5: Illustration of the mapping $S$ on $\succ, \geq, =$ when $B = 20$ and $\epsilon = 0.2$

B PROOF OF LIMIT OF $S(t = u)$

$$\lim_{B \to +\infty, t \to +\infty} S(t = u) = \lim_{B \to +\infty, t \to +\infty} \frac{1}{1 + e^{-B(t-u+\epsilon)}} = \begin{cases} 1 & t = u \\ 0 & t \neq u \end{cases}$$

Proof. Let $f(t, u; B, \epsilon) = \frac{1}{1 + e^{-B(t-u+\epsilon)}}$ and $g(t, u; B, \epsilon) = \frac{1}{1 + e^{-B(t-u-\epsilon)}}$. Then what we want to prove becomes

$$\lim_{B \to +\infty, t \to +\infty} (f(t, u; B, \epsilon) \otimes g(t, u; B, \epsilon)) = \begin{cases} 1 & t = u \\ 0 & t \neq u \end{cases}$$

Because all $f, g, \otimes$ are continuous in their domain, we have

$$\lim_{B \to +\infty, t \to +\infty} (f(t, u; B, \epsilon) \otimes g(t, u; B, \epsilon)) = \left( \lim_{B \to +\infty, t \to +\infty} f(t, u; B, \epsilon) \right) \otimes \left( \lim_{B \to +\infty, t \to +\infty} g(t, u; B, \epsilon) \right)$$

Using basic algebra, we get

$$\lim_{B \to +\infty, t \to +\infty} f(t, u; B, \epsilon) = \begin{cases} 1 & t \geq u \\ 0 & t < u \end{cases} \quad \lim_{B \to +\infty, t \to +\infty} g(t, u; B, \epsilon) = \begin{cases} 1 & t \leq u \\ 0 & t > u \end{cases}$$

Combing these results, we have

$$\left( \lim_{B \to +\infty, t \to +\infty} f(t, u; B, \epsilon) \right) \otimes \left( \lim_{B \to +\infty, t \to +\infty} g(t, u; B, \epsilon) \right) = \begin{cases} 0 \otimes 1 & t < u \\ 1 \otimes 1 & t = u \\ 1 \otimes 0 & t > u \end{cases}$$

For any t-norm, we have $0 \otimes 1 = 0$, $1 \otimes 1 = 1$, and $1 \otimes 0 = 0$. Put it altogether, we have

$$\lim_{B \to +\infty, t \to +\infty} (f(t, u; B, \epsilon) \otimes g(t, u; B, \epsilon)) = \begin{cases} 1 & t = u \\ 0 & t \neq u \end{cases}$$

which concludes the proof. \qed


C \hspace{1em} \textbf{Soundness and Completeness of CLNs}

\textbf{Soundness.} Given the SMT formula $F$, the CLN model $M_F$ constructed from $S(F)$ always preserves the truth value of $F$. It indicates that given a valid assignment to the terms $x$ in $F$, $F(x) = \text{True} \iff M_F(x) = 1$ and $F(x) = \text{False} \iff M_F(x) = 0$.

\textbf{Completeness.} For any SMT formula $F$, a CLN model $M$ can be constructed representing that formula. In other words, CLNs can express all SMT formulas on integers and reals.

We formally state these properties in Theorem 1. Before that we need to define a property for t-norms.

\textbf{Property 1.} $\forall t, (t > 0)$ and $(u > 0)$ implies $(t \otimes u > 0)$.

The product t-norm and Godel t-norm have this property, while the Lukasiewicz t-norm does not.

\textbf{Theorem 1.} For any quantifier-free linear SMT formula $F$, there exists CLN model $M$, such that

$\forall x \hspace{1em} F(x) = \text{True} \iff \lim_{x \rightarrow 0^+} M(x; B, \epsilon) = 1$ \hspace{1em} (2)

$\forall x \hspace{1em} F(x) = \text{False} \iff \lim_{x \rightarrow 0^+} M(x; B, \epsilon) = 0$ \hspace{1em} (3)

as long as the t-norm used in building $M$ satisfies Property 1.

\textit{Proof.} For convenience of the proof, we first remove all $\land, \lor, \neq$ and $\neq$ in $F$, by transforming $t < u$ into $\neg(t \geq u)$, $t \leq u$ into $\neg(t > u)$, $t = u$ into $(t \geq u) \land \neg(t > u)$, and $t \neq u$ into $(t > u) \lor \neg(t \geq u)$. Now the only operators that $F$ may contain are $\neg, \lor, \land \land \neg$. We prove Theorem 1 by induction on the constructor of formula $F$. In the following proof, we construct model $M$ given $F$ and show that it satisfied Eq.(1)(2). We leave the proof for why $M$ also satisfied Eq.(3) to readers.

\textbf{Atomic Case.} When $F$ is an atomic clause, then $F$ will be in the form of $x \cdot W + b > 0$ or $x \cdot W + b \geq 0$. For the first case, we construct a linear layer with weight $W$ and bias $b$ followed by a sigmoid function scaled with factor $B$ and right-shifted with distance $\epsilon$. For the second case, we construct the same linear layer followed by a sigmoid function scaled with factor $B$ and left-shifted with distance $\epsilon$. Simply evaluating the limits for each we arrive at

$\forall x \hspace{1em} F(x) = \text{True} \iff \lim_{x \rightarrow 0^+} M(x; B, \epsilon) = 1$

And from the definition of sigmoid function we know $0 \leq M(x; B, \epsilon) \leq 1$.

\textbf{Negation Case.} If $F = \neg F'$, from the induction hypothesis, $F'$ can be represented by models $M'$ satisfying Eq.(1)(2)(3). Let $p'$ output node of $M'$. We add a final output node $p = 1 - p'$. So $M(x; B, \epsilon) = 1 - M'(x; B, \epsilon)$. Using the induction hypothesis $0 \leq M'(x; B, \epsilon) \leq 1$, we conclude Eq.(1) $0 \leq M(x; B, \epsilon) \leq 1$.

Now we prove the $\implies$ side of Eq.(2). If $F(x) = \text{True}$, then $F'(x) = \text{False}$. From the induction hypothesis, we know $\lim_{x \rightarrow 0^+} M'(x; B, \epsilon) = 0$. So

$\lim_{x \rightarrow 0^+} M(x; B, \epsilon) = \lim_{x \rightarrow 0^+} 1 - M'(x; B, \epsilon) = 1 - 0 = 1$

Next we prove the $\iff$ side. If $\lim_{x \rightarrow 0^+} M(x; B, \epsilon) = 1$, we have

$\lim_{x \rightarrow 0^+} M'(x; B, \epsilon) = \lim_{x \rightarrow 0^+} 1 - M(x; B, \epsilon) = 1 - 1 = 0$

From the induction hypothesis we know that $F'(x) = \text{False}$. So $F(x) = \neg F'(x) = \text{True}$.
Conjunction Case. If $F = F_1 \land F_2$, from the induction hypothesis, $F_1$ and $F_2$ can be represented by models $M_1$ and $M_2$, such that both $(F_1, M_1)$ and $(F_2, M_2)$ satisfy Eq.(1)(2)(3). Let $p_1$ and $p_2$ be the output nodes of $M_1$ and $M_2$. We add a final output node $p = p_1 \otimes p_2$. So $M(x; B, \epsilon) = M_1(x; B, \epsilon) \otimes M_2(x; B, \epsilon)$. Since $(\otimes)$ is continuous and so are $M_1(x; B, \epsilon)$ and $M_2(x; B, \epsilon)$, we know their composition $M(x; B, \epsilon)$ is also continuous. (Readers may wonder why $M_1(x; B, \epsilon)$ is continuous. Actually the continuity of $M(x; B, \epsilon)$ should be proved inductively like this proof itself, and we omit it for brevity.) From the definition of $(\otimes)$, we have $M(1) \leq M(x; B, \epsilon) \leq 1$.

Now we prove the $\implies$ side of Eq.(2). For any $x$, if $F(x) = True$ which means both $F_1(x) = True$ and $F_2(x) = True$, from the induction hypothesis we know that $\lim_{B, \epsilon \to \infty} M_1(x; B, \epsilon) = 1$ and $\lim_{B, \epsilon \to \infty} M_2(x; B, \epsilon) = 1$. Then

$$\lim_{B, \epsilon \to \infty} M(x; B, \epsilon) = \lim_{B, \epsilon \to \infty} M_1(x; B, \epsilon) \otimes M_2(x; B, \epsilon) = 1 \otimes 1 = 1$$

Then we prove the $\impliedby$ side. From the induction hypothesis we know that $M_1(x; B, \epsilon) \leq 1$ and $M_2(x; B, \epsilon) \leq 1$. From the non-decreasing property of t-norms (see §2.3), we have

$$M_1(x; B, \epsilon) \otimes M_2(x; B, \epsilon) \leq M_1(x; B, \epsilon) \otimes 1$$

Then from the consistency property and the commutative property, we have

$$M_1(x; B, \epsilon) \otimes 1 = M_1(x; B, \epsilon)$$

Put them altogether we get

$$M(x; B, \epsilon) \leq M_1(x; B, \epsilon) \leq 1$$

Because we know $\lim_{B, \epsilon \to \infty} M(x; B, \epsilon) = 1$, according to the squeeze theorem in calculus, we get

$$\lim_{B, \epsilon \to \infty} M_1(x; B, \epsilon) = 1$$

From the induction hypothesis, we know that $F_1(x) = True$. We can prove $F_2(x) = True$ in the same manner. Finally we have $F(x) = F_1(x) \land F_2(x) = True$.

Disjunction Case. For the case $F = F_1 \lor F_2$, we construct $M$ from $M_1$ and $M_2$ as we did in the conjunctive case. This time we let the final output node be $p = p_1 \oplus p_2$. From the continuity of $(\oplus)$ and the definition of $(\oplus) (t \oplus u = 1 - (1 - t) \otimes (1 - u))$, $(\oplus)$ is also continuous. We conclude $M(x; B, \epsilon)$ is also continuous and $0 \leq M(x; B, \epsilon) \leq 1$ by the same argument as $F = F_1 \land F_2$.

Now we prove the “$\implies$” side of Eq.(2). For any assignment $x$, if $F(x) = True$ which means $F_1(x) = True$ or $F_2(x) = True$. Without loss of generality, we assume $F_1(x) = True$. From the induction hypothesis, we know $\lim_{B, \epsilon \to \infty} M_1(x; B, \epsilon) = 1$.

For any $(\oplus)$ and any $0 \leq t, t' \leq 1$, if $t \leq t'$, then

$$t \oplus u = 1 - (1 - t) \otimes (1 - u) \leq 1 - (1 - t') \otimes (1 - u) = t' \oplus u$$

Using this property and the induction hypothesis $M_2(x; B, \epsilon) \geq 0$, we have

$$M_1(x; B, \epsilon) \oplus 0 \leq M_1(x; B, \epsilon) \oplus M_2(x; B, \epsilon) = M(x; B, \epsilon)$$

From the induction hypothesis we also have $M_1(x; B, \epsilon) \leq 1$. Using the definition of $(\oplus)$ and the consistency of $(\oplus) (0 \otimes x = 0)$, we get $M_1(x; B, \epsilon) \oplus 0 = M_1(x; B, \epsilon)$. Put them altogether we get

$$M_1(x; B, \epsilon) \leq M(x; B, \epsilon) \leq 1$$

Because we know $\lim_{B, \epsilon \to \infty} M_1(x; B, \epsilon) = 1$, according to the squeeze theorem in calculus, we get

$$\lim_{B, \epsilon \to \infty} M(x; B, \epsilon) = 1.$$
This property can be proved by induction like this proof itself, thus omitted for brevity.

Let

\[
    c_1 = \lim_{B \to 0, \epsilon \to \infty} M_1(x; B, \epsilon) \quad \text{and} \quad c_2 = \lim_{B \to 0, \epsilon \to \infty} M_2(x; B, \epsilon)
\]

Then

\[
    \lim_{B \to 0^+, \epsilon \to \infty} M(x; B, \epsilon) = \lim_{B \to 0^+, \epsilon \to \infty} M_1(x; B, \epsilon) \oplus M_2(x; B, \epsilon) = c_1 \oplus c_2 = 1 - (1 - c_1) \otimes (1 - c_2)
\]

Since we have

\[
    \lim_{B \to 0^+, \epsilon \to \infty} M(x; B, \epsilon) = 1,
\]

we get

\[
    (1 - c_1) \otimes (1 - c_2) = 0
\]

Using Property 1 of (\otimes) (defined in §3), we have \( c_1 = 1 \lor c_2 = 1 \). Without loss of generality, we assume \( c_1 = 1 \). From the induction hypothesis, we know that \( F_1(x) = True \). Finally, \( F(x) = F_1(x) \lor F_2(x) = True \).

Careful readers may have found that if we use the continuous mapping function \( S \) in [6], then we have another perspective of the proof above, which can be viewed as two interwoven parts. The first part is that we proved the following lemma.

**Corollary 1.** For any quantifier-free linear SMT formula \( F \),

\[
    \forall x \; B \; \epsilon, \; 0 \leq S(F; B, \epsilon)(x) \leq 1
\]

\[
    \forall x, \; F(x) = True \iff \lim_{B \epsilon \to \infty} S(F; B, \epsilon)(x) = 1
\]

\[
    \forall x, \; F(x) = False \iff \lim_{B \epsilon \to \infty} S(F; B, \epsilon)(x) = 0
\]

Corollary 1 indicates the soundness of \( S \). The second part is that we construct a CLN model given \( S(F) \). In other words, we translate \( S(F) \) into vertices in a computational graph composed of differentiable operations on continuous truth values.

## D Optimality of CLNs

**Optimality.** For a subset of SMT formulas (conjunctions of multiple linear equalities), CLNs are guaranteed to converge at the global minimum. We formally state this in Theorem 2. We first define another property similar to strict monotonicity.

**Property 2.** \( \forall t_1, t_2, t_3, \; (t_1 < t_2) \text{ and } (t_3 > 0) \implies (t_1 \otimes t_3 < t_2 \otimes t_3) \).

**Theorem 2.** For any CLN model \( M_F \) constructed from a formula, \( F \), by the procedure shown in the proof of Theorem 1, if \( F \) is the conjunction of multiple linear equalities then any local minimum of \( M_F \) is the global minimum, as long as the t-norm used in building \( M_F \) satisfies Property 2.

**Proof.** Since \( F \) is the conjunction of linear equalities, it has the form

\[
    F = \bigwedge_{i=1}^{n} \left( \sum_{j=1}^{l_i} w_{ij}t_{ij} = 0 \right)
\]

Here \( W = \{ w_{ij} \} \) are the learnable weights, and \( \{ t_{ij} \} \) are terms (variables). We omit the bias \( b_i \) in the linear equalities, as the bias can always be transformed into a weight by adding a constant of 1 as a term. For convenience, we define \( f(x) = S(x = 0) = \frac{1}{1 + e^{-B(x + \epsilon)}} \otimes \frac{1}{1 + e^{B(x - \epsilon)}} \).

Given an assignment \( x \) of the terms \( \{ t_{ij} \} \), if we construct our CLN model \( M_F \) following the procedure shown in the proof of Theorem 1, the output of the model will be

\[
    M(x; W, B, \epsilon) = \bigotimes_{i=1}^{n} f\left( \sum_{j=1}^{l_i} w_{ij}t_{ij} \right)
\]
When we train our CLN model, we have a collection of \( m \) data points \( \{t_{ij1}, t_{ij2}, \ldots, t_{ijm}\} \), which satisfy formula \( F \). If \( B \) and \( \epsilon \) are fixed (unlearnable), then the loss function will be

\[
L(W) = \sum_{k=1}^{m} \mathcal{L}(M(x; W, B, \epsilon)) = \sum_{k=1}^{m} \mathcal{L}(\bigotimes_{i=1}^{n} f(\sum_{j=1}^{l_i} w_{ij} t_{ijk})) \tag{4}
\]

Suppose \( W^* = \{w_{ij}^*\} \) is a local minima of \( L(W) \). We need to prove \( W^* \) is also the global minima. To prove this, we use the definition of a local minima. That is,

\[
\exists \delta > 0, \forall W, \|W - W^*\| \leq \delta \implies L(W) \geq L(W^*) \tag{5}
\]

For convenience, we denote \( u_{ik} = \sum_{j=1}^{l_i} w_{ij} t_{ijk} \). Then we rewrite Eq.(4) as

\[
L(W) = \sum_{k=1}^{m} \mathcal{L}(\bigotimes_{i=1}^{n} f(u_{ik}))
\]

If we can prove at \( W^* \), \( \forall 1 \leq i \leq n, 1 \leq k \leq m, u_{ik} = 0 \). Then because (i) \( f \) reaches its global maximum at 0, (ii) the t-norm \( \bigotimes \) is monotonically increasing, (iii) \( \mathcal{L} \) is monotonically decreasing, we can conclude that \( W^* \) is the global minima.

Now we prove \( \forall 1 \leq i \leq n, 1 \leq k \leq m, u_{ik} = 0 \). Here we just show the case \( i = 1 \). The proof for \( i > 1 \) can be directly derived using the associativity of \( \bigotimes \).

Let \( \alpha_k = \bigotimes_{i=2}^{n} f(u_{ik}) \). Since \( f(x) > 0 \) for all \( x \in R \), using Property 2 of our t-norm \( \bigotimes \), we know that \( \alpha_k > 0 \). Now the loss function becomes

\[
L(W) = \sum_{k=1}^{m} \mathcal{L}(f(u_{1k}) \otimes \alpha_k)
\]

From Eq.(5), we have

\[
\exists 0 < \delta' < 1, \forall \gamma, |\gamma| \leq \delta' \implies \sum_{k=1}^{m} \mathcal{L}(f(u_{ik}(1 + \gamma)) \otimes \alpha_k) \geq \sum_{k=1}^{m} \mathcal{L}(f(u_{ik}) \otimes \alpha_k) \tag{6}
\]

Because (i) \( f(x) \) is an even function decreasing on \( x > 0 \) (which can be easily proved), (ii) \( \bigotimes \) is monotonically increasing, (iii) \( \mathcal{L} \) is monotonically decreasing, for \(-\delta' < \gamma < 0\), we have

\[
\sum_{k=1}^{m} \mathcal{L}(f(u_{ik}(1 + \gamma)) \otimes \alpha_k) = \sum_{k=1}^{m} \mathcal{L}(f(|u_{ik}(1 + \gamma)|) \otimes \alpha_k) \leq \sum_{k=1}^{m} \mathcal{L}(f(|u_{ik}|) \otimes \alpha_k) \tag{7}
\]

Combing Eq.(6) and Eq.(7), we have

\[
\sum_{k=1}^{m} \mathcal{L}(f(u_{ik}(1 + \gamma)) \otimes \alpha_k) = \sum_{k=1}^{m} \mathcal{L}(f(u_{ik}) \otimes \alpha_k)
\]

Now we look back on Eq.(7). Since (i) \( \mathcal{L} \) is strictly decreasing, (ii) the t-norm we used here has Property 2 (see [4] for definition), (iii) \( \alpha_k > 0 \), the only case when (=) holds is that for all \( 1 \leq k \leq m \), we have \( f(|u_{ik}(1 + \gamma)|) = f(|u_{ik}|) \). Since \( f(x) \) is strictly decreasing for \( x \geq 0 \), we have \( |u_{ik}(1 + \gamma)| = |u_{ik}| \). Finally because \(-1 < -\delta' < 0\), we have \( u_{ik} = 0 \).

E PRECONDITION STRENGTHENING

**Theorem 3.** Given a program \( C \): assert(\( P \)); while (\( LC \)) \{ \( C \) \} assert(\( Q \));
If we can find a loop invariant \( I' \) for program \( C' \): assert(\( P \land LC \)); while (\( LC \)) \{ \( C \) \} assert(\( Q \)) ; and \( P \land \neg LC \implies Q \), then \( I' \lor (P \land \neg LC) \) is a correct loop invariant for program \( C \).
Proof. Since $I'$ is a loop invariant of $C'$, we have

\[(P \land LC) \land LC \implies I' \quad (a) \quad \{I' \land LC\} C\{I'\} \quad (b) \quad I' \land \neg LC \implies Q \quad (c)\]

We want to prove $I' \lor (P \land \neg LC)$ is a valid loop invariant of $C$, which means

\[P \land LC \implies I' \lor (P \land \neg LC) \quad \{(I' \lor (P \land \neg LC)) \land LC\} C\{(I' \lor (P \land \neg LC))\}\]

\[(I' \lor (P \land \neg LC)) \land \neg LC \implies Q\]

We prove the three propositions separately. To prove $P \land LC \implies I' \lor (P \land \neg LC)$, we transform it into a stronger proposition $P \land LC \implies I'$, which directly comes from (a).

For $\{(I' \lor (P \land \neg LC)) \land LC\} C\{(I' \lor (P \land \neg LC))\}$, after simplification it becomes $\{I' \land LC\} C\{I' \lor (P \land \neg LC)\}$, which is a direct corollary of (b).

For $(I' \lor (P \land \neg LC)) \land \neg LC \implies Q$, after simplification it will become two separate propositions, $I' \land \neg LC \implies Q$ and $P \land \neg LC \implies Q$. The former is exactly (c), and the latter is a known condition in the theorem. \(\Box\)

\section{Implementation Details}

**Training Data Generation Example.** Figure 6 provides an example of our training data generation procedure. The uninitialized variable $k$ is sampled according to the precondition $k \leq 8$ within the predefined width $r = 10$. So we end up enumerating $k = 8, 7, \ldots, -2$. For each $k$, the loop is executed repeatedly until termination, thus generating a small set of samples. The final training set is the union of these small sets.

```
//pre: t=10
//pre: u=0
//pre: k<8
//invariant: 2t+u=20
while (t != k){
    t = t - 1;
    u = u + 2;
}
//post: u==20-2k
```

(a) The original loop program. (b) The sampling procedure.

Figure 6: Illustration of how training data is generated. After the sampling procedure in (b) we have a collection of 88 samples which will later be fed to the CLN model.

**Template Generation.** Templates are first generated from the pre- and post-conditions, followed by every pair of clauses extracted from the pre-condition, post-condition, and loop-condition. Generic templates are then constructed consisting of one or more general equality constraints containing all variables conjoined with inequality constraints.

Three iterations of the generic template generation are shown here:

1. $(W_1 X = a_1) \land (x_1 \leq l_1) \land (x_1 \geq l_1) \land \cdots \land (x_n \leq l_n) \land (x_n \geq l_n)$
2. $((W_1 X = a_1) \land (W_2 X = a_2)) \land (x_1 \leq u_1) \land (x_1 \geq l_1) \land \cdots \land (x_n \leq u_n) \land (x_n \geq l_n)$
3. $((W_1 X = a_1) \lor (W_2 X = a_2)) \land (x_1 \leq u_1) \land (x_1 \geq l_1) \land \cdots \land (x_n \leq u_n) \land (x_n \geq l_n)$

Algorithm [I] summarizes the template generation process. In it the following functions are defined:

- **construct_template:** Construct a template given an smt formula.
- **extract_clauses:** Extract individual clauses from smt formulas.
- **estimate_degrees:** Performs log-log linear regression to estimate degree of each variable.
- **polynomial_kernel:** Executes polynomial kernel on variables and data for a given degree.
- **is_single_constraint:** Checks if condition is a single inequality constraint.
- **extract_loop_constraint:** Converts the loop condition to learnable smt template.
Note that templates are generated on demand, so each template is used to infer a possible invariant before the next is generated.

Algorithm 1 Template Generation Algorithm.

```
Input: Pre ← pre-condition
       Post ← post-condition
       LC ← loop-condition
       Max_Template.Len ← max template eq clauses
       vars ← vars in program
       X ← training data

1: construct_template(Post)
2: construct_template(Pre)
3: clauses ← extract_clauses(Pre, Post, LC)
4: for (c₁, c₂) in all_pairs(clauses) do
5:   construct_template(c₁ ∧ c₂)
6:   construct_template(c₁ ∨ c₂)
7: end for
8: max_degree ← estimate_degrees(X)
9: if max_degree > 1 then
10:   vars ← polynomial_kernel(vars, max_degree)
11: end if
12: bound_constraints ← ()
13: for var in vars do
14:   bound_constraints ← bound_constraints ∧ (var ≤ uvar)
15:   bound_constraints ← bound_constraints ∧ (var ≥ lvar)
16: end for
17: if is_single_constraint(LC) then
18:   bound_constraints ← bound_constraints ∧ extract_loop_constraint(LC)
19: end if
20: construct_template((W · vars = b)) ∧ bound_constraints
21: eq_clauses ← [(W · vars = b)]
22: template_len ← 1
23: while template_len ≤ Max_Template.Len do
24:   for eq_clause in eq_clauses do
25:     construct_template((eq_clause ∧ (W · vars = b)) ∧ bound_constraints)
26:     eq_clauses ← [eq_clauses, eq_clause ∧ (W · vars = b)]
27:     construct_template((eq_clause ∨ (W · vars = b)) ∧ bound_constraints)
28:     eq_clauses ← [eq_clauses, eq_clause ∨ (W · vars = b)]
29:   end for
30: template_len ← template_len + 1
31: end while
```

G Properties of Gaussian Function

We use a Gaussian-like function $S(t = u) = \exp(-\frac{(t-u)^2}{2\sigma^2})$ to represent equalities in our experiments. It has the following two properties. First, it preserves the original semantic of $=$ when $\sigma \to 0$, similar to the mapping $S(t = u) = \frac{1}{1+e^{-(t-u)}} \otimes \frac{1}{1+e^{-(t-u)}}$ we defined in §3.

$$
\lim_{\sigma \to 0^+} \exp(-\frac{(t-u)^2}{2\sigma^2}) = \left\{ \begin{array}{ll}
1 & t = u \\
0 & t \neq u
\end{array} \right.
$$

Second, if we view $S(t = u)$ as a function over $t-u$, then it reaches its only local maximum at $t-u=0$, which means the equality is satisfied.
H INVALID PROBLEMS FROM CODE2INV DATASET

Here we provide an example which is Problem 106 in the dataset. The post condition \( a \geq m \) is wrong if we start from \( a = 0 \), \( m = 1 \), and \( k = 0 \).

```c
int k = 0;
int a, m;
assume(a <= m);
while (k < 1) {
  if (m < a) m = a;
  k = k + 1;
}
assert(a >= m);
```

Executing the program with these inputs results in \( a = 0 \), \( m = 1 \), and \( k = 1 \) as the if condition is never satisfied. But clearly, the post condition \( a \geq m \) is violated. Below we tabulate the counterexamples invalidating the nine removed problems from the dataset:

<table>
<thead>
<tr>
<th>Problem</th>
<th>Counterexample starting state</th>
</tr>
</thead>
<tbody>
<tr>
<td>26</td>
<td>( x, n = 0, 0 )</td>
</tr>
<tr>
<td>27</td>
<td>( x, n = 0, 0 )</td>
</tr>
<tr>
<td>31</td>
<td>( x, n = 0, 0 )</td>
</tr>
<tr>
<td>32</td>
<td>( x, n = 0, 0 )</td>
</tr>
<tr>
<td>61</td>
<td>( c, n = 0, 1 )</td>
</tr>
<tr>
<td>62</td>
<td>( c, n = 0, 1 )</td>
</tr>
<tr>
<td>72</td>
<td>( c, y = 0, 128 )</td>
</tr>
<tr>
<td>75</td>
<td>( c, y = 0, 128 )</td>
</tr>
<tr>
<td>106</td>
<td>( a, m, k = 0, 1, 0 )</td>
</tr>
</tbody>
</table>

I RUNTIME BREAKDOWN

In Table 3 we provide a more detailed analysis of how much time is spent on each stage of the invariant inference pipeline in CLN2INV. Measurements are averaged over 5 runs. The system spends most time on solver calls (0.6s avg.) and CLN training (0.5s avg.) with negligible time spent on preprocessing, sampling, and template generation. For most problems in the Code2Inv benchmark, CLN2INV completes CLN training quickly (less than 0.2s) and spends most of its time performing solver checks, but it requires more epochs to train on some complex problems with many variables.

<table>
<thead>
<tr>
<th>Preprocessing</th>
<th>Sampling</th>
<th>Template Gen.</th>
<th>CLN Training</th>
<th>Checking</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avg. Time</td>
<td>5ms</td>
<td>2ms</td>
<td>4ms</td>
<td>0.5s</td>
</tr>
<tr>
<td>Max Time</td>
<td>18ms</td>
<td>12ms</td>
<td>8ms</td>
<td>7.2s</td>
</tr>
</tbody>
</table>

J MORE DIFFICULT LOOP INVARIANTS

Multivariable Conjunction/Disjunction Invariants. In this subsection, we discuss in detail two of the 12 more difficult loop invariant problems we have created. The first problem is shown in Figure 7. The correct loop invariant for the program is \( ((t + u = 0) \lor (t - u = 0)) \land (t \leq 0) \). The plot of the trace in Figure 7B shows that the points lie on one of the two lines expressible as linear equality constraints. These constraints along with the inequality can be learned from the execution trace using CLN2INV in under 20 seconds.

Figure 8 shows the pseudocode for the second program. Here, we use `unknown` to denote an external function call that returns either true or false, which can be considered as sampling from a
Bernoulli distribution with success probability 0.5. Although the branching behavior is not deterministic, we know \((t + u = 0) \land (v + w = 0) \land (u + w \geq 0)\) is a correct invariant as it holds regardless of which branch is taken. Our CLN2INV can learn this invariant within 20 seconds. For both problems in Figure 7 and 8, both Code2inv and LoopInvGen time out after one hour without finding a solution.

\[
\text{//pre: } t=-10 \land u=10 \land v=-10 \land w=10 \\
\text{while } (u + w > 0) \{ \\
\quad \text{if (unknown()) } \\
\quad \quad t++; u--; \\
\quad \text{else } \\
\quad \quad v++; w--; \\
\} \\
\text{//post: } t=w \land u=v
\]

Figure 8: Pseudocode for Problem 2, which involves a conjunction of equalities

**Polynomial Invariants.** Here we provide results and an example of the higher order polynomial problems; more precisely, the power summation problems in the form \(u = \sum_{i=0}^{k} t^i\) for a given degree \(d\). We found that CLN2INV was able to learn invariants for programs with 10 terms within 2nd degree, but struggled with problems with 20 or more terms of 3rd degree or higher. Table 4 summarizes these results.

<table>
<thead>
<tr>
<th>Number Terms</th>
<th>Highest Degree</th>
<th>Solved?</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1</td>
<td>✓</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>✓</td>
</tr>
<tr>
<td>20</td>
<td>3</td>
<td>✗</td>
</tr>
<tr>
<td>35</td>
<td>4</td>
<td>✗</td>
</tr>
<tr>
<td>56</td>
<td>5</td>
<td>✗</td>
</tr>
</tbody>
</table>

Consider the example loop in Figure 9 which computes the sum of the first \(k\) cubes. We know this sum has a closed form solution:

\[
\sum_{i=0}^{k} k^3 = \frac{k^2(k+1)^2}{4} = \frac{k^4 + 2k^3 + k^2}{4}
\]

For this problem, we would hope to extract the invariant:

\((4u = t^4 + 2t^3 + t^2) \land (t \leq k)\)

However, by naively using the polynomial kernel just as methods like NumInv suggest (Nguyen et al., 2017), we will have 35 monomials of degree at most four over three variables as candidate terms \((t^4u, t^2k^2, tuk, ...),\) and the model must learn to ignore all the terms except \(u, t^4, t^2,\) and...
Additionally, by nature of polynomials the highest order term is a good approximation for the whole polynomial. Thus, \( u = t^4 \) is a good approximation based on the data. We observe our model will find the correct coefficient for \( t^4 \) but the accuracy degrades on the lower ordered terms. The difficulty of learning polynomial invariants using CLN is an interesting direction for future studies.

```plaintext
//pre: t = u = 0 \land k >= 0
while (t < k) {
    t++;
    u += t * t * t;
}
//post: 4u == k**2 * (k + 1)**2
```

Figure 9: Pseudocode for Polynomial Invariant Problem