Designing Neural Networks Using Logical Specs

Shikhar Singh, Marko Vasic, Sarfraz Khurshid
Dept. of Electrical and Computer Engineering
The University of Texas at Austin
Austin, USA
{shikhar_singh,vasic}@utexas.edu, khurshid@ece.utexas.edu

***Abstract*—As systems that deploy machine learning models become more and more pervasive, there is an urgent need to ensure their reliability and safety. While recent years have seen a lot of progress in techniques for verification and validation of machine learning models, reasoning about and explaining their behaviors remains challenging. This paper introduces a new approach for creating machine learning models where instead of the traditional supervised learning using data, the models are directly synthesized from specifications, and thus, are correct by construction. Our focus is binary classifiers with boolean features. Specifically, our approach translates relational specifications written in the well-known modeling language Alloy to neural networks that run on the widely used Tensorflow backend. Our key insight is that a slight enhancement of traditional boolean gates can provide a rich intermediate representation that readily translates to neural networks. To translate the enhanced gates to neural networks, we employ a state-of-the-art program synthesis framework that allows us to find *minimal* neural networks. The translation of Alloy specifications then follows the standard translation to boolean logic with the exception of utilizing the enhanced boolean gates, followed by a translation to neural networks. We embody our approach in a prototype tool and use it for experimental evaluation. The experimental results show that our approach allows synthesis of neural networks that are hard to create using traditional training methods.**

I. Introduction

Systems based on machine learning models are rapidly becoming ubiquitous and are being deployed in a wide range of settings, including critical domains such as autonomous vehicles. However, ensuring reliability and safety of these systems largely remains an open problem. While recent years have seen a lot of progress in techniques for gaining insights into machine learning models, development of effective verification and validation methods pose key technical challenges [1], [2]. Once a machine learning model such as a deep neural network is trained, it becomes a highly complex structure that is very hard to reason about.

This paper introduces an approach to create machine learning models that have the desired properties and are *correct by construction* [3]. Instead of the traditional method of training desired models using datasets until a desired performance (accuracy, precision etc.) is achieved, our approach directly synthesizes the models from given specifications. Our focus is *binary classifiers*, i.e., models that classify each input into one of two possible output classes. Specifically, our approach translates *relational* specifications written in Alloy [4] to neural networks that run on the widely used Tensorflow [5] backend.

Alloy is a first-order relational logic with transitive closure, which is particularly suitable for writing specifications of software systems. The Alloy toolset has a state-of-the-art analysis backend that employs off-the-shelf propositional satisfiability (SAT) solvers [6], [7]. Alloy performs *scope-bounded analysis* and the Alloy analyzer translates Alloy formulas into propositional logic with respect to a given bound on the universe of discourse. Alloy's intuitively familiar notation and efficient analysis make it a particularly attractive method to aid the development of complex systems. Alloy has been applied in many domains, including mobile computing, security, access control, software checking, and hardware vulnerability analysis [8]–[15].

This paper introduces the use of Alloy for specifying desired properties of machine learning models and creating models that guarantee those properties. Our key insight is that a slight enhancement of traditional boolean gates, which introduces gates that directly correspond to Alloy constructs, can provide an effective intermediate representation that not only simplifies translation from Alloy to (enhanced) boolean formulas but also readily translates to neural networks, thereby enabling a *faithful* translation from Alloy to neural networks. To illustrate, consider the Alloy keyword "`lone`" that applies to an expression, say e, and creates a formula that is true if and only if e represents either the empty set or a singleton set. We introduce a special *lone* gate that directly encodes this behavior rather than using a circuit of traditional boolean gates (and, or, and not).

To translate the enhanced gates to neural networks, we employ Rosette [16], a state-of-the-art program synthesis framework, which allows us to find *minimal* neural networks that are functionally equivalent to the gates. The translation of Alloy specifications then follows the standard translation to boolean logic with the exception of utilizing the enhanced boolean gates, followed by a translation to neural networks. In addition to employing Rosette for synthesis, we also utilize it for validating the implementation of our technique. Specifically, we formulate and check program verification problems to express the logical equivalence of Alloy formulas (written using Rosette's Alloy DSL) and the neural networks translated by our technique.

We experimentally evaluate our approach using a variety of relational specifications as subjects. The experimental results

This work was partially supported by NSF grant no. CCF-1718903.

2332-6549/20/$31.00 ©2020 IEEE
DOI 10.1109/ISSRE5003.2020.00024

Authorized licensed use limited to: University of Texas at Austin. Downloaded on September 03,2023 at 01:45:32 UTC from IEEE Xplore. Restrictions apply.

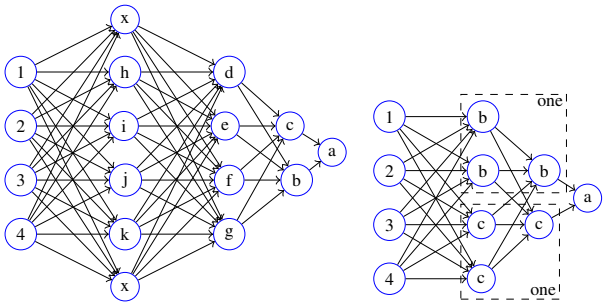
show that our approach allows synthesis of neural networks that are hard to create using traditional training methods.

Why synthesize neural networks? Supervised learning constructs models using labeled data. In contrast, we focus on using specifications to create such models without relying on datasets. We do not expect synthesis to be a replacement for training. Instead, we expect synthesis to complement training, and for the two to apply in synergy. Our overarching goal is to overcome two key issues with the traditional approach of training deep learning models: 1) lack of verifiability; and 2) lack of explainability. Addressing these challenges will enable deployment of machine learning models that can be trusted and have behaviors that can be explained using human understandable or machine verifiable logical reasoning. We believe our work provides a viable direction to achieve this goal. We envision a number of promising techniques in future work, including: 1) building composite models that consist of components that are synthesized from specifications, *and* components that are trained using datasets; 2) using models synthesized from specifications as seeds for training more sophisticated models that guarantee certain base properties; and 3) using advances in machine learning, e.g., model compression methods [17], to provide highly efficient execution of specifications, say for runtime verification of a software system.

Our work also takes inspiration from a recent study [18] that shows that relational specifications form a surprisingly challenging target for training traditional models. The study showed that off-the-shelf models achieve high performance (accuracy, precision etc.) in learning relational properties when evaluated in a traditional way (i.e., on test datasets), but the *true* performance of these models when evaluated on the *whole* (bounded) input space was much poorer, thereby indicating the difficulty in learning these properties using the traditional approach.

This paper makes the following contributions.

- **Synthesis of minimal neural networks.** We form program synthesis problems in Rosette to create minimal neural networks that are functionally equivalent to the boolean gates, including enhanced gates that we introduce to directly support Alloy constructs.
- **Translation of Alloy to neural networks.** We define an end-to-end translation that takes as input a specification written in the Alloy language and creates as output a neural network that is logically equivalent to the input specification and executable on modern machine learning frameworks.
- **Verification of neural networks.** We form program verification problems in Rosette that allow checking the equivalence between Alloy formulas written using the Alloy embedding in Rosette, and neural networks that are also encoded in Rosette. Doing so allows validating our translation. The verification process also allows checking neural networks that were created in other ways, e.g., using the training approach or training based on datasets.



(a) Without ***one*** gate (b) Using ***one*** gate

Fig. 1: Fully connected feed-forward networks with *perceptron* activations for the $Function$ predicate.

- **Evaluation.** We perform an experimental evaluation to answer key research questions. The experimental results show that (1) enhanced gates enable a substantial reduction in the size of the synthesized neural networks; (2) Rosette provides an effective framework for synthesis of minimal neural networks that are equivalent to (enhanced) boolean gates; and (3) Rosette also provides effective verification of neural networks that represent complex relational specifications.

II. Illustration

This section presents an example to illustrate our technique for creating neural networks from relational specifications. We also provide the necessary background on Alloy and Rosette.

Background. The following Alloy specification defines the *function* property, i.e., the binary relation r is a *total* function:

```
sig S { r: set S }

pred Function() { all s: S | one s.r }

run Function for exactly 2 S
```

The *sig* S declares a set of atoms, and its *field* r declares a binary relation of type $S \times S$. The keyword *pred* defines a predicate, which is a parameterized formula that can be invoked elsewhere. The keyword all is universal quantification, and the operator "." operator is *relational composition*. The quantification operator one requires its expression to represent a singleton set. Thus, the body of the predicate $Function$ specifies that for each atom s in sig S, the relational image of s under r has exactly one element. Hence, r is a total function. The *run* command represents a typical usage of the Alloy tool: to solve constraints. Specifically, the run command instructs the Alloy analyzer to find an *instance*, i.e., a valuation of the sets and relations in the model such that the predicate $Function$ is true. The command specifies a *scope* that bounds the universe of discourse. The scope of "$exactly\ 2\ S$" requires that each instance must have 2 atoms in sig S, e.g., the following valuation is an instance for this scope:

```
S = {S0, S1}
r = {S0->S1, S1->S1}
```

where S contains the atoms $S0$ and $S1$, and r maps each of $S0$ and $S1$ to $S1$.

We next illustrate two frameworks that enable translation of Alloy formulas to boolean logic and provide key enabling

Authorized licensed use limited to: University of Texas at Austin. Downloaded on September 03,2023 at 01:45:32 UTC from IEEE Xplore. Restrictions apply.

technologies for our approach: 1) Kodkod [19], which is the standard tool integrated with the Alloy analyzer for solving constraints using SAT solvers [6], [7]; and 2) Rosette [16], which provides the Ocelot [20] library that implements an embedding of Alloy, and allows solving constraints using SMT solvers [21]. Kodkod and Rosette create the following two formulas corresponding to the predicate $Function$ (w.r.t. scope of "$exactly\ 2\ S$"):

```
Kodkod: (((1|2)&(!2|!1))&((3|4)&(!4|!3)))
Ocelot: (&& (&& (|| (! r$0) (! r$1))
        (|| r$0 (&& r$1 (! r$0))))
        (&& (|| (! r$2) (! r$3))
        (|| r$2 (&& r$3 (! r$2)))))
```

Both formulas use 4 boolean variables, which represent the 2×2 boolean matrix that encodes the relation r. For example, the Kodkod variables 1, 2, 3, 4 respectively represent the entries (0, 0), (0, 1), (1, 0), and (1, 1) of the matrix. While the two formulas created by Kodkod and Ocelot are semantically equivalent, they are structurally different (modulo differences in notation). As Alloy's specialized backend, Kodkod performs several optimizations, which have an important impact on the size of the boolean formulas that it creates (see Appendix A).

Translation to Neural Networks. We next illustrate how our approach translates Alloy specifications to neural networks that are correct by construction. Kodkod/Ocelot already provide vehicles to translate Alloy to boolean logic. We enhance this translation to introduce special gates that directly correspond to *multiplicity* constructs in the Alloy models, such as keywords $lone$ and one, which can both quantify expressions and also define quantified formulas. Moreover, we also define a translation from boolean logic with enhanced gates to neural networks. Overall, our approach has three primary components: 1) synthesis of minimal neural networks that are equivalent to (enhanced) boolean gates (using Rosette); 2) translation of (enhanced) boolean logic to *fully connected*, *feed-forward* neural networks (using Kodkod); and 3) verification of the neural networks with respect to relational specifications (using Rosette).

Figure 1a shows the resulting network generated by translating the Kodkod formula for the $Function$ predicate for the scope of $exactly\ 2\ S$. The network comprises of 4 layers in addition to an input layer. The neurons marked 1 to 4 are the four inputs to the network. Units assigned letters a to k represent the various logical operations performed by the neurons. These letters correspond to superscript letters following the logical symbols in the annotated Kodkod formula:

$$(((1|^{d}2)\&^{b}(!^{h}2|^{e}!^{i}1))\&^{a}((3|^{f}4)\&^{c}(!^{j}4|^{g}!^{k}3)))$$

Since we use the feed-forward architecture, there are no *skip* connections between the layers. The buffer neurons (marked x) are used when output of a neuron needs to be forwarded to a non-immediate layer. The connection weights and bias values in the network are configured using the values of the corresponding logic gates synthesized with Rosette. Any connections that are not required have the corresponding weights set to zero.

1) Enhanced Neural Gates: We next illustrate our enhanced translation that provides a richer set of neural gates, which results in a more compact encoding and leads to the creation of smaller, more efficient networks. We augment the Kodkod boolean formula generator to add support for `one` and `lone` gates, which correspond to how the keywords are defined by Alloy. Observe the `function` predicate uses the keyword `one`. The boolean expression (w.r.t to same scope of 2) built using the enhanced `one` gates is "$((1@^{b}2)\&^{a}(3@^{c}4))$", where the one gate is identified using the symbol "@". Figure 1b shows the network with `one` gates. Compared to the original, this network encodes the same formula with one less layer and 6 fewer neurons. The neurons are labeled by letters depicting operation they are performing (e.g., a is the `and` operation as depicted in the formula). The `one` gate implementation requires three neurons. The values of the various network parameters are derived from the corresponding synthesis problems solved using Rosette. The advantage of using these enhanced gates become more substantial for larger scopes. For a scope of 10, using the `one` gate for the $Function$ predicate results in a network that has a third of the number of layers, 93% fewer connections, and is 20 times faster in making predictions in comparison with the network that does not use the enhanced gates.

III. Technique

This section presents our technique. First, we explain how we synthesize neural networks that represent (enhanced) boolean logic gates using program synthesis methods. Specifically, we synthesize networks that are minimal in terms of the number of layers and neurons. These neural networks are used as building blocks in our technique to translate formulas written in Alloy into equivalent fully connected feed-forward neural networks.

A. Synthesis of logic gates as neural networks

At an abstract level the synthesis problem has two key elements: 1) specification that provides reference behavior; 2) skeletal implementation that provides a template that needs to be completed by the synthesizer. In our case the specification describes the behavior of the desired boolean gate (that has 1+ boolean inputs and 1 boolean output), the skeletal implementation describes the fixed structure of a fully connected feed-forward neural network, and the synthesizer outputs weights and biases on the edges and neurons that are part of the skeleton. We fix the range of values weights can take to $\{-1, 0, +1\}$, and the range of values biases can take to $\{-n, -n+1, ..., n-1, +n\}$, n being the number of inputs to the gate. To find a *minimal* network, we start with no hidden layers, and iteratively increase the number of hidden layers and neurons per layer until the synthesizer solves the problem.

1) Neural network template: Neurons in every layer compute a weighted sum over inputs received from neurons in the previous layer, with a bias term added to the sum. The resulting value is then propagated through an activation function to generate the neuron output. We use the well-known *perceptron* [22] activation for each neuron; the following equation computes the output of a neuron; w and x are vectors of

Authorized licensed use limited to: University of Texas at Austin. Downloaded on September 03,2023 at 01:45:32 UTC from IEEE Xplore. Restrictions apply.

weights and inputs, and b is the bias where the size of the vectors is equal to the number of inputs to the neuron:

$$f(x) = \begin{cases} 1 & w.x + b > 0 \\ 0 & otherwise \end{cases}$$

The synthesis problem searches for values of the *weights* (w) and *bias* (b) terms – indeed, these are the *learnable* parameters inferred during the training procedure of traditional machine learning methods for training neural networks.

2) Problem formulation: In the synthesis-based approach, these learnable parameters are represented as symbolic values that are solved for with respect to the given specification (without using any datasets). For a given network shape, the forward pass is encoded as a formula which represents computations performed by the network. Given a formula that uses the symbolic values, for a given test input, the task of training gets transformed into the task of searching for concrete bindings of the symbolic values such that the network produces the desired output. SMT solvers efficiently accomplish this search. The expected behavior of the network is encoded as constraints, which are then provided to the SMT solver. Let $N_{n,S,W,B}$ represent a fully-connected feed-forward network comprised of n layers. The parameter S represents the shape/topology of the network, a mapping from the layer to the neurons in that layer. W is the set of all connection weights, while B is the set of bias values of the neurons in the network. ϕ_{spec} represents the semantics of the boolean operation, for which we want to synthesize a network. For a network of fixed size and shape, the synthesis problem reduces to determining the values of W and B such that the following equation is satisfied.

$$\exists W, B. \forall i \; N_{n,S,W,B}(i) = \phi_{spec}(i)$$

In words, the equation asks if there exists a set of weights and biases, such that the output of the network equals the output of the specification for the entire input space. We employ the well-known *counter-example guided inductive synthesis* (CEGIS) framework to handle the quantifier alternation in the equation [23], [24]. A typical CEGIS setup requires two components - a *synthesizer* that generates candidate programs (neural networks in our case) and a *verifier* that checks the candidates against a specification. To generate *better* candidates, the synthesizer utilizes counter-examples provided by the verifier.

3) Synthesis Loop: Algorithm 1 shows the synthesis loop. It has the following inputs:

- Number of inputs to the logic gate.
- The specification (*spec*) is a piece of code that captures the behavior of the logic gate.
- The shape of the network is represented by the structure *config*, which stores and manages information about the layers, neurons in each layer, and their connections.

The output of the algorithm (*parms*) is a set of parameters (weights and biases) for the given neural network configuration (*config*). The network with these parameters exhibits behavior equivalent to the specification (*spec*).

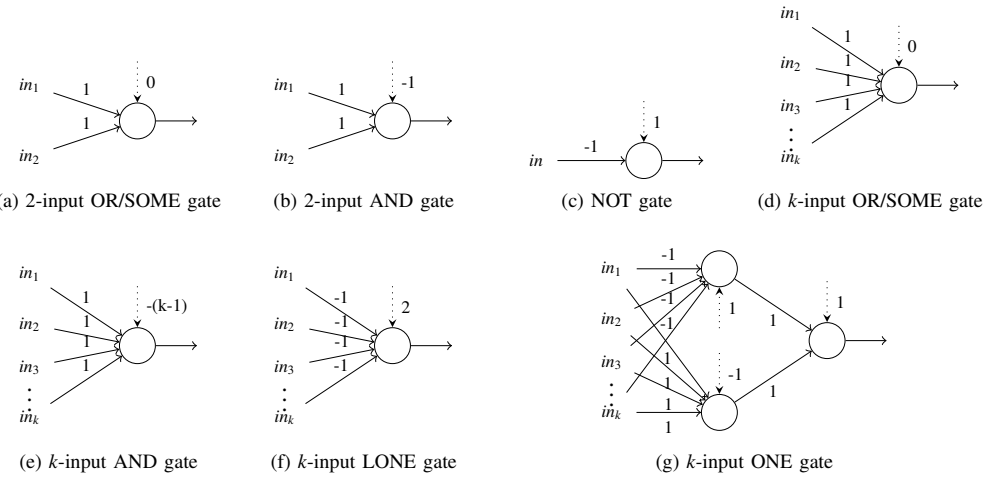
Algorithm 1 Synthesis loop

Input: num_input, config, spec
Output: parms

```
1:  for layer in config do
2:    for neuron in layer do
3:      for input in neuron.inputs do
4:        input.weight ← create_symbolic_const()
5:        addConstraint(input.weight ≤ 1)
6:        addConstraint(input.weight ≥ -1)
7:      end for
8:      neuron.bias ← create_symbolic_const()
9:      addConstraint(neuron.bias ≤ num_input)
10:     addConstraint(neuron.bias ≥ -num_input)
11:   end for
12: end for
13: sParms ← config.Symbolics
14: parms ← initialize(config)
15: for i in num_input do
16:   input[i] ← create_symbolic_const()
17:   addConstraint(input[i]==1 || input[i]==0)
18: end for
19: while true do
    ▷ Search for a counter-example
20:   model ← verify(spec(input) ⇔ net(config,parms,input))
21:   if model is not null then                    ▷ counter-example found
    ▷ Add a new constraint with the counter-example
22:     addConstraint(spec(model)==net(config,sParms,model))
23:     m ← solveInc()
24:     if m is not null then
25:       parms ← m                                  ▷ m is new solution
26:     else
27:       problem UNSAT                ▷ no solution exists for the config
28:       break
29:     end if
30:   else                                          ▷ no counter-example found
31:     problem SAT                                  ▷ parms is the solution
32:     break
33:   end if
34: end while
```

The algorithm begins by reading the configuration and initializing symbolic constants for the network parameters. The *config* allows for reading the network layer-by-layer and iterating over the neurons in each layer (lines 1,2). For every input feeding to a neuron, we create a symbolic *weight* value (*create_symbolic_const*) (lines 3,4). In our implementation, the symbolic weights can only be bound to integer values, which range from -1 to 1 (lines 4-6). The function *add_constraint* takes as input an assertion condition and adds it to the global *assertion store*. Similar to the input *weights*, we create symbolic *bias* integers whose values are restricted to range between the number of inputs to the gate and its negation (lines 8-10). Allowing the symbolic weights and biases to have integer bindings and restricting their values to a finite range speeds up the synthesis process and makes it scalable. *sParms* is the set of all symbolic parameters created from *config* (line 13). Before starting the synthesis loop, we initialize the first candidate set of concrete parameters that are randomly generated by *initialize* (line 14) (*parms* is the concrete-valued counterpart of *sParms*). These values in *parms* adhere to the bounds set for their symbolic counterparts. We also define a symbolic input array(*input*), which represents the space of all possible inputs to the neural network. The size of the array is

Authorized licensed use limited to: University of Texas at Austin. Downloaded on September 03,2023 at 01:45:32 UTC from IEEE Xplore. Restrictions apply.



(a) 2-input OR/SOME gate (b) 2-input AND gate (c) NOT gate (d) k-input OR/SOME gate

(e) k-input AND gate (f) k-input LONE gate (g) k-input ONE gate

Fig. 2: Sample neural networks synthesized for various logical gates

equal to the number of inputs, and each element of the array is restricted to a value of 0 or 1 (lines 15-18). The synthesis loop starts with a call to the *verify* function (line 20). This function takes as input an assertion condition and searches for concrete bindings of all the symbolic constants that violate the input assertion while satisfying the assertions in the *assertion store*. In our context, we use *verify* to determine if there is an input that results in different outputs for the specification (*spec*) and the current network candidate (*parms*). The function *spec* produces the output of the specification for the given *input*. The function *net* creates a formula that represents the operations performed in a forward pass of a neural network, whose shape and structure are defined by *config*, and the values of the parameters are set to those in *parms*. It then computes the output of the network for a given input. Verify fails to return a model, if no such binding exists, and the current candidate matches the output of the *spec* on every possible binary-valued input. The synthesis loop completes, and *parms* contains the desired values for the network parameters (lines 30-32). On the other hand, if such a binding exists, verify returns a counter-example - a concrete input that produces different outputs on the spec and the current candidate. Using the new *counter-example*, we add a constraint, which ensures any subsequent candidates agree with the spec on the output for this input (line 22). The next candidate is generated by issuing a call to the *solveInc* function, which invokes the underlying solver to try and incrementally satisfy all the assertions in the global *assertion store* (line 23). If a model(m) is found, it produces new concrete bindings for *sParm*. It becomes the current candidate (line 25) and the search progresses to look for another counter-example. If the solver is unable to satisfy the assertions, the current network configuration, which includes the network shape and bound on the parameters, is unable to replicate the behavior of the *spec*. The synthesis is unsuccessful in this case (lines 26-27). Not shown in the algorithm is an enumerator that uses iterative deepening to generate network configurations (*config*) of increasing size and complexity. The first network the enumerator generates has no hidden layers (the inputs directly feed into an output neuron). The next network has a single hidden layer with one neuron, and then a single layer with two neurons and so on. Once the number of neurons in the hidden layer reaches a threshold (currently set to the number of inputs), another layer is added to the network. The synthesis loop is called with configurations of increasing size and complexity, and the process continues until a feasible candidate is found. This ensures that we synthesize the smallest possible neural network, given our bounds on the *weight* and *biases*. Fig. 3 implements the proposed algorithm to synthesize a three input `lone` gate in Rosette.

Verification of networks: We synthesize networks (up to 10 inputs) that are the smallest possible implementing the desired operation. We observe patterns in the weights and bias values of the synthesized networks based on which we manually design larger networks (larger number of inputs). Then, we use Rosette to verify correctness of these manually constructed networks. While synthesis of large networks is prohibitive due to the computational complexity, our extrapolation approach improves the scalability and allows us to create larger networks. Fig. 2 depicts the various neural gates synthesized using our technique.

B. From Kodkod to neural networks

This section describes the method to translate Alloy formulas into equivalent neural networks. Our translation system is built using Kodkod, a SAT solver based framework written in Java, for first-order logic relations. Alloy possesses a Kodkod embedding, which enables the translation of Alloy formulas into the Kodkod format, making use of its API. We utilize the Kodkod front-end that coverts the first-order relation into a boolean formula and constructs an *abstract syntax tree* (AST). We use the AST to perform two tasks. First, using the *visitor design pattern*, we implement a custom translator which uses the enhanced boolean gates (`one` and `lone`) when generating

Authorized licensed use limited to: University of Texas at Austin. Downloaded on September 03,2023 at 01:45:32 UTC from IEEE Xplore. Restrictions apply.

```
#lang rosette
(current-bitwidth 10) ;reasoning bit-width
(define numInputs 3)
(define inc (solve+)) ;incremental solver
(define wt_upper_bound 1) (define wt_lower_bound -1)
(define bias_upper_bound 3) (define bias_lower_bound -3)
(define (spec inputs) ;specification
  (define cnt 0)
  (for ([i inputs])
    (if (equal? i 1) (set! cnt (add1 cnt)) '()))
  (if (< cnt 2) 1 0))
(define (net input wts bias) ;network
  (define out0 (+ (* (list-ref wts 0) (list-ref input 0))
                  (* (list-ref wts 1) (list-ref input 1))
                  (* (list-ref wts 2) (list-ref input 2))))
  (if (> (+ out0 bias) 0) 1 0))
(define (create_symbolic_wts n)
  (for/list ([i (in-range n)]) ;Algo1. line 3
    (define-symbolic* w integer?) ;Algo1. line 4
    (inc (and (>= w wt_lower_bound) (<= w
        wt_upper_bound))) w)) ;Algo1. lines 5,6
(define sym_wts (create_symbolic_wts numInputs)) ;Algo1.
    line 13
(define-symbolic* sym_b integer?) ;Algo1. line 8,13
(inc (and (>= sym_b bias_lower_bound) (<= sym_b
    bias_upper_bound))) ;Algo1. line 9,10
(define wts (list 0 0 0)) (define b (list 0)) ;line 14
(define inConstraints #t)
(define symIn
  (for/list ([x (in-range numInputs)]) ;Algo1. line 15
    (define-symbolic* h integer?) ;Algo1. line 16
    (set! inConstraints (and (or (= h 1) (= h 0))
        inConstraints)) h)) ;Algo1. line 17
(for ([cnt (in-naturals)]) ;Algo1. line 19 (infinite
    loop)
  (define model (verify #:assume (assert inConstraints)
    #:guarantee (assert (= (spec symIn) (net symIn wts
        b))))) ;Algo1. line 20
  (when (unsat? model) (display "Success")) ;Algo1. line
      31
  #:break (unsat? model) ;Algo1. line 32
  (define newTest (evaluate symIn model)) ;newTest ->
      counter-example
  (define condT (equal? (net newTest sym_wts sym_b)
      (spec newTest))) ;Algo1. line 22
  (define m (inc condT)) ;Algo1. line 23
  (when (unsat? m) (display "Fail")) ;Algo1. line 27
  #:break (unsat? m) ;Algo1. line 28
  (set! wts (evaluate sym_wts m)) ;Algo1. line 25
  (set! b (evaluate sym_b m))) ;Algo1. line 25
```

Fig. 3: Synthesizing a 3-input *lone* gate

a boolean formula. With this new translator, the multiplicity operators, instead of being interpreted using the fundamental boolean gates, are now represented with the enhanced custom gates. Secondly, we use another *visitor* to traverse the AST and generate neural network equivalents of the logical operations being performed at each node. The networks are connected using the parent-child relations in the AST to output a single neural network that is equivalent to the boolean formula. The following paragraphs describe the implementation of the latter. The first visitor is implemented in a similar fashion.

1) Top-Level Translator: The figure below shows a code snippet of the primary translator class called *NeuralNetTranslator*, which implements the *GateVisitor* interface. The interface specifies visit methods for four types of nodes in the AST. A *BooleanVariable* represents the primary variables used as inputs to the formula. *NaryGate* and *BinaryGate* classes represent multi-input and two-input logical gates, respectively, while *NotGate* represents a not gate. All the four classes extend the abstract class *BooleanFormula*. The neural network translator is invoked by calling the translate function and providing it the root node of the AST as an argument. The visit methods first visits the children (if present) followed by the addition of the node to the neural network. The translator object upon instantiation creates a new neural network object whose details are described next.

```
class NeuralNetTranslator implements GateVisitor {
  Network net;

  void translate(BooleanFormula o) {
    //creat a new network
    net = new Network(numInputs);
    o.accept(this); }

  //visiting a boolean variable
  void visit(BooleanVariable var) {
    net.addToNet(var); } // add to the network

  //visiting a nary gate
  void visit(NaryGate gate) {
    Iterator<BooleanFormula> it = gate.iterator();
    while (it.hasNext()) {
      //visiting the children of the nary gate
      BooleanFormula b = it.next();
      b.accept(this);
    }
    net.addToNet(gate); } // add to the network

  //...
}
```

2) Neural network implementation: A neural network is composed of three different types of objects. The class *Neuron* represents a single neuron characterized by its position in a network layer, a *bias* value, the depth of the layer where this neuron is located, and a reference to the node in the AST of the original formula, which this neuron represents. *Layer* represents a layer of neurons in the network. The *Connection* class is used to store connections between two neurons in the network. It holds a reference to the source and destination of the connection and an integer weight.

```
class Neuron {
  int id; //position in the layer
  int bias;
  int depth; //depth of the layer containing this neuron
  BooleanFormula orig; //reference to the ast node
  //...
}
public class Layer {
  int numUnits; //number of neurons in the layer
  ArrayList<Neuron> units;
  //...
}
class Connection { //represents the connection "from"->"to"
  Neuron to;
  Neuron from;
  int weight; //connection weight
  //...
}
```

The *Network* class shown below represents the entire network. An object of this class is instantiated by *NeuralNetTranslator*. *Network* maintains a record of every layer in the network, connections between neurons, and their relationship to the nodes in the boolean formula AST. It provides the functionality to add logical gates to the network by using the appropriate networks synthesized with Rosette (discussed in III-A). Upon instantiation, the network creates an input layer whose size is equal to the number of inputs supplied to the network.

```
class Network {
  int numInputs; //inputs to the network
```


Authorized licensed use limited to: University of Texas at Austin. Downloaded on September 03,2023 at 01:45:32 UTC from IEEE Xplore. Restrictions apply.

```
3  ArrayList<Layer> net; //layers in the network
4  ArrayList<Connection> connections;
5  Map<BooleanFormula, Neuron> nMap;
6
7  Network(int numInputs) {
8    //...
9    net.add(new Layer()); //adding the input layer
10   for (int x = 0; x < numInputs; x++) {
11     net.get(0).addToLayer(new Neuron(0, x, 0, null));
12   } }
13
14 void addToNet(BooleanVariable var) {
15   net.get(0).setInput(var);
16   nMap.put(var, (net.get(0)).getInput(var.label() - 1)); }
17
18 void addToNet(NaryGate gate) {
19   Neuron output = new Neuron(0, 0, -1, gate);
20   ArrayList<Neuron> inputs = new ArrayList<Neuron>();
21   Iterator<BooleanFormula> it = gate.iterator();
22   while(it.hasNext()) {
23     BooleanFormula b = it.next();
24     Neuron in = nMap.get(b);
25     inputs.add(in);
26   }
27   output.depth = maxDepth+1;
28   resolveConnections(output, inputs);
29   addNeuron(output); }
30 //...
31 void addNeuron(Neuron n) {
32   if (n.depth >= net.size()) {
33     net.add(new Layer());
34     depth++;
35   }
36   net.get(n.depth).addToLayer(n);
37   if(n.orig != null) {
38     nMap.put(n.orig, n);
39   } }
40 //...
41 }
```

Adding to the network: The addition of AST nodes to the neural network is performed via a call to the *addToNet* function, which is overloaded to handle the four types of nodes. *BooleanVariable* represents an input to the network, and a call to *addToNet* in this case, simply adds the variable to the input layer at the appropriate location. Boolean variables are given unique integer labels ranging from 1 to the number of inputs. These labels are used to determine the location of the neuron in the input layer. The *addToNet* for a *NaryGate* first creates a new neuron which represents the output of the gate. Network maintains a mapping (*nMap*) of AST nodes and their corresponding neurons. The neurons representing the inputs to this gate are collected and passed to the *connect* function along with the output neuron. The connections between the input neurons and output neurons are made within *connect*. Finally, the output neuron is added to the network. Binary gates and Not gates are added using the same procedure as Nary gates.
Making connections: The *connect* method takes as input the output neuron and a list of input neurons. It uses the structure of the neural networks synthesized using Rosette to determine the values of various network parameters. For example, when connecting the input and output neurons to construct an *and* network, we know that the smallest network will have no hidden layers. The weights of each connection between an input neuron and the output neuron should be *+1*, and the output neuron will have a bias of *-(n-1)*, where *n* equals the number of inputs. The construction of `one` gate requires one hidden layer with two neurons. The weights and bias values shown in the code snippet are derived from the corresponding Rosette model. The rest of the gates representing various logical operations and multiplicities are handled similarly.

```
1 void connect(Neuron n, ArrayList<Neuron> inputs) {
2   if (n.orig.op().ordinal() == 0) { // AND Gate
3     n.bias = -1 * (inputs.size() - 1);
4     for (Neuron in : inputs) {
5         Connection c = new Connection(n, in, 1);
6         connections.add(c);
7     }
8   }
9   else if (n.orig.op().ordinal() == 9) { //ONE Gate
10    //requires a hidden layer with two neurons
11    int depth = inputs.get(0).depth+1;
12    Neuron h1 = new Neuron(-1, 0, depth, null);
13    Neuron h2 = new Neuron(1, 0, depth, null);
14    n.bias = 1;
15    n.depth = h1.depth+1;
16    addNeuron(h1);
17    addNeuron(h2);
18    connections.add(new Connection(n, h1, -1));
19    connections.add(new Connection(n, h2, -1));
20    for(Neuron in:inputs) {
21      connections.add(new Connection(h1, in, 1));
22      connections.add(new Connection(h2, in, -1));
23    }
24  }
25  //...
26 }
```

C. Validation of networks

```
1 #lang rosette
2 (require ocelot)
3 (define num_inputs 9)
4 (define U (universe '(n1 n2 n3)))
5 (define node (declare-relation 1 "node"))
6 (define rel (declare-relation 2 "reflexive"))
7 (define bound_node (make-exact-bound node '((n1) (n2)
      (n3))))
8 (define bound_rel (make-product-bound rel '(n1 n2 n3) '(n1
      n2 n3)))
9 (define all_bounds (bounds U (list bound_rel bound_node)))
10 (define iB (instantiate-bounds all_bounds))
11 (define reflexive_rel (all ([n node]) (in (-> n n) rel)))
12 (define reflexive (interpret* reflexive_rel iB))
13 (define (bool2int inputs)
14   (for/list ([i (in-range num_inputs)])
15     (if (equal? (list-ref inputs i) #t) 1 0)))
16 (define (int2bool in)
17   (if (= in 0) #f #t))
18 (define (network in)
19   (define weights (list 1 0 0 0 1 0 0 0 1))
20   (define bias (list -2))
21   (define parms (list weights bias))
22   (define out0 (+
23     (* (list-ref (list-ref parms 0) 0) (list-ref in 0))
24     (* (list-ref (list-ref parms 0) 1) (list-ref in 1))
25     (* (list-ref (list-ref parms 0) 2) (list-ref in 2))
26     (* (list-ref (list-ref parms 0) 3) (list-ref in 3))
27     (* (list-ref (list-ref parms 0) 4) (list-ref in 4))
28     (* (list-ref (list-ref parms 0) 5) (list-ref in 5))
29     (* (list-ref (list-ref parms 0) 6) (list-ref in 6))
30     (* (list-ref (list-ref parms 0) 7) (list-ref in 7))
31     (* (list-ref (list-ref parms 0) 8) (list-ref in 8))))
32   (if (> (+ out0 (list-ref (list-ref parms 1) 0)) 0) 1 0))
33 (define allS (symbolics iB))
34 (define model (verify #:guarantee (assert (equal?
      reflexive (int2bool (network (bool2int allS)))))))
```

Fig. 4: Verifying a scope-3 *reflexive* network

The final step in the process is to validate the generated networks. While the networks are *correct by construction*, formally verifying them assists in revealing any potential issues in the translation framework. The validation approach can also be used for verifying networks created by traditional training techniques. We use Rosette's *verify* functionality to establish the equivalence of a neural network with the relational property it is supposed to represent. Using Ocelot, we

Authorized licensed use limited to: University of Texas at Austin. Downloaded on September 03,2023 at 01:45:32 UTC from IEEE Xplore. Restrictions apply.

create a specification of the relational property against which the network needs to be checked. Also, we create a network function that carries out the neural network computation on a given input using the provided set of weight and bias parameters. The size and shape of the network, and the parameter values are determined by the Kodkod translation we carried out in the previous step. With a model of the specification and the network, we invoke the *verify* feature to try to discover a counter-example. The absence of a counter-example proves the equivalence. Using this methodology, we verify neural networks for all the properties studied in this paper for a scope bound of 10, except for the 4 linked list properties, which are validated for a scope bound of 5. Fig. 4 shows the sample code for verifying the *reflexive* property. A reflexive relation over some set is one that relates every element in the set to itself. In the code, we verify reflexivity for a scope bound of 3. A binary relation with a scope bound of 3 has 9 inputs, one for each ordered pair. We define the *universe of discourse* (line 4), declare the relations (line 5,6), and define their bounds (lines 7-10). We specify the reflexive property (`reflexive_rel`) and *interpret* it to generate the boolean formula (`reflexive`), which encodes the property for the given scope (line 11, 12). The function `network` represents a neural network with 9 inputs, no hidden layers, and 1 output neuron. It has 9 weights (one for each input) and a bias term, whose values are determined by the translator (lines 18, 21). The output of the function indicates whether the *reflexive* property holds for the given input. The variable `allS` is the list of boolean inputs used in the formula and provided to the `network` (`bool2int` and `int2bool` are utilities to convert boolean values into integers and vice-versa). We use `verify` to check the equivalence of `reflexive` and `network`.

IV. Evaluation

A. Subjects

1) Relational properties: We apply our technique to 17 relational properties expressed in Alloy and adapted into the input format of Kodkod and Rosette. Out of the 17, 8 are *base* properties – *reflexive, irreflexive, symmetric, anti-symmetric, transitive, connex, functional, function*. The remaining properties are *compositional*, described using two or more base properties. They include *equivalence, bijective, injective, surjective, strict order, non-strict order, partial order, pre-order and total order* relationships. For example, a *total order* relationship is one that is *transitive*, *anti-symmetric* and *connex*. These properties were used in a recent study on learnability of relational specifications [18].

2) Linked List: We also model in Alloy two properties of linked lists – *cyclicity* and *acyclicity*. These subjects are more complex than the relational properties and use *transitive closures* and multiplicities as quantifiers. This allows us to evaluate our technique in a more complex context.

B. Methodology

Our experimental methodology aims to evaluate our translation scheme and the efficiency of the synthesized neural networks. We import the relational properties written in Alloy into the Kodkod environment to generate the boolean formulas. We use two scope bounds, 5 and 10, which represent a state-space of 2^{25} and 2^{100}, respectively. We translate each formula to a neural network. The networks are then exported into *Keras* [25], a machine learning library that runs of top of the *Tensorflow* framework. We study the inference times of these networks in both *online* and *batch* inference settings using 10000 test inputs. Batch or offline inference is used when the application does not have stringent latency requirements. The system waits for a certain number of prediction requests to accrue and then performs the inference on the entire batch as one unit of data. This amortizes the overhead of moving data to and from memory, resulting in high throughput. We use a batch-size of 1000. For applications that demand fast response times, online inference is more suited. In this situation, a prediction is generated for a request as soon as the request is made (batch-size is 1). All the experiments were carried out on a 4 core Intel(R) Core(TM) i5-6200U CPU with 4GB of memory, running Ubuntu 16.04.

C. Research Questions

We answer the following research questions:
RQ1. What impact does introducing enhanced `one` and `lone` gates have on the size of the efficiency of the networks?
RQ2. How do the neural network sizes scale when increasing the scope from 5 to 10?
RQ3. What is the impact of using multiplicity operators as quantifiers?

Table I shows the topology and performance of neural networks for the 17 properties generated using Kodkod. As mentioned previously, we generate networks for two scope bounds of 5 and 10. Table II presents the results of networks generated from *cyclic* and *acyclic* linked list properties. In this case, we only use scope 5 to construct the networks. The shape of the network is quantified using two parameters - the number of layers in the network and the number of connections. We present the latter as a ratio of the number of connections having non-zero weights to the total number of connections in a fully-connected setting. The performance of the networks is quantified by the batch (batch-size 1000) and online (batch-size 1) inference times using a corpus of 10000 tests that are generated at random. By measuring the total inference time of all the tests, we determine the average time taken to generate a prediction on a single input for both scenarios; note, under the batch mode, to process 1 input we must process 1000 inputs, so we report the time for the entire batch (average over 10 batches); under the online mode, we report the time per input (average over 10,000 inputs). We use Table I to study *RQ1* and *RQ2*, and Table II for *RQ3*.

1) RQ1. Impact of enhanced gates: Our translation uses neural networks that directly compute multiplicity operations (`lone` and `one`). 5 out of the 17 properties make use of these multiplicity operators. To measure the impact of adding the `lone` and `one` gates, we generate networks with and without these gates for the relevant subjects. In the table,

Authorized licensed use limited to: University of Texas at Austin. Downloaded on September 03,2023 at 01:45:32 UTC from IEEE Xplore. Restrictions apply.

Property	Scope 5				Scope 10			
	Neural net config.		Inference times(ms)		Neural net config.		Inference times(ms)	
	Layers	NZW/TC	Batch	Online	Layers	NZW/TC	Batch	Online
Antisymmetric	3	40/360	34	4	3	180/5670	29	5
Bijective	9	566/23894	50	6	14	5726/2155974	1284	18
*Bijective**	4	126/590	21	4	4	446/4170	26	4
Connexivity	2	40/390	19	4	2	155/5555	26	4
Equivalence	5	752/56497	61	6	5	6947/4653382	2714	22
Function	7	295/7705	34	5	12	3300/807210	507	10
*Function**	3	65/305	20	5	3	230/2210	25	5
Functional	7	245/6130	32	6	12	2740/554110	362	8
*Functional**	2	30/130	19	4	2	110/1010	21	4
Injective	8	282/6723	34	6	13	2863/562493	368	8
*Injective**	3	62/272	20	4	3	222/2042	24	5
Irreflexive	2	10/130	18	4	2	20/1010	20	4
Non-Strict Order	5	692/47787	56	6	5	6677/4297387	2503	21
Partial Order	5	683/47084	56	5	5	6663/4290664	2514	18
Pre Order	5	650/43151	54	4	5	6495/4105996	2395	19
Reflexive	1	5/25	17	4	1	10/100	19	4
Strict Order	5	654/43895	55	5	5	6504/4121485	2403	21
Surjective	8	282/6723	34	5	13	2862/562493	366	8
*Surjective**	3	62/272	20	4	3	222/2042	25	5
Symmetric	3	100/1820	21	4	3	450/34290	45	4
Total Order	6	728/50536	59	6	6	6823/4392756	2560	21
Transitive	4	640/42480	50	4	4	6480/4099410	2379	20

TABLE I: Size and performance of neural networks created using Kodkod
(*NZW* - Non-zero weights, *TC* - Total connections)

the name of the property followed by an *asterisk*(*) indicates that the network is generated using these enhanced gates. For properties that use the `lone` gates (*functional, injective, bijective, surjective*), the size of their network decreases by 5 layers for scope-5 and 10 layers for scope-10. The resulting networks are significantly smaller, with far fewer connections. For example, the *injective* relation for a scope of 10 requires only 222 connections using the `lone` gate as opposed to almost 3000 required without it. Smaller networks also result in reduced inference latencies. The scope 10 *bijective* relation, utilizing `lone` gates, witnesses a $50x$ and $4x$ speed-up for batch and online inferences, respectively. These speed-ups are more pronounced for the larger scope 10 networks than scope 5 networks. Using `one` gates reduces the network size of the *function* network by 4 layers for scope 5 and 9 layers for scope 10. The resulting network is $20x$ and $2.3x$ faster in batch and online inference settings (for a scope of 10). Another benefit of using these gates is that the number of layers in the network does not increase with the scope bounds. For example, going from a scope of 5 to a scope of 10 requires the same number of layers. However, the shape of the network does change with more neurons and connections in each layer for scope 10.

2) RQ2. Network sizes for different scopes: The subjects we study can be divided into two groups: properties that do not use *multiplicity* operators, and properties that use operators like `lone` and `one`. For the first group, the network layers remain the same when increasing the scope. For the latter, increasing the scope bounds increases the number of layers. For the five properties which use either `lone` or `one` operator, going from a scope of 5 to 10 adds 5 layers to the network.

Property	Neural net config.		Inference times(ms)	
	Layers	NZW/TC	Batch	Online
Acyclicity1	12	19520/12540365	7437	47
*Acyclicity1**	12	19330/12131670	7156	45
Acyclicity2	17	43162/60222914	35937	178
*Acyclicity2**	13	19080/11785352	6884	42
Cyclicity1	12	21143/14694170	8858	54
*Cyclicity1**	12	20953/14251475	8295	50
Cyclicity2	17	43307/60937359	35767	179
*Cyclicity2**	13	19135/11861212	6928	44

TABLE II: Size and performance of neural networks for Linked List properties
(*NZW* - Non-zero weights, *TC* - Total connections)

However, this can be solved by the use of *enhanced* gates. Increasing the scope changes the network shape, irrespective of whether layers are added or not, as there are more neurons and connections in the network.

3) RQ3. Multiplicity operators as quantifiers: The relational properties we have studied so far only used universal(`all`) quantification in their formulas, and the multiplicity operators were only used with expressions. However, operators like `lone` and `one` can also be used as quantifiers. The code below shows two ways of specifying the *acyclic* property of a linked list. The predicate `AcyclicSinglyLinked1` uses the universal quantification, while `AcyclicSinglyLinked2` uses both `all` and `one`; both the predicates are equivalent. The formulas also make use of *transitive*(^) and *reflexive-transitive*(*) closures. In this section, we study the impact of having multiplicity operators as quantifiers and also present the translation of formulas that involve closures of relations.

Authorized licensed use limited to: University of Texas at Austin. Downloaded on September 03,2023 at 01:45:32 UTC from IEEE Xplore. Restrictions apply.

```
one sig Header extends Node {}
sig Node {
   link: set Node
}
pred AcyclicSinglyLinked1() {
  all n: Header.*link {
    n !in n.^link
    lone n.link
  }
}
pred AcyclicSinglyLinked2() {
  all n: Header.*link | lone n.link
  one n: Header.*link | no n.link
}
```

We model two types of linked lists - *acyclic* and *cyclic*, for our analysis. Like the acyclic property, we specify the cyclic property in two ways - with and without the use of multiplicity operators as quantifiers. Table II provides the details of networks generated for the two types of list and their variations. All the networks correspond to a formula of scope 5. A suffix of 1 to the property name implies the absence of multiplicity operators as quantifiers, while a suffix of 2 implies the use of the `one` operator as a quantifier. An asterisk(*) indicates the use of enhanced multiplicity gates to generate the network. For formulas that do not contain multiplicities as quantifiers, the use of enhanced gates has minimal impact. We observe only a marginal decrease in the number of connections, while the number of layers remains the same. The inference times remain similar as well. These results show that the for such formulas reflexive-transitive closure is the primary determinant of the length and complexity of the formula. However, for predicates like `AcyclicSinglyLinked2`, which use a multiplicity quantifier over reflexive-transitive closure, we see a significant reduction in the size of the network. This is because now the translation of quantifier `one` can be optimized using the enhanced `one` gates. *Cyclicity* uses identical quantifiers, and we see a similar trend for this property.

V. Limitations

The properties of the networks we generated using our translation technique depend on how the specifications are written. As seen in our experiments with the linked list subjects, the same specification can be written in multiple ways leading to the generation of distinct networks that exhibit significant variation in performance. One way of addressing this issue is to determine the combinations of Alloy constructs that require complex boolean interpretations and try to avoid them as much as possible when creating the networks. Another approach to mitigate this issue is to create a richer and more sophisticated database of *enhanced* gates which learn to represent these complex constructs that bloat the boolean formulas. For example, like the `one` and `lone` gates, we are working to synthesize networks that preserve and represent expressions with *transitive closures*. This would allow us to compactly represent such expressions with such *closure* gates instead of interpreting them using standard logical operators.

VI. Related Work

Verification of ML models. LIME [26] is a technique that given a neural network and an input example, produces an explanation about the parts of the input which contributed mostly to the model's decision. While explanations of LIME can be useful, there's no guarantee that such explanations are correct. Viper [27] and MoET [28] are previous techniques used in deep reinforcement learning (RL), where an agent in a form of a neural network is mimicked to produce a tree-like model. Viper's decision tree model is then translated to an SMT formula to provide rich verification in RL setting. Our work is different in spirit in that it produces provably correct neural networks from the very beginning.

Program Synthesis of Neural Networks. Previous works have framed neural network training as a program synthesis task. In [29], the authors represented a neural network as a sketch where weights are symbolic values to be synthesized. While this work relies on a search procedure and SAT solving to find a solution, our technique is different in that it directly translates an Alloy formula to a neural network without any search or solving involved, except for the predefined gates.

Propositional Logic in Neural Networks. Minsky and Papert, in 1969, proved that the XOR function cannot be implemented via perceptron (single layer neural network) [30]. However, later work showed that the XOR function can be realized by the addition of hidden layers to the network. Implementing boolean functions in neural networks was studied before [31]. However, this work focuses directly on boolean logic, not Alloy, and it does not propose techniques to optimize the translation and reduce the number of gates. In another line of work that focused on training, Evans and Grefenstette introduced a differentiable inductive logic programming framework allowing one to to train a network with symbolic representation [32], while Payani and Fekri introduced a technique to train a network consisting of boolean logic nodes by defining a set of differentiable Boolean operators [33].

VII. Conclusion

This paper presented a new technique for creating desired machine learning models where instead of the traditional approach of training them using datasets, the models are directly synthesized from logical specifications, and thus, are correct by construction. Specifically, the technique translates relational specifications written in the well-known logic Alloy to neural networks that run on the widely used Tensorflow backend. The key insight is that a slight enhancement of traditional boolean gates can provide a rich intermediate representation that readily translates to neural networks. Enhanced gates are translated to minimal neural networks by employing the state-of-the-art program synthesis framework Rosette. The translation of Alloy specifications then follows the standard translation to boolean logic with the exception of utilizing the enhanced boolean gates, followed by a translation to neural networks. The technique is embodied in a prototype tool and used for experimental evaluation. The experimental results show that the technique allows synthesis of neural networks that are hard to create using traditional training methods.

Authorized licensed use limited to: University of Texas at Austin. Downloaded on September 03,2023 at 01:45:32 UTC from IEEE Xplore. Restrictions apply.

REFERENCES

[1] F. Leofante, N. Narodytska, L. Pulina, and A. Tacchella, "Automated verification of neural networks: Advances, challenges and perspectives," 2018.

[2] K. Pei, Y. Cao, J. Yang, and S. Jana, "Towards practical verification of machine learning: The case of computer vision systems," 2017.

[3] D. Kourie and B. Watson, *The Correctness-By-Construction Approach to Programming*. Springer, 2012.

[4] D. Jackson, "Alloy: A lightweight object modelling notation," *ACM Trans. Softw. Eng. Methodol.*, vol. 11, no. 2, p. 256290, Apr. 2002. [Online]. Available: https://doi.org/10.1145/505145.505149

[5] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 265–283. [Online]. Available: https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf

[6] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: engineering an efficient sat solver," in *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*, 2001, pp. 530–535.

[7] N. Eén and N. Sörensson, "An extensible SAT-solver," in *Theory and Applications of Satisfiability Testing*, E. Giunchiglia and A. Tacchella, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 502–518.

[8] N. Chong, T. Sorensen, and J. Wickerson, "The semantics of transactions and weak memory in x86, power, arm, and c++," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 211225. [Online]. Available: https://doi.org/10.1145/3192366.3192373

[9] D. Jackson and K. Sullivan, "Com revisited: Tool-assisted modelling of an architectural framework," in *Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering: Twenty-First Century Applications*, ser. SIGSOFT 00/FSE-8. New York, NY, USA: Association for Computing Machinery, 2000, p. 149158. [Online]. Available: https://doi.org/10.1145/355045.355065

[10] S. Khurshid and D. Jackson, "Exploring the design of an intentional naming scheme with an automatic constraint analyzer," in *ASE*, 2000, pp. 13–22.

[11] J. Wickerson, M. Batty, T. Sorensen, and G. A. Constantinides, "Automatically comparing memory consistency models," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 190204. [Online]. Available: https://doi.org/10.1145/3009837.3009838

[12] P. Zave, "Reasoning about identifier spaces: How to make chord correct," *IEEE Transactions on Software Engineering*, vol. 43, no. 12, pp. 1144–1156, 2017.

[13] D. Jackson and M. Vaziri, "Finding bugs with a constraint solver," in *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 00. New York, NY, USA: Association for Computing Machinery, 2000, p. 1425. [Online]. Available: https://doi.org/10.1145/347324.383378

[14] D. Marinov and S. Khurshid, "TestEra: A novel framework for automated testing of Java programs," in *ASE*, 2001, pp. 22–31.

[15] C. Trippel, D. Lustig, and M. Martonosi, "Checkmate: Automated synthesis of hardware exploits and security litmus tests," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-51. IEEE Press, 2018, p. 947960. [Online]. Available: https://doi.org/10.1109/MICRO.2018.00081

[16] E. Torlak and R. Bodik, "Growing solver-aided languages with rosette," in *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, ser. Onward! 2013. New York, NY, USA: Association for Computing Machinery, 2013, p. 135152. [Online]. Available: https://doi.org/10.1145/2509578.2509586

[17] Y. Cheng, D. Wang, P. Zhou, and T. Zhang, "A survey of model compression and acceleration for deep neural networks," 2017.

[18] M. Usman, W. Wang, K. Wang, M. Vasic, H. Vikalo, and S. Khurshid, "A study of the learnability of relational properties (model counting meets machine learning)," *arXiv preprint arXiv:1912.11580*, 2019.

[19] E. Torlak and D. Jackson, "Kodkod: A relational model finder," in *Tools and Algorithms for the Construction and Analysis of Systems*, O. Grumberg and M. Huth, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 632–647.

[20] J. Bornholt and E. Torlak, "Synthesizing memory models from framework sketches and litmus tests," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 467481. [Online]. Available: https://doi.org/10.1145/3062341.3062353

[21] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340.

[22] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain," *Psychological Review*, vol. 65, no. 6, pp. 386–408, 1958.

[23] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat, "Combinatorial sketching for finite programs," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XII. New York, NY, USA: Association for Computing Machinery, 2006, p. 404415. [Online]. Available: https://doi.org/10.1145/1168857.1168907

[24] A. Solar-Lezama, C. G. Jones, and R. Bodik, "Sketching concurrent data structures," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 08. New York, NY, USA: Association for Computing Machinery, 2008, p. 136148. [Online]. Available: https://doi.org/10.1145/1375581.1375599

[25] F. Chollet *et al.*, "Keras," https://keras.io, 2015.

[26] M. T. Ribeiro, S. Singh, and C. Guestrin, ""Why Should I Trust You?": Explaining the Predictions of Any Classifier," in *KDD*, 2016.

[27] O. Bastani, Y. Pu, and A. Solar-Lezama, "Verifiable reinforcement learning via policy extraction," in *Advances in Neural Information Processing Systems*, 2018, pp. 2499–2509.

[28] M. Vasic, A. Petrovic, K. Wang, M. Nikolic, R. Singh, and S. Khurshid, "Moët: Interpretable and verifiable reinforcement learning via mixture of expert trees," *CoRR*, 2019.

[29] J. Bornholt, E. Torlak, D. Grossman, and L. Ceze, "Optimizing synthesis with metasketches," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016, pp. 775–788.

[30] M. Minsky and S. A. Papert, *Perceptrons: An introduction to computational geometry*. MIT press, 2017.

[31] B. Steinbach and R. Kohut, "Neural networks–a model of boolean functions," in *Boolean Problems, Proceedings of the 5th International Workshop on Boolean Problems*, 2002, pp. 223–240.

[32] R. Evans and E. Grefenstette, "Learning explanatory rules from noisy data," *Journal of Artificial Intelligence Research*, vol. 61, pp. 1–64, 2018.

[33] A. Payani and F. Fekri, "Inductive logic programming via differentiable deep neural logic networks," 2019.

APPENDIX A
COMPARISON OF KODKOD WITH OCELOT

Ocelot is a Rosette library that provides tools to construct relational specifications. Ocelot provides an embedding of Alloy in Rosette, which makes it simple to convert properties written in Alloy into Rosette. To translate Ocelot formulas, we developed a Java-based framework that has a front-end parser which treats the formula generated by Ocelot as a string input and constructs an AST. The back-end of the framework, using a methodology similar to the one used with Kodkod, generates a neural network from the AST. Currently, the Ocelot/Rosette translator does not support the enhanced `one` and `lone` gates. Table III shows the details of the networks created using the boolean formulas generated from Ocelot. Kodkod produces a more concise encoding of boolean formulas than Ocelot. For

Authorized licensed use limited to: University of Texas at Austin. Downloaded on September 03,2023 at 01:45:32 UTC from IEEE Xplore. Restrictions apply.

Property	Scope 5				Scope 10			
	Neural net config.		Inference times(ms)		Neural net config.		Inference times(ms)	
	Layers	NZW/TC	Batch	Online	Layers	NZW/TC	Batch	Online
Antisymmetric	4	85/1005	12	4	4	370/18010	28	5
Bijective	18	1418/90901	109	7	33	64753/153415076	90750	428
Connexivity	3	75/755	12	4	3	300/11010	22	4
Equivalence	7	988/64160	58	5	7	8868/4924010	2901	21
Function	14	1130/69105	73	5	29	63680/151516310	87782	421
Functional	16	1325/85955	82	7	31	64470/153152710	88727	422
Injective	17	1371/88407	113	7	32	64611/153283852	90444	433
Irreflexive	2	10/130	11	4	2	20/1010	13	43.33
Non-Strict Order	8	952/60077	55	6	8	8692/4761132	2768	20
Partial Order	7	939/59250	54	6	7	8674/4753945	2759	22
Pre Order	7	862/49478	50	4	7	8317/4351298	2511	21
Reflexive	1	5/25	9	4	1	10/100	11	4
Strict Order	7	866/50222	50	6	7	8326/4366787	2543	21
Surjective	17	1371/88407	105	7	32	64611/153283852	90552	428
Symmetric	4	125/1905	12	4	4	550/35110	37	5.04
Total Order	8	1015/65235	58	6	8	8970/4946800	2850	23
Transitive	6	850/48730	46	5	6	8300/4344410	2505	21

TABLE III: Size and performance of neural networks created using Ocelot/Rosette (*NZW* - Non-zero weights, *TC* - Total connections)

simpler properties like *reflexivity* and *irreflexivity*, the formulas are identical, and so are their networks. For properties like *transitivity*, Ocelot's interpretation requires two additional layers and around 30% more connections for both scopes of 5 and 10. Despite the difference in the number of connections, the inference times remain mostly comparable. In this case, while the scope 5 networks show similar performance, the scope 10 Kodkod network is about 120ms faster in batch inference setting. The differences become more pronounced for relations that use *multiplicities*. In the case of the *function* property, Ocelot requires $2x$ more layers compared to Kodkod for scope 5 and $2.4x$ more layers for scope 10. This has a significant impact on the efficiency of the networks. The scope 10 *function* network from Kodkod is $172x$ and $40x$ faster than Ocelot's network for batch and online inference, respectively.

Authorized licensed use limited to: University of Texas at Austin. Downloaded on September 03,2023 at 01:45:32 UTC from IEEE Xplore. Restrictions apply.