Learning Nonlinear Loop Invariants with Gated Continuous Logic Networks

Jianan Yao*
Columbia University, USA
jianan@cs.columbia.edu

Gabriel Ryan*
Columbia University, USA
gabe@cs.columbia.edu

Justin Wong*
Columbia University, USA
justin.wong@columbia.edu

Suman Jana
Columbia University, USA
suman@cs.columbia.edu

Ronghui Gu
Columbia University, CertiK, USA
rgu@cs.columbia.edu

Abstract

Verifying real-world programs often requires inferring loop invariants with nonlinear constraints. This is especially true in programs that perform many numerical operations, such as control systems for avionics or industrial plants. Recently, data-driven methods for loop invariant inference have shown promise, especially on linear loop invariants. However, applying data-driven inference to nonlinear loop invariants is challenging due to the large numbers of and large magnitudes of high-order terms, the potential for overfitting on a small number of samples, and the large space of possible nonlinear inequality bounds.

In this paper, we introduce a new neural architecture for general SMT learning, the Gated Continuous Logic Network (G-CLN), and apply it to nonlinear loop invariant learning. G-CLNs extend the Continuous Logic Network (CLN) architecture with gating units and dropout, which allow the model to robustly learn general invariants over large numbers of terms. To address overfitting that arises from finite program sampling, we introduce fractional sampling—a sound relaxation of loop semantics to continuous functions that facilitates unbounded sampling on the real domain. We additionally design a new CLN activation function, the Piecewise Biased Quadratic Unit (PBQU), for naturally learning tight inequality bounds.

We incorporate these methods into a nonlinear loop invariant inference system that can learn general nonlinear loop invariants. We evaluate our system on a benchmark of nonlinear loop invariants and show it solves 26 out of 27 problems, 3 more than prior work, with an average runtime of 53.3 seconds. We further demonstrate the generic learning ability of G-CLNs by solving all 124 problems in the linear Code2Inv benchmark. We also perform a quantitative stability evaluation and show G-CLNs have a convergence rate of 97.5% on quadratic problems, a 39.2% improvement over CLN models.

CCS Concepts: • **Software and its engineering → Software verification**; • **Computing methodologies → Neural networks**.

Keywords: Loop Invariant Inference, Program Verification, Continuous Logic Networks

ACM Reference Format:
Jianan Yao, Gabriel Ryan, Justin Wong, Suman Jana, and Ronghui Gu. 2020. Learning Nonlinear Loop Invariants with Gated Continuous Logic Networks. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '20), June 15–20, 2020, London, UK*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3385412.3385986

1 Introduction

Formal verification provides techniques for proving the correctness of programs, thereby eliminating entire classes of critical bugs. While many operations can be verified automatically, verifying programs with loops usually requires inferring a sufficiently strong loop invariant, which is undecidable in general [5, 10, 15]. Invariant inference systems are therefore based on heuristics that work well for loops that appear in practice. Data-driven loop invariant inference is one approach that has shown significant promise, especially for learning linear invariants [30, 35, 40]. Data-driven inference operates by sampling program state across many executions of a program and trying to identify an Satisfiability Modulo Theories (SMT) formula that is satisfied by all the sampled data points.

However, verifying real-world programs often requires loop invariants with nonlinear constraints. This is especially true in programs that perform many numerical operations, such as control systems for avionics or industrial

*Equal contribution

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PLDI '20, June 15–20, 2020, London, UK
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-7613-6/20/06…$15.00
https://doi.org/10.1145/3385412.3385986

plants [6, 20]. Data-driven nonlinear invariant inference is fundamentally difficult because the space of possible nonlinear invariants is large, but sufficient invariants for verification must be inferred from a finite number of samples. In practice, this leads to three distinct challenges when performing nonlinear data-driven invariant inference: (i) *Large search space with high magnitude terms.* Learning nonlinear terms causes the space of possible invariants to grow quickly (i.e. polynomial expansion of terms grows exponentially in the degree of terms). Moreover, large terms such as x^2 or x^y dominate the process and prevent meaningful invariants from being learned. (ii) *Limited samples.* Bounds on the number of loop iterations in programs with integer variables limit the number of possible samples, leading to overfitting when learning nonlinear invariants. (iii) *Distinguishing sufficient inequalities.* For any given finite set of samples, there are potentially infinite valid inequality bounds on the data. However, verification usually requires specific bounds that constrain the loop behavior as tightly as possible.

Figure 1a and 1b illustrate the challenges posed by loops with many higher-order terms as well as nonlinear inequality bounds. The loop in Figure 1a computes a cubic power and requires the invariant $(x = n^3) \wedge (y = 3n^2 + 3n + 1) \wedge (z = 6n + 6)$ to verify its postcondition $(x = a^3)$. To infer this invariant, a typical data-driven inference system must consider 35 possible terms, ranging from n to x^3, only seven of which are contained in the invariant. Moreover, the higher-order terms in the program will dominate any error measure in fitting an invariant, so any data-driven model will tend to only learn the constraint $(x = n^3)$. Figure 1b shows a loop for computing integer square root where the required invariant is $(n \geq a^2)$ to verify its postcondition. However, a data-driven model must identify this invariant from potentially infinite other valid but loosely fit inequality invariants.

Most existing methods for nonlinear loop invariant inference address these challenges by limiting either the structure of invariants they can learn or the complexity of invariants they can scale to. Polynomial equation solving methods such as Numinv and Guess-And-Check are able to learn equality constraints but cannot learn nonlinear inequality invariants [21, 33]. In contrast, template enumeration methods such as PIE can potentially learn arbitrary invariants but struggle to scale to loops with nonlinear invariants because space of possible invariants grows too quickly [26].

In this paper, we introduce an approach that can learn general nonlinear loop invariants. Our approach is based on Continuous Logic Networks (CLNs), a recently proposed neural architecture that can learn SMT formulas directly from program traces [30]. CLNs use a parameterized relaxation that relaxes SMT formulas to differentiable functions. This allows CLNs to learn SMT formulas with gradient descent, but a template that defines the logical structure of the formula has to be manually provided.

We base our approach on three developments that address the challenges inherent in nonlinear loop invariant inference: First, we introduce a new neural architecture, the *Gated Continuous Logic Network* (G-CLN), a more robust CLN architecture that is not dependent on formula templates. Second, we introduce *Fractional Sampling*, a principled program relaxation for dense sampling. Third, we derive the *Piecewise Biased Quadratic Unit* (PBQU), a new CLN activation function for inequality learning. We provide an overview of these methods below.

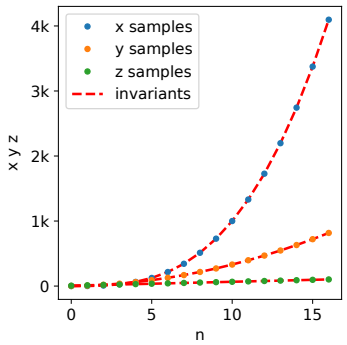
Gated Continuous Logic Networks. G-CLNs improve the CLN architecture by making it more robust and general. Unlike CLNs, G-CLNs are not dependent on formula templates for logical structure. We adapt three different methods from deep learning to make G-CLN training more stable and combat overfitting: gating, dropout, and batch normalization [2, 13, 16, 37]. To force the model to learn a varied combination of constraints, we apply *Term Dropout*, which operates similarly to dropout in feedforward neural networks by zeroing out a random subset of terms in each clause. Gating makes the CLN architecture robust by allowing it to ignore subclauses that cannot learn satisfying coefficients for their inputs, due to poor weight initialization or dropout. To stabilize training in the presence of high magnitude nonlinear terms, we apply normalization to the input and weights similar to batch normalization.

By combining dropout with gating, G-CLNs are able to learn complex constraints for loops with many higher-order terms. For the loop in Figure 1a, the G-CLN will set the n^2 or n^3 terms to zero in several subclauses during dropout, forcing the model to learn a conjunction of all three equality constraints. Clauses that cannot learn a satisfying set of coefficients due to dropout, i.e. a clause with only x and n but no n^3 term, will be ignored by a model with gating.

Fractional Sampling. When the samples from the program trace are insufficient to learn the correct invariant due to bounds on program behavior, we perform a principled relaxation of the program semantics to continuous functions. This allows us to perform Fractional Sampling, which generates samples of the loop behavior at intermediate points between integers. To preserve soundness, we define the relaxation such that operations retain their discrete semantics relative to their inputs but operate on the real domain, and any invariant for the continuous relaxation of the program must be an invariant for the discrete program. This allows us to take potentially unbounded samples even in cases where the program constraints prevent sufficient sampling to learn a correct invariant.

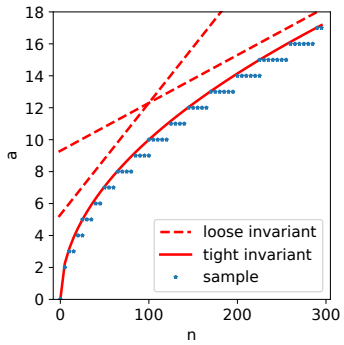
Piecewise Biased Quadratic Units. For inequality learning, we design a PBQU activation, which penalizes loose fits and converges to tight constraints on data. We prove this function will learn a tight bound on at least one point and demonstrate empirically that it learns precise bounds invariant bounds, such as the $(n \geq a^2)$ bound shown in Figure 1b.

```
// pre: (a >= 0)
n=0; x=0;
y=1; z=6;
// compute cube:
while(n != a){
    n += 1;
    x += y;
    y += z;
    z += 6;
}
return x;
// post: x == a^3
```



(a) Loop for computing cubes that requires the invariant $(x = n^3) \wedge (y = 3n^2+3n+1) \wedge (z = 6n+6)$ to infer its postcondition $(x = a^3)$. A data-driven model must simultaneously learn a cubic constraint that changes by 1000s and a linear constraint that increments by 6.

```
// pre: (n >= 0)
a=0; s=1; t=1;
// compute sqrt:
while (s <= n) {
  a += 1;
  t += 2;
  s += t;
}
return a;
//post: a^2 <= n
//and n < (a+1)^2
```



(b) Loop for computing integer approximation to square root. The graph shows three valid inequality invariants, but only the tight quadratic inequality invariant $(n \geq a^2)$ is sufficient to verify that the final value of a is between $\lfloor sqrt(n) \rfloor$ and $\lceil sqrt(n) \rceil$.

Figure 1. Example problems demonstrating the challenges of nonlinear loop invariant learning.

We use G-CLNs with Fractional Sampling and PBQUs to develop a unified approach for general nonlinear loop invariant inference. We evaluate our approach on a set of loop programs with nonlinear invariants, and show it can learn invariants for 26 out of 27 problems, 3 more than prior work, with an average runtime of 53.3 seconds. We also perform a quantitative stability evaluation and show G-CLNs have a convergence rate of 97.5% on quadratic problems, a 39.2% improvement over CLN models. We also test the G-CLN architecture on the linear Code2Inv benchmark [35] and show it can solve all 124 problems.

In summary, this paper makes the following contributions:

- We develop a new general and robust neural architecture, the Gated Continuous Logic Network (G-CLN), to learn general SMT formulas without relying on formula templates for logical structure.
- We introduce Fractional Sampling, a method that facilitates sampling on the real domain by applying a principled relaxation of program loop semantics to continuous functions while preserving soundness of the learned invariants.
- We design PBQUs, a new activation function for learning tight bounds inequalities, and provide convergence guarantees for learning a valid bound.
- We integrate our methods in a general loop invariant inference system and show it solves 26 out 27 problems in a nonlinear loop invariant benchmark, 3 more than prior work. Our system can also infer loop invariants for all 124 problems in the linear Code2Inv benchmark.

The rest of the paper is organized as follows: In §2, we provide background on the loop invariant inference problem, differentiable logic, and the CLN neural architecture for SMT learning. Subsequently, we introduce the high-level workflow of our method in §3. Next, in §4, we formally define the gated CLN construction, relaxation for fractional sampling, and PBQU for inequality learning, and provide soundness guarantees for gating and convergence guarantees for bounds learning. We then provide a detailed description of our approach for nonlinear invariant learning with CLNs in §5. Finally we show evaluation results in §6 and discuss related work in §7 before concluding in §8.

2 Background

In this section, we provide a brief review of the loop invariant inference problem and then define the differentiable logic operators and the Continuous Logic Network architecture used in our approach.

2.1 Loop Invariant Inference

Loop invariants encapsulate properties of the loop which are independent of the iterations and enable verification to be performed over loops. For an invariant to be sufficient for verification, it must simultaneously be weak enough to be derived from the precondition and strong enough to conclude the post-condition. Formally, the loop invariant inference problem is, given a loop "`while(LC) C`," a precondition P, and a post-condition Q, we are asked to find an inductive invariant I that satisfies the following three conditions:

$$P \implies I \qquad \{I \wedge LC\}\, C\, \{I\} \qquad I \wedge \neg LC \implies Q$$

where the inductive condition is defined using a Hoare triple.

Loop invariants can be encoded in SMT, which facilitates efficient checking of the conditions with solvers such as Z3 [4, 7]. As such, our work focuses on inferring likely candidate invariant as validating a candidate can be done efficiently.

Data-driven Methods. Data-driven loop invariant inference methods use program traces recording the state of each variable in the program on every iteration of the loop to

guide the invariant generation process. Since an invariant must hold for any valid execution, the collected traces can be used to rule out many potential invariants. Formally, given a set of program traces X, data-driven invariant inference finds SMT formulas F such that:

$$\forall x \in X, F(x) = True$$

2.2 Basic Fuzzy Logic

Our approach to SMT formula learning is based on a form of differentiable logic called Basic Fuzzy Logic (BL). BL is a relaxation of first-order logic that operates on continuous truth values on the interval $[0, 1]$ instead of on boolean values. BL uses a class of functions called *t-norms* ($\otimes$), which preserves the semantics of boolean conjunctions on continuous truth values. T-norms are required to be consistent with boolean logic, monotonic on their domain, commutative, and associative [14]. Formally, a t-norm is defined $\otimes : [0, 1]^2 \to [0, 1]$ such that:

- $\otimes$ is consistent for any $t \in [0, 1]$:

$$t \otimes 1 = t \qquad t \otimes 0 = 0$$

- $\otimes$ is commutative and associative for any $t \in [0, 1]$:

$$t_1 \otimes t_2 = t_2 \otimes t_1 \qquad t_1 \otimes (t_2 \otimes t_3) = (t_1 \otimes t_2) \otimes t_3$$

- $\otimes$ is monotonic (nondecreasing) for any $t \in [0, 1]$:

$$t_1 \leq t_2 \implies t_1 \otimes t_3 \leq t_2 \otimes t_3$$

BL additionally requires that t-norms be continuous. *T-conorms* ($\oplus$) are derived from t-norms via DeMorgan's law and operate as disjunctions on continuous truth values, while negations are defined $\neg t := 1 - t$.

In this paper, we keep t-norms abstract in our formulations to make the framework general. Prior work [30] found product t-norm $x \otimes y = x \cdot y$ perform better in Continuous Logic Networks. For this reason, we use product t-norm in our final implementation, although other t-norms (e.g., Godel) can also be used.

2.3 Continuous Logic Networks

We perform SMT formula learning with Continuous Logic Networks (CLNs), a neural architecture introduced in [30] that are able to learn SMT formulas directly from data. These can be used to learn loop invariants from the observed behavior of the program.

CLNs are based on a parametric relaxation of SMT formulas that maps the SMT formulation from boolean first-order logic to BL. The model defines the operator $\mathcal{S}$. Given an quantifier-free SMT formula $F : X \to \{True, False\}$, $\mathcal{S}$ maps it to a continuous function $\mathcal{S}(F) : X \to [0, 1]$. In order for the continuous model to be both usable in gradient-guided optimization while also preserving the semantics of boolean logic, it must fulfill three conditions:

1. It must preserve the meaning of the logic, such that the continuous truth values of a valid assignment are always greater than the value of an invalid assignment:

$$(F(x) = True \wedge F(x') = False) \implies (\mathcal{S}(F)(x) > \mathcal{S}(F)(x'))$$

2. It must be must be continuous and smooth (i.e. differentiable almost everywhere) to facilitate training.
3. It must be strictly increasing as an unsatisfying assignment of terms approach satisfying the mapped formula, and strictly decreasing as a satisfying assignment of terms approach violating the formula.

$\mathcal{S}$ is constructed as follows to satisfy these requirements. The logical relations $\{\wedge, \vee, \neg\}$ are mapped to their continuous equivalents in BL:

$$\begin{aligned}
&\text{Conjunction:} && \mathcal{S}(F_1 \wedge F_2) \triangleq \mathcal{S}(F_1) \otimes \mathcal{S}(F_2) \\
&\text{Disjunction:} && \mathcal{S}(F_1 \vee F_2) \triangleq \mathcal{S}(F_1) \oplus \mathcal{S}(F_2) \\
&\text{Negation:} && \mathcal{S}(\neg F) \triangleq 1 - \mathcal{S}(F)
\end{aligned}$$

where any F is an SMT formula. $\mathcal{S}$ defines SMT predicates $\{=, \neq, <, \leq, >, \geq\}$ with functions that map to continuous truth values. This mapping is defined for $\{>, \geq\}$ using sigmoids with a shift parameter ϵ and smoothing parameter B:

$$\begin{aligned}
&\text{Greater Than:} && \mathcal{S}(x_1 > x_2) \triangleq \frac{1}{1 + e^{-B(x_1 - x_2 - \epsilon)}} \\
&\text{Greater or Equal to:} && \mathcal{S}(x_1 \geq x_2) \triangleq \frac{1}{1 + e^{-B(x_1 - x_2 + \epsilon)}}
\end{aligned}$$

where $x_1, x_2 \in R$. Mappings for other predicates are derived from their logical relations to $\{>, \geq\}$:

$$\begin{aligned}
&\text{Less Than:} && \mathcal{S}(x_1 < x_2) = \mathcal{S}(\neg(x_1 \geq x_2)) \\
&\text{Less or Equal to:} && \mathcal{S}(x_1 \leq x_2) = \mathcal{S}(\neg(x_1 > x_2)) \\
&\text{Equality:} && \mathcal{S}(x_1 = x_2) = \mathcal{S}((x_1 \geq x_2) \wedge (x_1 \leq x_2)) \\
&\text{Inequality:} && \mathcal{S}(x_1 \neq x_2) = \mathcal{S}(\neg(x_1 = x_2))
\end{aligned}$$

Using these definitions the parametric relaxation $\mathcal{S}$ satisfies all three conditions for sufficiently large B and sufficiently small ϵ. Based on this parametric relaxation $S(F)$, we build a Continuous Logic Network model M, which is a computational graph of $S(F)(x)$ with learnable parameters W. When training a CLN, loss terms are applied to penalize small B, ensuring that as the loss approaches 0 the CLN will learn a precise formula. Under these conditions, the following relationship holds between a trained CLN model M with coefficients W and its associated formula F for a given set of data points, X:

$$\forall x \in X, M(x; W) = 1 \iff F(x; W) = True$$

Figure 2 shows an example CLN for the formula on a single variable x:

$$F(x) = (x = 1) \vee (x \geq 5) \vee (x \geq 2 \wedge x \leq 3)$$

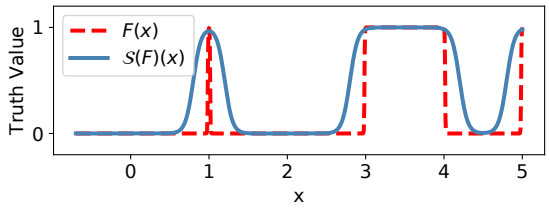


Figure 2. Plot of the formula $F(x) \triangleq (x = 1) \vee (x \geq 5) \vee (x \geq 2 \wedge x \leq 3)$ and its associated CLN $M(x)$.

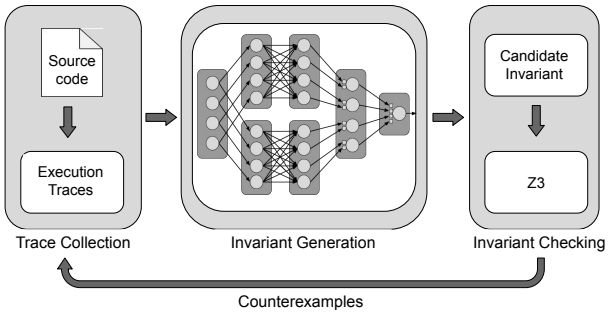


Figure 3. Overview of method consisting of 3 phases: trace generation from source code file, G-CLN training, and invariant extraction followed by checking with Z3.

```
// pre: (n >= 0)
a=0; s=1; t=1;
while (s<=n){
  log(a,s,t,n);
  a += 1;
  t += 2;
  s += t;
}
log(a,s,t,n);
```

(a) Program instrumented to log samples.

1	a	t	...	as	t^2	st
1	0	1		0	1	1
1	1	3		4	9	12
1	2	5		18	25	45
1	3	7		48	49	112

(b) Sample data points generated with maximum degree of 2.

Figure 4. Training data generation for the program shown in Figure 1b.

3 Workflow

Figure 3 illustrates our overall workflow for loop invariant inference. Our approach has three stages: (i) We first instrument the program and execute to generate trace data. (ii) We then construct and train a G-CLN model to fit the trace data. (iii) We extract a candidate loop invariant from the model and check it against a specification.

Given a program loop, we modify it to record variables for each iteration and then execute the program to generate samples. Figure 4a illustrates this process for the `sqrt` program from Figure 1b. The program has the input n with precondition $(n \geq 0)$, so we execute with values $n = 0, 1, 2, \ldots$ for inputs in a set range. Then we expand the samples to all candidate terms for the loop invariant. By default, we enumerate all the monomials over program variables up to a given degree *maxDeg*, as shown in Figure 4b. Our system can be configured to consider other non-linear terms like x^y.

We then construct and train a G-CLN model using the collected trace data. We use the model architecture described in §5.2.1, with PBQUs for bounds learning using the procedure in §5.2.2. After training the model, the SMT formula for the invariant is extracted by recursively descending through the model and extracting clauses whose gating parameters are above 0.5, as outlined in Algorithm 1. On the `sqrt` program, the model will learn the invariant $\left(a^2 \leq n\right) \wedge (t = 2a + 1) \wedge \left(su = (a + 1)^2\right)$.

Finally, if z3 returns a counterexample, we will incorporate it into the training data, and rerun the three stages with more generated samples. Our system repeats until a valid invariant is learned or times out.

4 Theory

In this section, we first present our gating construction for CLNs and prove gated CLNs are *sound* with regard to their underlying discrete logic. We then describe *Piecewise Biased Quadratic Units*, a specific activation function construction for learning tight bounds on inequalities, and provide theoretical guarantees. Finally we present a technique to relax loop semantics and generate more samples when needed.

4.1 Gated t-norms and t-conorms

In the original CLNs [30], a formula template is required to learn the invariant. For example, to learn the invariant $(x + y = 0) \vee (x - y = 0)$, we have to provide the template $(w_1 x + w_2 y + b_1 = 0) \vee (w_3 x + w_4 y + b_2 = 0)$, which can be constructed as a CLN model to learn the coefficients. So, we have to know in advance whether the correct invariant is an atomic clause, a conjunction, a disjunction, or a more complex logical formula. To tackle this problem, we introduce gated t-norms and gated t-conorms.

Given a classic t-norm $T(x, y) = x \otimes y$, we define its associated gated t-norm as

$$T_G(x, y; g_1, g_2) = (1 + g_1(x - 1)) \otimes (1 + g_2(y - 1))$$

Here $g_1, g_2 \in [0, 1]$ are gate parameters indicating if x and y are activated, respectively. The following equation shows the intuition behind gated t-norms.

$$T_G(x, y; g_1, g_2) = \begin{cases} x \otimes y & g_1 = 1, g_2 = 1 \\ x & g_1 = 1, g_2 = 0 \\ y & g_1 = 0, g_2 = 1 \\ 1 & g_1 = 0, g_2 = 0 \end{cases}$$

Informally speaking when $g_1 = 1$, the input x is activated and behaves as in the classic t-norm. When $g_1 = 0$, x is deactivated and discarded. When $0 < g_1 < 1$, the value of g_1 indicates how much information we should take from x. This pattern also applies for g_2 and y.

We can prove that $\forall g_1, g_2 \in [0,1]$, the gated t-norm is continuous and monotonically increasing with regard to x and y, thus being well suited for training.

Like the original t-norm, the gated t-norm can be easily extended to more than two operands. In the case of three operands, we have the following:

$$T_G(x,y,z;g_1,g_2,g_3) = (1+g_1(x-1)) \otimes (1+g_2(y-1)) \otimes (1+g_3(z-1))$$

Using De Morgan's laws $x \oplus y = 1-(1-x)\otimes(1-y)$, we define gated t-conorm as

$$T'_G(x,y;g_1,g_2) = 1-(1-g_1x)\otimes(1-g_2y)$$

Similar to gated t-norms, gated t-conorms have the following property.

$$T'_G(x,y;g_1,g_2) = \begin{cases} x \oplus y & g_1=1, g_2=1 \\ x & g_1=1, g_2=0 \\ y & g_1=0, g_2=1 \\ 0 & g_1=0, g_2=0 \end{cases}$$

Now we replace the original t-norms and t-conorms in CLN with our gated alternatives, which we diagram in Figure 5. Figure 6 demonstrates a gated CLN for representing an SMT formula. With the gated architecture, the gating parameters g_1, g_2 for each gated t-norm or gated t-conorm are made learnable during model training, such that the model can decide which input should be adopted and which should be discarded from the training data. This improves model flexibility and does not require a specified templates.

Now, we formally state the procedure to retrieve the SMT formula from a gated CLN model recursively in Algorithm 1. Abusing notation for brevity, $\mathcal{M}_i$ in line 1 represent the output node of model $\mathcal{M}_i$ rather than the model itself, and the same applies for line 8 and line 15. *BuildAtomicFormula* in line 18 is a subroutine to extract the formula for a model with no logical connectives (e.g., retrieving $x+y+z=0$ in Figure 6). The linear weights which have been learned serve as the coefficients for the terms in the equality or inequality depending on the associated activation function.

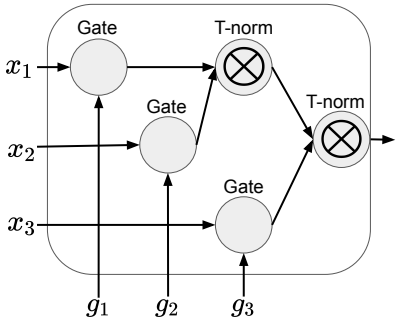


Figure 5. Example of gated t-norm with three operands constructed from binary t-norms. The gated t-conorm is done similarly.

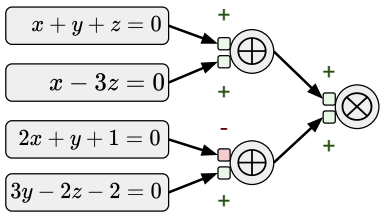


Figure 6. An instance of gated CLN. "+" means activated (g=1) and "-" means deactivated (g=0). The SMT formula learned is $(3y-3z-2=0) \wedge ((x-3z=0) \vee (x+y+z=0))$.

Algorithm 1 Formula Extraction Algorithm.

Input: A gated CLN model $\mathcal{M}$, with input nodes $\mathcal{X} = \{x_1, x_2, ..., x_n\}$ and output node p.
Output: An SMT formula F
Procedure ExtractFormula($\mathcal{M}$)

1: **if** $p = T_G(\mathcal{M}_1, ..., \mathcal{M}_n; g_1, ..., g_n)$ **then**
2: F := True
3: **for** $i := 1$ **to** n **do**
4: **if** $g_i > 0.5$ **then**
5: $F := F \wedge ExtractFormula(\mathcal{M}_i)$
6: **else if** $p = T'_G(\mathcal{M}_1, ..., \mathcal{M}_n; g_1, ..., g_n)$ **then**
7: F := False
8: **for** $i := 1$ **to** n **do**
9: **if** $g_i > 0.5$ **then**
10: $F := F \vee ExtractFormula(\mathcal{M}_i)$
11: **else if** $p = 1 - \mathcal{M}_1$ **then**
12: $F := \neg ExtractFormula(\mathcal{M}_1)$
13: **else**
14: $F := BuildAtomicFormula(\mathcal{M})$

Finally, we need to round the learned coefficients to integers. We first scale the coefficients so that the maximum is 1 and then round to the nearest rational number using a maximum possible denominator. We check if each rounded invariant fits all the training data and discard the invalid ones.

In Theorem 4.1, we will show that the extracted SMT formula is equivalent to the gated CLN model under some constraints. We first introduce a property of t-norms that is defined in the original CLNs [30].

Property 1. $\forall t\ u,\ (t>0) \wedge (u>0) \implies (t \otimes u > 0)$.

The product t-norm $x \otimes y = x \cdot y$, which is used in our implementation, has this property.

Note that the hyperparameters $c_1, c_2, \epsilon, \sigma$ in Theorem 4.1 will be formally introduced in §4.2 and are unimportant here. One can simply see

$$\lim_{\substack{c_1 \to 0, c_2 \to \infty \\ \sigma \to 0, \epsilon \to 0}} \mathcal{M}(x; c_1, c_2, \sigma, \epsilon)$$

as the model output $\mathcal{M}(x)$.

Theorem 4.1. *For a gated CLN model $\mathcal{M}$ with input nodes $\{x_1, x_2, ..., x_n\}$ and output node p, if all gating parameters $\{g_i\}$ are either 0 or 1, then using the formula extraction algorithm, the recovered SMT formula F is equivalent to the gated CLN model $\mathcal{M}$. That is, $\forall x \in R^n$,*

$$F(x) = True \iff \lim_{\substack{c_1\to 0, c_2\to\infty \\ \sigma\to 0, \epsilon\to 0}} \mathcal{M}(x; c_1, c_2, \sigma, \epsilon) = 1 \quad (1)$$

$$F(x) = False \iff \lim_{\substack{c_1\to 0, c_2\to\infty \\ \sigma\to 0, \epsilon\to 0}} \mathcal{M}(x; c_1, c_2, \sigma, \epsilon) = 0 \quad (2)$$

as long as the t-norm in $\mathcal{M}$ satisfies Property 1.

Proof. We prove this by induction over the formula structure considering four cases: atomic, negation, T-norm, and T-conorm. For brevity, we sketch the T-norm case here and provide the full proof in our extended technical report [39].

T-norm Case. If $p = T_G(\mathcal{M}_1, ..., \mathcal{M}_n; g_1, ..., g_n)$, which means the final operation in $\mathcal{M}$ is a gated t-norm, we know that for each submodel $\mathcal{M}_1, ..., \mathcal{M}_n$ the gating parameters are all either 0 or 1. By the induction hypothesis, for each $\mathcal{M}_i$, using Algorithm 1, we can extract an equivalent SMT formula F_i satisfying Eq. (1)(2). Then we can prove the full model $\mathcal{M}$ and the extracted formula F also satisfy Eq. (1)(2), using the induction hypothesis and the properties and t-norms. □

The requirement of all gating parameters being either 0 or 1 indicates that no gate is partially activated (e.g., $g_1 = 0.6$). Gating parameters between 0 and 1 are acceptable during model fitting but should be eliminated when the model converges. In practice this is achieved by gate regularization which will be discussed in §5.2.1.

Theorem 4.1 guarantees the soundness of the gating methodology with regard to discrete logic. Since the CLN architecture is composed of operations that are sound with regard to discrete logic, this property is preserved when gated t-norms and t-conorms are used in the network.

Now the learnable parameters of our gated CLN include both linear weights W as in typical neural networks, and the gating parameters $\{g_i\}$, so the model can represent a large family of SMT formulas. Given a training set X, when the gated CLN model $\mathcal{M}$ is trained to $\mathcal{M}(x) = 1$ for all $x \in X$, then from Theorem 4.1 the recovered formula F is guaranteed to hold true for all the training samples. That is, $\forall x \in X, F(x) = True$.

4.2 Parametric Relaxation for Inequalities

For learned inequality constraints to be useful in verification, they usually need to constrain the loop behavior as tightly as possible. In this section, we define a CLN activation function, the *bounds learning activation*, which naturally learns tight bounds during training while maintaining the soundness guarantees of the CLN mapping to SMT.

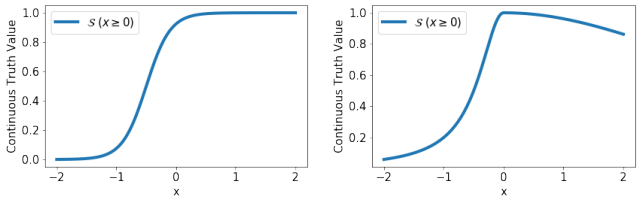
$$\mathcal{S}(t \geq u) \triangleq \begin{cases} \frac{c_1^2}{(t-u)^2+c_1^2} & t < u \\ \frac{c_2^2}{(t-u)^2+c_2^2} & t \geq u \end{cases} \quad (3)$$

Here c_1 and c_2 are two constants. The following limit property holds.

$$\lim_{\substack{c_1\to 0 \\ c_2\to\infty}} S(t \geq u) = \begin{cases} 0 & t < u \\ 1 & t \geq u \end{cases}$$

Intuitively speaking, when c_1 approaches 0 and c_2 approaches infinity, $\mathcal{S}(x \geq 0)$ will approach the original semantic of predicate $\geq$. Figure 7b provides an illustration of our parametric relaxation for $\geq$.

Compared with the sigmoid construction in the original CLNs (Figure 7a), our parametric relaxation penalizes very large x, where $x \geq 0$ is absolutely correct but not very informative because the bound is too weak. In general, our piecewise mapping punishes data points farther away from the boundary, thus encouraging to learn a tight bound of the samples. On the contrary, the sigmoid construction encourages samples to be far from the boundary, resulting in loose bounds which are not useful for verification.



(a) Plot of $\mathcal{S}(x \geq 0)$ with the CLNs' sigmoid construction.

(b) Plot of $\mathcal{S}(x \geq 0)$ with our piecewise construction.

Figure 7. Comparison of the mapping $\mathcal{S}$ on $\geq$. The hyperparameters are $B = 5$, $\epsilon = 0.5$, $c_1 = 0.5$, and $c_2 = 5$.

Since the samples represent only a subset of reachable states from the program, encouraging a tighter bound may potentially lead to overfitting. However, we ensure soundness by later checking learned invariants via a solver. If an initial bound is too tight, we can incorporate counterexamples to the training data. Our empirical results show this approach works well in practice.

Given a set of n k-dimensional samples $\{[x_{11}, ..., x_{1k}], ..., [x_{n1}, ..., x_{nk}]\}$, where x_{ij} denotes the value of variable x_j in the i-th sample, we want to learn an inequality $w_1x_1 + ... + w_kx_k + b \geq 0$ for these samples. The desirable properties of such an inequality is that it should be valid for all points, and have as tight as a fit as possible. Formally, we define a "desired" inequality as:

$$\begin{aligned} &\forall i \in \{1, ..., n\}, w_1x_{i1} + ... + w_kx_{ik} + b \geq 0 \\ &\exists j \in \{1, ..., n\}, w_1x_{j1} + ... + w_kx_{jk} + b = 0 \end{aligned} \quad (4)$$

Our parametric relaxation for $\geq$ shown in Eq. 3 can always learn an inequality which is very close to a "desired" one with proper c_1 and c_2. Theorem 4.2 put this formally.

Theorem 4.2. *Given a set of n k-dimensional samples with the maximum L2-norm l, if $c_1 \leq 2l$ and $c_1 \cdot c_2 \geq 8\sqrt{n}l^2$, and the weights are constrained as $\sum_{i=1}^{k} w_i^2 = 1$, then when the model converges, the learned inequality has distance at most $c_1/\sqrt{3}$ from a "desired" inequality.*

Proof. See the extended technical report [39]. □

Recall that c_1 is a small constant, so $c_1/\sqrt{3}$ can be considered as the error bound of inequality learning. Although we only proved the theoretical guarantee when learning a single inequality, our parametric relaxation for inequalities can be connected with other inequalities and equalities with conjunctions and disjunctions under a single CLN model.

Based on our parametric relaxation for $\geq$, other inequality predicates can be defined accordingly.

$$\mathcal{S}(t \leq u) \triangleq \begin{cases} \frac{c_2^2}{(t-u)^2+c_2^2} & t < u \\ \frac{c_1^2}{(t-u)^2+c_1^2} & t \geq u \end{cases}$$

$$\mathcal{S}(t > u) \triangleq \mathcal{S}(t \geq (u+\epsilon)) \quad \mathcal{S}(t < u) \triangleq \mathcal{S}(t \leq (u-\epsilon))$$

where ϵ is a set small constant.

For our parametric relaxation, some axioms in classic logic just approximately rather than strictly hold (e.g., $t \leq u = \neg(t > u)$). They will strictly hold when $c_1 \to 0$ and $c_2 \to \infty$.

We reuse the Gaussian function as the parametric relaxation for equalities [30]. Given a small constant σ,

$$\mathcal{S}(t = u) \triangleq \exp\left(-\frac{(t-u)^2}{2\sigma^2}\right)$$

4.3 Fractional Sampling

In some cases, the samples generated from the original program are insufficient to learn the correct invariant due to dominating growth of some terms (higher-order terms in particular) or limited number of loop iterations. To generate more fine-grained yet valid samples, we perform *Fractional Sampling* to relax the program semantics to continuous functions without violating the loop invariants by varying the initial value of program variables. The intuition is as follows.

Any numerical loop invariant I can be viewed as a predicate over program variables V initialized with V_0 such that

$$\forall V,\ V_0 \mapsto^* V \implies I(V) \tag{5}$$

where $V_0 \mapsto^* V$ means starting from initial values V_0 and executing the loop for 0 or more iterations ends with values V for the variables.

Now we relax the initial values X_0 and see them as input variables V_I, which may carry arbitrary values. The new loop program will have variables $V \cup V_I$. Suppose we can learn an invariant predicate I' for this new program, i.e.,

$$\forall V_I\ V,\ V_I \mapsto^* V \implies I'(V_I \cup V) \tag{6}$$

Then let $V_I = V_0$, Eq. (6) will become

$$\forall V,\ V_0 \mapsto^* V \implies I'(V_0 \cup V) \tag{7}$$

Now V_0 is a constant, and $I'(V_0 \cup V)$ satisfies Eq. (5) thus being a valid invariant for the original program. In fact, if we learn predicate I' successfully then we have a more general loop invariant that can apply for any given initial values.

Figure 8 shows how Fractional Sampling can generate more fine-grained samples with different initial values, making model fitting much easier in our data-driven learning system. The correct loop invariant for the program in Figure 8a is

$$(4x = y^4 + 2y^3 + y^2) \wedge (y \leq k)$$

To learn the equality part $(4x = y^4 + 2y^3 + y^2)$, if we choose $maxDeg = 4$ and apply normal sampling, then six terms $\{1, y, y^2, y^3, y^4, x\}$ will remain after the heuristic filters in §5.1.3. Figure 8b shows a subset of training samples without Fractional Sampling (the column of term 1 is omitted).

When y becomes larger, the low order terms $1, y$, and y^2 become increasingly negligible because they are significantly smaller than the dominant terms y^4 and x. In practice we observe that the coefficients for x^4 and y can be learned accurately but not for $1, y, y^2$. To tackle this issue, we hope to generate more samples around $y = 1$ where all terms are on the same level. Such samples can be easily generated by feeding more initial values around $y = 1$ using Fractional

```
//pre: x = y = 0
//     /\ k >= 0
while (y < k) {
  y++;
  x += y * y * y;
}
//post: 4x == k^2
//    * (k + 1)^2
```

(a) The ps4 program in the benchmark.

x	y	y^2	y^3	y^4
0	0	0	0	0
1	1	1	1	1
9	2	4	8	16
36	3	9	27	81
100	4	16	64	256
225	5	25	125	625

(b) Training data generated without Fractional Sampling.

x	y	y^2	y^3	y^4	x_0	y_0	y_0^2	y_0^3	y_0^4
-1	-0.6	0.36	-0.22	0.13	-1	-0.6	0.36	-0.22	0.13
-0.9	0.4	0.16	0.06	0.03	-1	-0.6	0.36	-0.22	0.13
1.8	1.4	1.96	2.74	3.84	-1	-0.6	0.36	-0.22	0.13
0	-1.2	1.44	-1.73	2.07	0	-1.2	1.44	-1.73	2.07
0	-0.2	0.04	-0.01	0.00	0	-1.2	1.44	-1.73	2.07
0.5	0.8	0.64	0.52	0.41	0	-1.2	1.44	-1.73	2.07

(c) Training data generated with fractional sampling.

Figure 8. An example of Fractional Sampling.

Sampling. Table 8c shows some generated samples from $x_0 = -1, y_0 = -0.6$ and $x_0 = 0, y_0 = -1.2$.

Now we have more samples where terms are on the same level, making the model easier to converge to the accurate solution. Our gated CLN model can correctly learn the relaxed invariant $4x - y^4 - 2y^3 - y^2 - 4x_0 + y_0^4 + 2y_0^3 + y_0^2 = 0$. Finally we return to the exact initial values $x_0 = 0, y_0 = 0$, and the correct invariant for the original program will appear $4x - y^4 - 2y^3 - y^2 = 0$.

Note that for convenience, in Eq. (5)(6)(7), we assume all variables are initialized in the original program and all are relaxed in the new program. However, the framework can easily extends to programs with uninitialized variables, or we just want to relax a subset of initialized variables. Details on how fractional sampling is incorporated in our system are provided in §5.4.

5 Nonlinear Invariant Learning

In this section, we describe our overall approach for nonlinear loop invariant inference. We first describe our methods for stable CLN training on nonlinear data. We then give an overview of our model architecture and how we incorporate our inequality activation function to learn inequalities. Finally, we show how we extend our approach to also learn invariants that contain external functions.

5.1 Stable CLN Training on Nonlinear Data

Nonlinear data causes instability in CLN training due to the large number of terms and widely varying magnitudes it introduces. We address this by modifying the CLN architecture to normalize both inputs and weights on a forward execution. We then describe how we implement term dropout, which helps the model learn precise SMT coefficients.

5.1.1 Data Normalization. Exceedingly large inputs cause instability and prevent the CLN model from converging to precise SMT formulas that fit the data. We therefore modify the CLN architecture such that it rescales its inputs so the L2-norm equals a set value l. In our implementation, we used $l = 10$.

We take the program in Figure 1b as an example. The raw samples before normalization is shown in Figure 4b. The monomial terms span in a too wide range, posing difficulty for network training. With data normalization, each sample (i.e., each row) is proportionally rescaled to L2-norm 10. The normalized samples are shown in Table 1.

Table 1. Training data after normalization for the program in Figure 1b, which computes the integer square root.

1	a	t	...	as	t^2	st
0.70	0	0.70		0	0.70	0.70
0.27	0.27	0.81		1.08	2.42	3.23
0.13	0.25	0.63		2.29	3.17	5.71
0.06	0.19	0.45		3.10	3.16	7.23

Now the samples occupy a more regular range. Note that data normalization does not violate model correctness. If the original sample $(t_1, t_2, ...t_k)$ satisfies the equality $w_1t_1 + w_2t_2 + ... + w_kt_k = 0$ (note that t_i can be a higher-order term), so does the normalized sample and vice versa. The same argument applies to inequalities.

5.1.2 Weight Regularization. For both equality invariant $w_1x_1 + ... + w_mx_m + b = 0$ and inequality invariant $w_1x_1 + ... + w_mx_m + b \geq 0$, $w_1 = ... = w_m = b = 0$ is a true solution. To avoid learning this trivial solution, we require at least one of $\{w_1, ..., w_m\}$ is non-zero. A more elegant way is to constrain the L^p-norm of the weight vector to constant 1. In practice we choose L2-norm as we did in Theorem 4.2. The weights are constrained to satisfy

$$w_1^2 + ... + w_m^2 = 1$$

5.1.3 Term Dropout. Given a program with three variables $\{x, y, z\}$ and $maxDeg = 2$, we will have ten candidate terms $\{1, x, y, z, x^2, y^2, z^2, xy, xz, yz\}$. The large number of terms poses difficulty for invariant learning, and the loop invariant in a real-world program is unlikely to contain all these terms. We use two methods to select terms. First the growth-rate-based heuristic in [33] is adopted to filter out unnecessary terms. Second we apply a random dropout to discard terms before training.

Dropout is a common practice in neural networks to avoid overfitting and improve performance. Our dropout is randomly predetermined before the training, which is different from the typical weight dropout in deep learning [37]. Suppose after the growth-rate-based filter, seven terms $\{1, x, y, z, x^2, y^2, xy\}$ remain. Before the training, each input term to a neuron may be discarded with probability p.

The importance of dropout is twofold. First it further reduces the number of terms in each neuron. Second it encourages G-CLN to learn more simple invariants. For example, if the desired invariant is $(x - y - 1 = 0) \wedge (x^2 - z = 0)$, then a neuron may learn their linear combination (e.g., $2x - 2y - 2 + x^2 - z = 0$) which is correct but not human-friendly. If the term x is discarded in one neuron then that neuron may learn $x^2 - z = 0$ rather than $2x - 2y - 2 + x^2 - z = 0$. Similarly, if the terms x^2 and xy are discarded in another neuron, then that neuron may learn $x - y - 1 = 0$. Together, the entire network consisting of both neurons will learn the precise invariant.

Since the term dropout is random, a neuron may end up having no valid invariant to learn (e.g., both x and x^2 are discarded in the example above). But when gating (§4.1) is adopted, this neuron will be deactivated and remaining neurons may still be able to learn the desired invariant. More details on gated model training will be provided in §5.2.1.

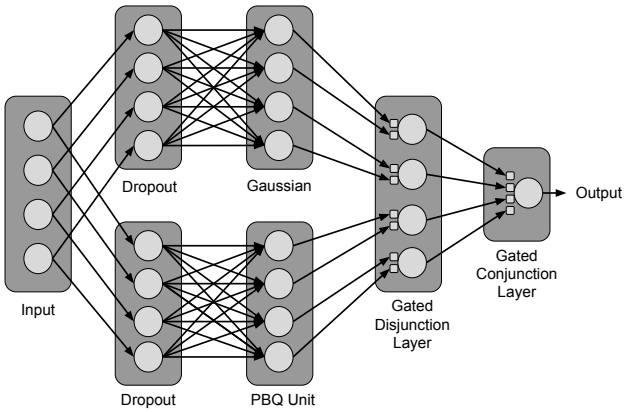
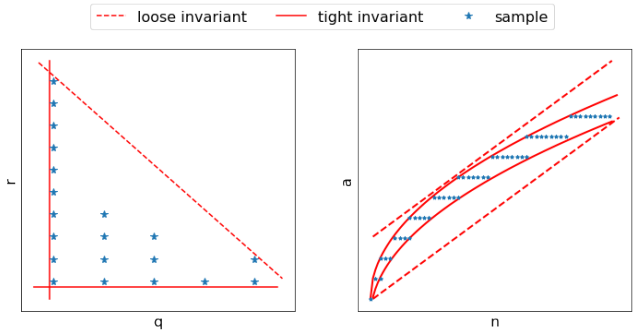


Figure 9. Diagram of G-CLN model. Additional disjunction and conjunction layers may be added to learn more complex SMT formulas.



(a) Learned inequality bounds. **(b)** Learned inequality bounds on `sqrt`.

Figure 10. Examples of 2 dimensional bound fitting.

5.2 Gated CLN Invariant Learning

Here we describe the Gated CLN architecture and how we incorporate the bounds learning activation function to learn general nonlinear loop invariants.

5.2.1 Gated CLN Architecture. Architecture. In §3.1, we introduce gated t-norm and gated t-conorm, and illustrate how they can be integrated in CLN architecture. Theoretically, the gates can cascade to many layers, while in practice, we use a gated t-conorm layer representing logical OR followed by a gated t-norm layer representing logical AND, as shown in Figure 9. The SMT formula extracted from such a gated CLN architecture will be in conjunctive normal form (CNF). In other words, G-CLN is parameterized by m and n, where the underlying formula can be a conjunction of up to m clauses, where each clause is a disjunction of n atomic clauses. In the experiments we set m=10, n=2.

Gated CLN Training. For the sake of discussion, consider a gated t-norm with two inputs. We note that the gating parameters g_1 and g_2 have an intrinsic tendency to become 0 in our construction. When $g_1 = g_2 = 0$, $T_G(x, y; g_1, g_2) = 1$ regardless of the truth value of the inputs x and y. So when training the gated CLN model, we apply regularization on g_1, g_2 to penalize small values. Similarly, for a gated t-conorm, the gating parameters g_1, g_2 have an intrinsic tendency to become 1 because $x \oplus y$ has a greater value than x and y. To resolve this we apply regularization pressure on g_1, g_2 to penalize close-to-1 values.

In the general case, given training set X and gate regularization parameters λ_1, λ_2, the model will learn to minimize the following loss function with regard to the linear weights W and gating parameters G,

$$\mathcal{L}(X; W, G) = \sum_{x \in X} (1 - \mathcal{M}(\mathbf{x}; W, G)) + \lambda_1 \sum_{g_i \in T_G} (1 - g_i) + \lambda_2 \sum_{g_i \in T'_G} g_i$$

By training the G-CLN with this loss formulation, the model tends to learn a formula F satisfying each training sample (recall $F(x) = True \Leftrightarrow \mathcal{M}(x) = 1$ in §4.1). Together, gating and regularization prunes off poorly learned clauses, while preventing the network from pruning off too aggressively. When the training converges, all the gating parameters will be very close to either 1 or 0, indicating the participation of the clause in the formula. The invariant is recovered using Algorithm 1.

5.2.2 Inequality Learning. Inequality learning largely follows the same procedure as equality learning with two differences. First, we use the PBQU activation (i.e., the parametric relaxation for $\geq$) introduced in §4.2, instead of the Gaussian activation function (i.e., the parametric relaxation for $=$). This difference is shown in Figure 9. As discussed in §4.2, the PBQU activation will learn tight inequalities rather than loose ones.

Second, we structure the dropout on inequality constraints to consider all possible combinations of variables up to a set number of terms and maximum degree (up to 3 terms and 2nd degree in our evaluation). We then train the model following the same optimization used in equality learning, and remove constraints that do not fit the data based on their PBQU activations after the model has finished training.

When extracting a formula from the model we remove poorly fit learned bounds that have PBQU activations below a set threshold. As discussed in §4.2, PBQU activations penalizes points that are farther from its bound. The tight fitting bounds in Figures 10a and 10b with solid red lines have PBQU activations close to 1, while loose fitting bounds with dashed lines have PBQU activations close to 0. After

selecting the best fitting bounds, we check against the loop specification and remove any remaining constraints that are unsound. If the resulting invariant is insufficient to prove the postcondition, the model is retrained using the counterexamples generated during the specification check.

5.3 External Function Calls

In realistic applications, loops are not entirely whitebox and may contain calls to external functions for which the signature is provided but not the code body. In these cases, external functions may also appear in the loop invariant. To address these cases, when an external function is present, we sample it during loop execution. To sample the function, we execute it with all combinations of variables in scope during sampling that match its call signature.

For example, the function $gcd : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$, for greatest common divisor, is required in the invariant for four of the evaluation problems that compute either greatest common divisor or least common multiple: (`egcd2`, `egcd3`, `lcm1`, and `lcm2`). In practice, we constrain our system to binary functions, but it is not difficult to utilize static-analysis to extend the support to more complex external function calls. This procedure of constructing terms containing external function calls is orthogonal to our training framework.

5.4 Fractional Sampling Implementation

We apply fractional sampling on a per-program basis when we observe the model is unable to learn the correct polynomial from the initial samples. We first sample on 0.5 intervals, then 0.25, etc. until the model learns a correct invariant. We do not apply fractional sampling to variables involved in predicates and external function calls, such as gcd. In principle, predicate constraints can be relaxed to facilitate more general sampling. We will investigate this in future work.

Among all the programs in our evaluation, only two of them, ps5 and ps6, require fractional sampling. For both of them sampling on 0.5 intervals is sufficient to learn the correct invariant, although more fine grained sampling helps the model learn a correct invariant more quickly. The cost associated with fractional sampling is small ($< 5s$).

6 Evaluation

We evaluate our approach on NLA, a benchmark of common numerical algorithms with nonlinear invariants. We first perform a comparison with two prior works, Numinv and PIE, that use polynomial equation solving and template learning respectively. We then perform an ablation of the methods we introduce in this paper. Finally, we evaluate the stability of our approach against a baseline CLN model.

Evaluation Environment. The experiments described in this section were conducted on an Ubuntu 18.04 server with an Intel XeonE5-2623 v4 2.60GHz CPU, 256Gb of memory, and an Nvidia GTX 1080Ti GPU.

System Configuration. We implement our method with the PyTorch Framework and use the Z3 SMT solver to validate the correctness of the inferred loop invariants. For the four programs involving greatest common divisors, we manually check the validity of the learned invariant since gcd is not supported by z3. We use a G-CLN model with the CNF architecture described in §5.2.1, with a conjunction of ten clauses, each with up to two literals. We use adaptive regularization on the CLN gates. λ_1 is set to $(1.0, 0.999, 0.1)$, which means that λ_1 is initialized as 1.0, and is multiplied by 0.999 after each epoch until it reaches the threshold 0.1. Similarly, λ_2 is set to $(0.001, 1.001, 0.1)$. We try three maximum denominators, $(10, 15, 30)$, for coefficient extraction in §4.1. For the parametric relaxation in §4.2, we set $\sigma = 0.1, c_1 = 1, c_2 = 50$. The default dropout rate in §5.1.3 is 0.3, and will decrease by 0.1 after each failed attempt until it reaches 0. We use the Adam optimizer with learning rate 0.01, decay 0.9996, and max epoch 5000.

6.1 Polynomial Invariant Dataset

We evaluate our method on a dataset of programs that require nonlinear polynomial invariants [22]. The problems in this dataset represent various numerical algorithms ranging from modular division and greatest common denominator (*gcd*) to computing geometric and power series. These algorithms involve up to triply nested loops and sequential loops, which we handle by predicting all the requisite invariants using the model before checking their validity. We sample within input space of the whole program just as we do with single loop problems and annotate the loop that a recorded state is associated with. The invariants involve up to 6^{th} order polynomial and up to thirteen variables.

Performance Comparison. Our method is able to solve 26 of the 27 problems as shown in Table 2, while NumInv solves 23 of 27. Our average execution time was 53.3 seconds, which is a minor improvement to NumInv who reported 69.9 seconds. We also evaluate LoopInvGen (PIE) on a subset of the simpler problems which are available in a compatible format[1]. It was not able to solve any of these problems before hitting a 1 hour timeout. In Table 2, we indicate solved problems with ✓, unsolved problems with ✗, and problems that were not attempted with –.

The single problem we do not solve, `knuth`, is the most difficult problem from a learning perspective. The invariant for the problem, $(d^2 * q - 4 * r * d + 4 * k * d - 2 * q * d + 8 * r == 8 * n) \wedge (mod(d, 2) == 1)$, is one of the most complex in the benchmark. Without considering the external function call to *mod* (modular division), there are already 165 potential terms of degree at most 3, nearly twice as many as next most complex problem in the benchmark, making it difficult to learn a precise invariant with gradient based optimization.

[1]LoopInvGen uses the SyGuS competition format, which is an extended version of smtlib2.

Table 2. Table of problems requiring nonlinear polynomial invariant from NLA dataset. We additionally tested Code2Inv on the same problems as PIE and it fails to solve any within 1 hour. Numinv results are based on Table 1 in [21]. G-CLN solves 26 of 27 problems with an average execution time of 53.3 seconds.

Problem	Degree	# Vars	PIE	NumInv	G-CLN
divbin	2	5	-	✓	✓
cohendiv	2	6	-	✓	✓
mannadiv	2	5	✗	✓	✓
hard	2	6	-	✓	✓
sqrt1	2	4	-	✓	✓
dijkstra	2	5	-	✓	✓
cohencu	3	5	-	✓	✓
egcd	2	8	-	✓	✓
egcd2	2	11	-	✗	✓
egcd3	2	13	-	✗	✓
prodbin	2	5	-	✓	✓
prod4br	3	6	✗	✓	✓
fermat1	2	5	-	✓	✓
fermat2	2	5	-	✓	✓
freire1	2	3	-	✗	✓
freire2	3	4	-	✗	✓
knuth	3	8	-	✓	✗
lcm1	2	6	-	✓	✓
lcm2	2	6	✗	✓	✓
geo1	2	5	✗	✓	✓
geo2	2	5	✗	✓	✓
geo3	3	6	✗	✓	✓
ps2	2	4	✗	✓	✓
ps3	3	4	✗	✓	✓
ps4	4	4	✗	✓	✓
ps5	5	4	-	✓	✓
ps6	6	4	-	✓	✓

We plan to explore better initialization and training strategies to scale to complex loops like `knuth` in future work.

Numinv is able to find the equality constraint in this invariant because its approach is specialized for equality constraint solving. However, we note that NumInv only infers octahedral inequality constraints and does not in fact infer the nonlinear and 3 variable inequalities in the benchmark.

We handle the *mod* binary function successfully in `fermat1` and `fermat2` indicting the success of our model in supporting external function calls. Additionally, for four problems (`egcd2`, `egcd3`, `lcm1`, and `lcm2`), we incorporate the *gcd* external function call as well.

6.2 Ablation Study

We conduct an ablation study to demonstrate the benefits gained by the normalization/regularization techniques and term dropouts as well as fractional sampling. Table 3 notes that data normalization is crucial for nearly all the problems,

Table 3. Table with ablation of various components in the G-CLN model. Each column reports which problems can be solved with G-CLN when that feature ablated.

Problem	Data Norm.	Weight Reg.	Drop-out	Frac. Sampling	Full Method
divbin	✗	✗	✓	✓	✓
cohendiv	✗	✗	✗	✓	✓
mannadiv	✗	✗	✓	✓	✓
hard	✗	✗	✓	✓	✓
sqrt1	✗	✗	✗	✓	✓
dijkstra	✗	✗	✓	✓	✓
cohencu	✗	✗	✓	✓	✓
egcd	✗	✓	✗	✓	✓
egcd2	✗	✓	✗	✓	✓
egcd3	✗	✓	✗	✓	✓
prodbin	✗	✓	✓	✓	✓
prod4br	✗	✓	✓	✓	✓
fermat1	✗	✓	✓	✓	✓
fermat2	✗	✓	✓	✓	✓
freire1	✗	✓	✓	✓	✓
freire2	✗	✓	✗	✓	✓
knuth	✗	✗	✗	✗	✗
lcm1	✗	✓	✓	✓	✓
lcm2	✗	✓	✓	✓	✓
geo1	✗	✓	✓	✓	✓
geo2	✗	✓	✓	✓	✓
geo3	✗	✓	✓	✓	✓
ps2	✓	✗	✓	✓	✓
ps3	✗	✗	✓	✓	✓
ps4	✗	✗	✓	✓	✓
ps5	✗	✗	✓	✗	✓
ps6	✗	✗	✓	✗	✓

especially for preventing high order terms from dominating the training process. Without weight regularization, the problems which involve inequalities over multiple variables cannot be solved; 7 of the 27 problems cannot be solved without dropouts, which help avoid the degenerate case where the network learns repetitions of the same atomic clause. Fractional sampling helps to solve high degree (5^{th} and 6^{th} order) polynomials as the distance between points grow fast.

6.3 Stability

We compare the stability of the gated CLNs with standard CLNs as proposed in [30]. Table 4 shows the result. We ran the two CLN methods without automatic restart 20 times per problem and compared the probability of arriving at a solution. We tested on the example problems described in [30] with disjunction and conjunction of equalities, two problems from Code2Inv, as well as ps2 and ps3 from NLA. As we expected, our regularization and gated t-norms vastly improves the stability of the model as clauses with poorly initialized weights can be ignored by the network. We saw

Table 4. Table comparing the stability of CLN2INV with our method. The statistics reported are over 20 runs per problem with randomized initialization.

Problem	Convergence Rate of CLN	Convergence Rate of G-CLN
Conj Eq	75%	95%
Disj Eq	50%	100%
Code2Inv 1	55%	90%
Code2Inv 11	70%	100%
ps2	70%	100%
ps3	30%	100%

improvements across all the six problems, with the baseline CLN model having an average convergence rate of 58.3%, and the G-CLN converging 97.5% of the time on average.

6.4 Linear Invariant Dataset

We evaluate our system on the Code2Inv benchmark [35] of 133 linear loop invariant inference problems with source code and SMT loop invariant checks. We hold out 9 problems shown to be theoretically unsolvable in [30]. Our system finds correct invariants for all remaining 124 theoretically solvable problems in the benchmark in under 30s.

7 Related Work

Numerical Relaxations. Inductive logic programming (ILP) has been used to learn a logical formula consistent with a set of given data points. More recently, efforts have focused on differentiable relaxations of ILP for learning [8, 17, 27, 38] or program synthesis [36]. Other recent efforts have used formulas as input to graph and recurrent nerual networks to solve Circuit SAT problems and identify Unsat Cores [1, 31, 32]. FastSMT also uses a neural network select optimal SMT solver strategies [3]. In contrast, our work relaxes the semantics of the SMT formulas allowing us to learn SMT formulas.

Counterexample-Driven Invariant Inference. There is a long line of work to learn loop invariants based on counterexamples. ICE-DT uses decision tree learning and leverages counterexamples which violate the inductive verification condition [11, 12, 40]. Combinations of linear classifiers have been applied to learning CHC clauses [40].

A state-of-the-art method, LoopInvGen (PIE) learns the loop invariant using enumerative synthesis to repeatedly add data consistent clauses to strengthen the post-condition until it becomes inductive [25, 26, 34]. For the strengthening procedure, LoopInvGen uses PAC learning, a form of boolean formula learning, to learn which combination of candidate atomic clauses is consistent with the observed data. In contrast, our system learn invariants from trace data.

Neural Networks for Invariant Inference. Recently, neural networks have been applied to loop invariant inference. Code2Inv combines graph and recurrent neural networks to model the program graph and learn from counterexamples [35]. In contrast, CLN2INV uses CLNs to learn SMT formulas for invariants directly from program data [30]. We also use CLNs but incorporate gating and other improvements to be able to learn general nonlinear loop invariants.

Polynomial Invariants. There have been efforts to utilize abstract interpretation to discover polynomial invariants [28, 29]. More recently, Compositional Recurrance Analysis (CRA) performs analysis on abstract domain of transition formulas, but relies on over approximations that prevent it from learning sufficient invariants [9, 18, 19]. Data-driven methods based on linear algebra such as *Guess-and-Check* are able to learn polynomial equality invariants accurately [33]. Guess-and-check learns equality invariants by using the polynomial kernel, but it cannot learn disjunctions and inequalities, which our framework supports natively.

NumInv [21, 23, 24] uses the polynomial kernel but also learns octahedral inequalities. NumInv sacrifices soundness for performance by replacing Z3 with KLEE, a symbolic executor, and in particular, treats invariants which lead to KLEE timeouts as valid. Our method instead is sound and learns more general inequalities than NumInv.

8 Conclusion

We introduce G-CLNs, a new gated neural architecture that can learn general nonlinear loop invariants. We additionally introduce Fractional Sampling, a method that soundly relaxes program semantics to perform dense sampling, and PBQU activations, which naturally learn tight inequality bounds for verification. We evaluate our approach on a set of 27 polynomial loop invariant inference problems and solve 26 of them, 3 more than prior work, as well as improving convergence rate to 97.5% on quadratic problems, a 39.2% improvement over CLN models.

Acknowledgements

The authors are grateful to our shepherd, Aditya Kanade, and the anonymous reviewers for valuable feedbacks that improved this paper significantly. This work is sponsored in part by NSF grants CNS-18-42456, CNS-18-01426, CNS-16-17670, CCF-1918400; ONR grant N00014-17-1-2010; an ARL Young Investigator (YIP) award; an NSF CAREER award; a Google Faculty Fellowship; a Capital One Research Grant; a J.P. Morgan Faculty Award; a Columbia-IBM Center Seed Grant Award; and a Qtum Foundation Research Gift. Any opinions, findings, conclusions, or recommendations that are expressed herein are those of the authors, and do not necessarily reflect those of the US Government, ONR, ARL, NSF, Google, Capital One J.P. Morgan, IBM, or Qtum.

References

[1] Saeed Amizadeh, Sergiy Matusevych, and Markus Weimer. 2019. Learning To Solve Circuit-SAT: An Unsupervised Differentiable Approach. In *International Conference on Learning Representations*. https://openreview.net/forum?id=BJxgz2R9t7

[2] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473* (2014).

[3] Mislav Balunovic, Pavol Bielik, and Martin Vechev. 2018. Learning to solve SMT formulas. In *Advances in Neural Information Processing Systems*. 10317–10328.

[4] A. Biere, H. van Maaren, and T. Walsh. 2009. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. IOS Press, Amsterdam, The Netherlands, The Netherlands.

[5] Andreas Blass and Yuri Gurevich. 2001. Inadequacy of computable loop invariants. *ACM Transactions on Computational Logic (TOCL)* 2, 1 (2001), 1–11.

[6] Werner Damm, Guilherme Pinto, and Stefan Ratschan. 2005. Guaranteed termination in the verification of LTL properties of non-linear robust discrete time hybrid systems. In *International Symposium on Automated Technology for Verification and Analysis*. Springer, 99–113.

[7] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[8] Richard Evans and Edward Grefenstette. 2018. Learning explanatory rules from noisy data. *Journal of Artificial Intelligence Research* 61 (2018), 1–64.

[9] Azadeh Farzan and Zachary Kincaid. 2015. Compositional recurrence analysis. In *2015 Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 57–64.

[10] Carlo A Furia, Bertrand Meyer, and Sergey Velder. 2014. Loop invariants: Analysis, classification, and examples. *ACM Computing Surveys (CSUR)* 46, 3 (2014), 34.

[11] Pranav Garg, Christof Löding, P Madhusudan, and Daniel Neider. 2014. ICE: A robust framework for learning invariants. In *International Conference on Computer Aided Verification*. Springer, 69–87.

[12] Pranav Garg, Daniel Neider, Parthasarathy Madhusudan, and Dan Roth. 2016. Learning invariants using decision trees and implication counterexamples. In *ACM Sigplan Notices*, Vol. 51. ACM, 499–512.

[13] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. 1999. Learning to forget: Continual prediction with LSTM. (1999).

[14] Petr Hájek. 2013. *Metamathematics of fuzzy logic*. Vol. 4. Springer Science & Business Media.

[15] Charles Antony Richard Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (1969), 576–580.

[16] Sergey Ioffe and Christian Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167* (2015).

[17] Angelika Kimmig, Stephen Bach, Matthias Broecheler, Bert Huang, and Lise Getoor. 2012. A short introduction to probabilistic soft logic. In *Proceedings of the NIPS Workshop on Probabilistic Programming: Foundations and Applications*. 1–4.

[18] Zachary Kincaid, Jason Breck, Ashkan Forouhi Boroujeni, and Thomas Reps. 2017. Compositional recurrence analysis revisited. *ACM SIGPLAN Notices* 52, 6 (2017), 248–262.

[19] Zachary Kincaid, John Cyphert, Jason Breck, and Thomas Reps. 2017. Non-linear reasoning for invariant synthesis. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–33.

[20] Hai Lin, Panos J Antsaklis, et al. 2014. Hybrid dynamical systems: An introduction to control and verification. *Foundations and Trends® in Systems and Control* 1, 1 (2014), 1–172.

[21] ThanhVu Nguyen, Timos Antonopoulos, Andrew Ruef, and Michael Hicks. 2017. Counterexample-guided approach to finding numerical invariants. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 605–615.

[22] ThanhVu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. 2012. Using dynamic analysis to discover polynomial and array invariants. In *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 683–693.

[23] ThanhVu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. 2012. Using dynamic analysis to discover polynomial and array invariants. In *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 683–693.

[24] Thanhvu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. 2014. DIG: a dynamic invariant generator for polynomial and array invariants. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 23, 4 (2014), 30.

[25] Saswat Padhi and Todd D. Millstein. 2017. Data-Driven Loop Invariant Inference with Automatic Feature Synthesis. *CoRR* abs/1707.02029 (2017). arXiv:1707.02029 http://arxiv.org/abs/1707.02029

[26] Saswat Padhi, Rahul Sharma, and Todd Millstein. 2016. Data-driven precondition inference with learned features. *ACM SIGPLAN Notices* 51, 6 (2016), 42–56.

[27] Ali Payani and Faramarz Fekri. 2019. Inductive Logic Programming via Differentiable Deep Neural Logic Networks. *arXiv preprint arXiv:1906.03523* (2019).

[28] Enric Rodríguez-Carbonell and Deepak Kapur. 2004. Automatic generation of polynomial loop invariants: Algebraic foundations. In *Proceedings of the 2004 international symposium on Symbolic and algebraic computation*. ACM, 266–273.

[29] Enric Rodríguez-Carbonell and Deepak Kapur. 2007. Generating all polynomial invariants in simple loops. *Journal of Symbolic Computation* 42, 4 (2007), 443–476.

[30] Gabriel Ryan, Justin Wong, Jianan Yao, Ronghui Gu, and Suman Jana. 2020. CLN2INV: Learning Loop Invariants with Continuous Logic Networks. In *International Conference on Learning Representations*. https://openreview.net/forum?id=HJlfuTEtvB

[31] Daniel Selsam and Nikolaj Bjørner. 2019. Guiding high-performance SAT solvers with unsat-core predictions. In *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 336–353.

[32] Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L. Dill. 2019. Learning a SAT Solver from Single-Bit Supervision. In *International Conference on Learning Representations*. https://openreview.net/forum?id=HJMC_iA5tm

[33] Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, Percy Liang, and Aditya V Nori. 2013. A data driven approach for algebraic loop invariants. In *European Symposium on Programming*. Springer, 574–592.

[34] Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, and Aditya V Nori. 2013. Verification as learning geometric concepts. In *International Static Analysis Symposium*. Springer, 388–411.

[35] Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. 2018. Learning loop invariants for program verification. In *Advances in Neural Information Processing Systems*. 7751–7762.

[36] Xujie Si, Mukund Raghothaman, Kihong Heo, and Mayur Naik. 2019. Synthesizing datalog programs using numerical relaxation. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence*. AAAI Press, 6117–6124.

[37] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research* 15, 1 (2014), 1929–1958.

[38] Fan Yang, Zhilin Yang, and William W Cohen. 2017. Differentiable learning of logical rules for knowledge base reasoning. In *Advances in Neural Information Processing Systems*. 2319–2328.

[39] Jianan Yao, Gabriel Ryan, Justin Wong, Suman Jana, and Ronghui Gu. 2020. Learning Nonlinear Loop Invariants with Gated Continuous Logic Networks. arXiv:arXiv:2003.07959

[40] He Zhu, Stephen Magill, and Suresh Jagannathan. 2018. A data-driven CHC solver. In *ACM SIGPLAN Notices*, Vol. 53. ACM, 707–721.