Lecture 8: Verifying Robustness

Trustworthy Machine Learning

Spring 2024

Adversarial Robustness

- **So far:**
  - **Adversarial Examples**
  - **Adversarial training**
  - **Certified robustness via randomized smoothing**
  - **Sample of current research on robustness for LLMs**

- **Next: Formal methods for verified robustness**
  - **Formalizing program verification: Pre/post conditions**
  - **Verification as constraint solving**
  - **Robustness checking as program verification**
  - **Specialized constraint solver ReluPlex for neural network verification**
  - **Verifying robustness by abstract interpretation (box and zonotopes)**

Robustness Verification

- **Key publications:**
  - **Reluplex: An efficient SMT solver for verifying deep neural networks; Katz et al; CAV 2017**
  - **An abstract domain for certifying neural networks; Singh et al; POPL 2019**

- **Acknowledgement for slides:**
  - **CIS 6730 (Computer Aided Verification) Lectures**
  - **Formal methods in AI: Gagandeep Singh and Madhu Parthasarathy (UIUC)**

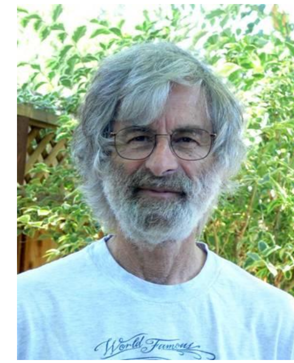
A Brief Introduction to Program Verification and Constraint Solving

Verified Programming

An axiomatic basis for computer programming
Tony Hoare; CACM 1969

In this paper an attempt is made to explore the logical foundations of computer programming by use of techniques which were first applied in the study of geometry and have later been extended to other branches of mathematics. This involves the elucidation of sets of axioms and rules of inference which can be used in proofs of the properties of computer programs. Examples are given of such axioms and rules, and a formal proof of a simple theorem is displayed. Finally, it is argued that important advantage, both theoretical and practical, may follow from a pursuance of these topics.

Formal Specifications



Who builds a house without drawing blueprints ?
Leslie Lamport; CACM 2015

Architects draw detailed plans before a brick is laid or a nail is hammered. But few programmers write even a rough sketch of what their programs will do before they start coding. We can learn from architects. A blueprint for a program is called a *specification*. An architect's blueprint is a useful metaphor for a software specification.

The need for specifications follows from two observations. The first is that it is a good idea to think about what we are going to do before doing it, and as the cartoonist Guindon wrote: "Writing is nature's way of letting you know how sloppy your thinking is." The second observation is that to write a good program, we need to think above the code level.

The main reason for writing a formal spec is to apply tools to check it. Tools cannot find design errors in informal specifications.

Specifications for programs

Input: int x, int y
Output: int r

```
z := x + y;
z' := x – y;
r := z * z'
```

Post-condition / assertion / "ensures" (in Dafny): $r = x^2 - y^2$

Cannot define correctness without specifications (logical constraints)

Verification Conditions

Input: int x, int y
Output: int r

```
z := x + y;
z' := x - y;
r := z * z'
```

Ensures $r = x^2 - y^2$

Verifying correctness means checking validity of the following logical formula, called VC (Verification Condition):

$$(z = x+y \;\&\&\; z' = x-y \;\&\&\; r = z*z') \rightarrow r = x^2 - y^2$$

From VC to Constraint Satisfaction

Verification problem: check validity of

$(z = x+y \ \&\& \ z' = x-y \ \&\& \ r = z*z') \rightarrow r = x^2 - y^2$

The answer is NO if the following set of constraints over integer variables are satisfiable:

$\{ z = x+y; \ z' = x-y; \ r = z*z'; r \ != x^2 - y^2 \}$

Satisfying solution to these constraints is a "counterexample" or a bug that shows why the program is incorrect

Example Constraint Satisfaction Problem

x, y, z : Boolean variables

Find values such that all the following constraints are satisfied

x | ~y | ~z
~x | y | ~z
~x | ~y | z

Example Constraint Satisfaction Problem

x, y, z : real-valued variables
Find values such that all the following constraints are satisfied

$$2x + 3y - 5z \leq 7$$
$$-5x + 2y \leq 106$$
$$9x - 4y + 29z \leq -1$$

What if we change the type to "integer" variables

Example Constraint Satisfaction Problem

x, y, z : real-valued variables

Find values such that all the following constraints are satisfied

$$2x + 3y - 5z \leq 7 \mid 3x - 4z \leq 9$$

$$-5x + 2y \leq 106 \mid 6y - 78z \leq 567$$

$$9x - 4y + 29z \leq -1 \mid 5x - 7y + z \leq 98$$

Mixture of Boolean and linear constraints

Solving Sudoku As Constraint Satisfaction

					3		8	5
		1		2				
			5		7			
		4				1		
	9							
5							7	3
		2		1				
				4				9

Solving Sudoku As Constraint Satisfaction

					3		8	5
		1		2				
			5		7			
		4				1		
	9							
5							7	3
		2		1				
				4				9

Variables: x(i,j), for i=1..9 and j=1..9, of integer value between 1 and 9
Find values such that all the following constraints are satisfied

Some values are known:

x(3,3)=1, x(2,6)=3, …

Every value appears in each row:

for each k=1, .., 9:

for each i=1, .. 9:

x(i,1)=k | x(i,2)=k | … | x(i,9) =k

Every value appears in each column: similar constraints

Every value appears in each square:

for each k = 1,… 9,

x(1,1)=k | x(1,2)= k | x(1,3)=k | …. | x(3,2)=k |x(3,3)=k

A Verification Example

```
Input:  int[32] x,  y  /* fixed precision 32 bit integers (as in C) */
Output: int[32] r
 int[32] z := x + y;        /*what's the precise semantics of addition?? */
 r := z / 2
Ensures  (x + y = 2r)            /* r is average of x and y */
```

Verification condition: Validity over int[32] variables

z = x + y && r = z/2 ➔ (x + y = 2r)

Constraint satisfaction problem :

{ z = x + y; r = z/2; x + y != 2r }

Satisfying solution: $x=y=2^{32} - 1$; r=z=0

Types matter!

Need to capture precise semantics of language constructs!

Logic for Specifications

Specification is a logical formula constructed from expressions over program variables using logical operators AND &&, OR |, NOT ~

Different choices based on:

- Allowed types of variables:
  - Boolean (bool), integer (int), natural numbers (nat), real numbers (real)
  - Enumerated types
  - Bit-vectors (fixed precision integers): int[32], int[64]
  - Arrays and matrices
- Operators allowed in expressions, e.g. for integers, three classes:
  - Difference constraints: $x - y \leq 5$
  - Linear constraints: $2x + 3y - 5z \leq 7$
  - Full arithmetic: $r = x^2 - y^2$
- Whether quantifiers over variables are allowed

The more restricted the specification logic, easier it is for a tool to solve the resulting constraint satisfaction problem

Verifying Programs with Loops

Input: int x1, x2
Output: int y1, y2

```
Requires (0 ≤ x1 && x2 > 0)     /* Assumption on inputs; also called pre-condition */
 y1 := 0;
 y2 := x1;
 while (x2 ≤ y2) {
          y1 := y1+1;
          y2 := y2 - x2
          };
Ensures ( y1 = x1 div x2 && y2 = x1 rem x2 )
```

Division Example

Current verification tools require the programmer also to specify "loop invariants"

Condition over program variables that is satisfied at the beginning of each iteration

```
Requires (0 ≤ x1 && x2 > 0)
 y1 := 0;
 y2 := x1;
 while (x2 ≤ y2) {
                    Invariant ϕ : (x1 = y1*x2 + y2 && 0 ≤ x1 && x2 > 0)
          y1 := y1+1;
          y2 := y2 – x2
          };
Ensures ( y1 = x1 div x2 && y2 = x1 rem x2 )
```

Division Example

```
Requires (0 ≤ x1 && x2 > 0)
 y1 := 0;
 y2 := x1;
 while (x2 ≤ y2) { Invariant ϕ : (x1 = y1*x2 + y2 && 0 ≤ x1 && x2 > 0)
        y1 := y1+1;
        y2 := y2 – x2
        };
Ensures ( y1 = x1 div x2 && y2 = x1 rem x2 )
```

Verification conditions:

1. Initialization ensures that invariant holds on first iteration

   (0 ≤ x1 && x2 > 0) ➔ ϕ[y1/0; y2/x1]

2. Execution of the loops preserves loop invariant

   (x2 ≤ y2) && ϕ ➔ ϕ[y1/y1+1; y2/y2-x2]

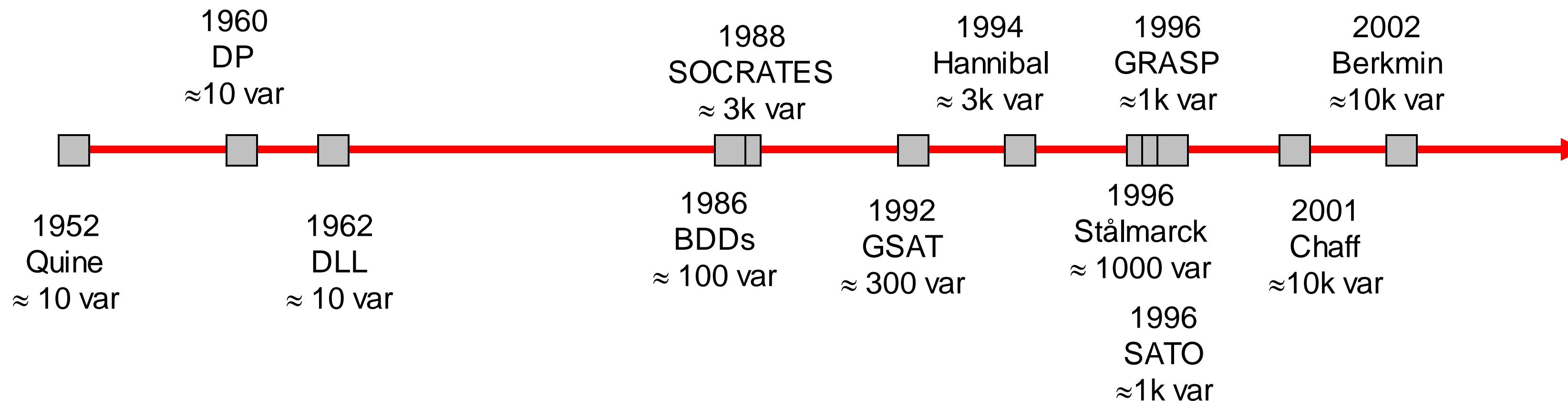
3. Post-condition holds upon termination of loop

   ~(x2 ≤ y2) && ϕ ➔ (y1 = x1 div x2 && y2 = x1 rem x2)

SAT Solvers

Propositional Satisfiability: Given a formula over Boolean variables, is there an assignment of 0/1's to variables which makes the formula true

- Canonical NP-hard problem (Cook 1973)
- Enormous progress in tools that can solve instances with thousands of variables and millions of clauses
- Also at the core of SMT solvers (extensions to richer classes of constraints)



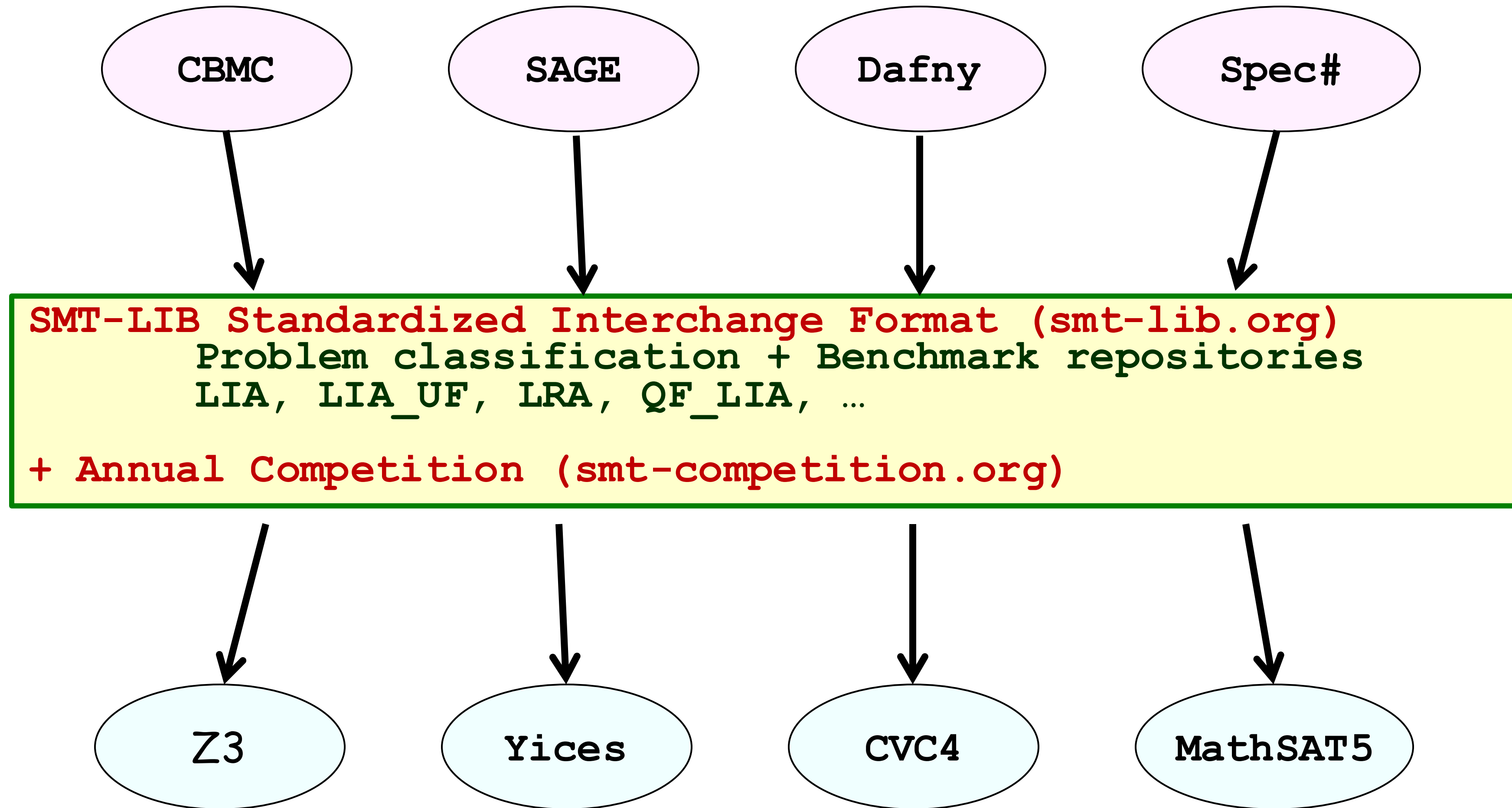
SMT: Satisfiability Modulo Theories

- Computational problem: Find a satisfying assignment to a formula
  - Boolean + Int types, logical connectives, arithmetic operators
  - Bit-vectors + bit-manipulation operations in C
  - Boolean + Int types, logical/arithmetic ops + Uninterpreted functions
- "Modulo Theory": Interpretation for symbols is fixed
  - Can use specialized algorithms (e.g. for arithmetic constraints)
- Progress in improved SMT solvers

Little Engines of Proof

SAT; Linear arithmetic; Congruence closure

SMT Solvers



Ever-growing scalability and use in different applications

Summary of Program Verification

- Proving correctness of programs requires programmer to write logical specifications
  - Pre/Post conditions for each function
  - Invariants for loops

- Proving correctness can then be translated automatically to constraint solving
  - Scalable SAT solvers for constraints over Boolean variable
  - Specialized scalable SMT solvers for more general classes of constraints / variable types

- Useful references

  - Satisfiability modulo theories: introduction and applications; de Moura and Bjorner; CACM 2011
  - The dogged pursuit of bug-free C programs: The Frama-C software analysis platform; Baudin et al; CACM, 2021
  - Dafny programming and verification language: https://dafny.org/
  - Talk by Byron Cook of Automated Reasoning Group at Amazong Web Services: An AWS Approach to Higher Standards of Assurance w/ Provable Security

Program Verification in the age of LLMs

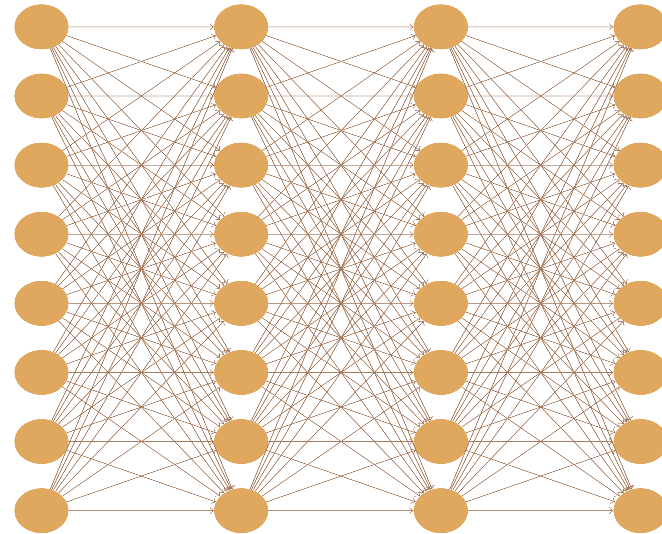
- Use of LLMs to assist program verification
  - Synthesis of candidate pre/post conditions and loop invariants

- Use program verification tools to ensure correctness of code generated by LLMs

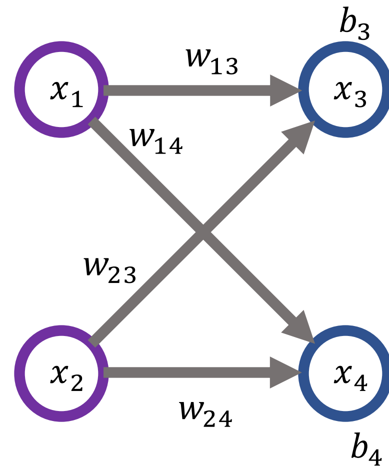
- Active research area in programming systems / software engineering

Formal Verification of Neural Networks

What is a deep neural network?



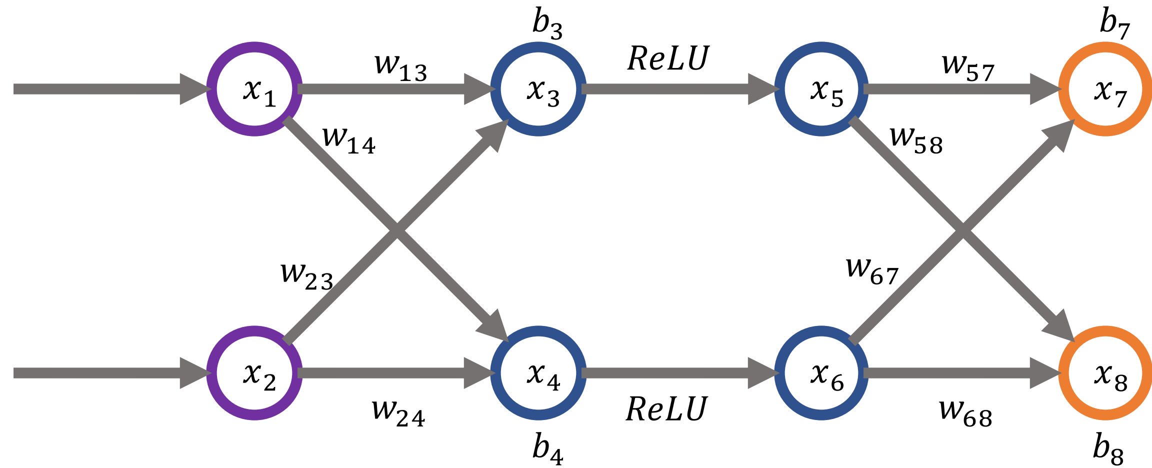
Each layer is a function



$(x_3, x_4) \leftarrow f_1(x_1, x_2)$ where
$x_3 = w_{13} \cdot x_1 + w_{23} \cdot x_2 + b_3$
$x_4 = w_{14} \cdot x_1 + w_{24} \cdot x_2 + b_4$

$(x_5, x_6) \leftarrow f_2(x_3, x_4)$ where
$x_5 = ReLU(x_3) = \max(0, x_3)$
$x_6 = ReLU(x_4) = \max(0, x_4)$

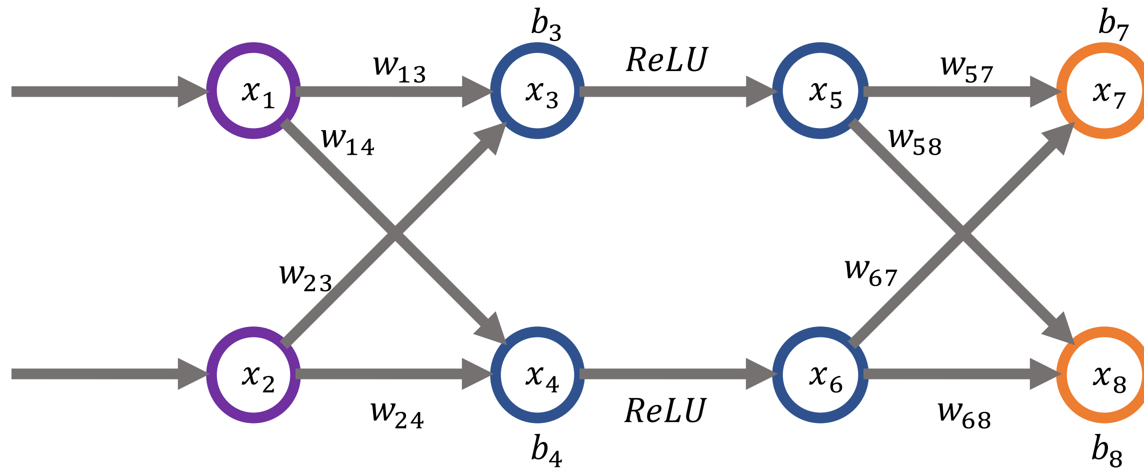
DNN is composition of layerwise functions



$(x_7, x_8) \leftarrow f(x_1, x_2) = f_3 \circ f_2 \circ f_1(x_1, x_2)$ where f_3 is the function computed by the third layer

DNNs and Programs

- DNNs can be seen as straight-line programs (programs without loops)



def $f(x_1, x_2)$:

$x_3 = w_{13} \cdot x_1 + w_{23} \cdot x_2 + b_3$

$x_4 = w_{14} \cdot x_1 + w_{24} \cdot x_2 + b_4$

$x_5 = \max(0, x_3)$

$x_6 = \max(0, x_4)$

$x_7 = w_{57} \cdot x_5 + w_{67} \cdot x_6 + b_7$

$x_8 = w_{56} \cdot x_5 + w_{68} \cdot x_6 + b_8$

return x_7, x_8

Specifications over DNNs

Precondition

$\forall x_1, x_2 . l_1 \leq x_1 \leq u_1, l_2 \leq x_2 \leq u_2$

DNN f

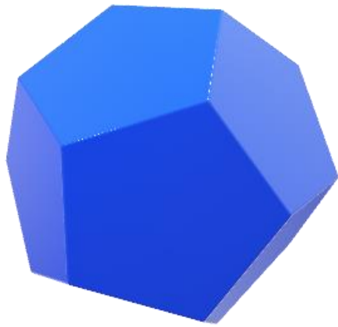
def $f(x_1, x_2)$:
$x_3 = w_{13} \cdot x_1 + w_{23} \cdot x_2 + b_3$
$x_4 = w_{14} \cdot x_1 + w_{24} \cdot x_2 + b_4$
$x_5 = \max(0, x_3)$
$x_6 = \max(0, x_4)$
$x_7 = w_{57} \cdot x_5 + w_{67} \cdot x_6 + b_7$
$x_8 = w_{56} \cdot x_5 + w_{68} \cdot x_6 + b_8$
return x_7, x_8

Postcondition

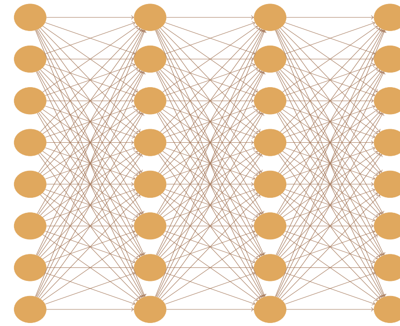
$x_7 > x_8$

Either prove that the network output satisfies the postcondition for all inputs in the pre-condition or find a counterexample

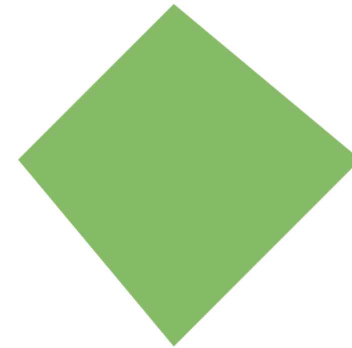
Neural network certification: problem statement



Precondition over network inputs ϕ

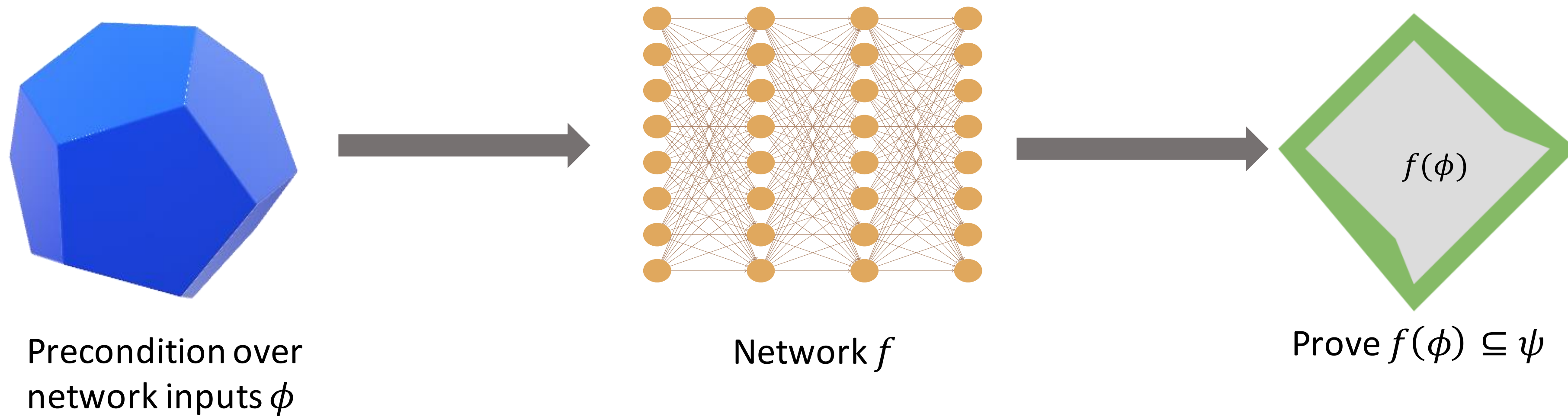


Network f

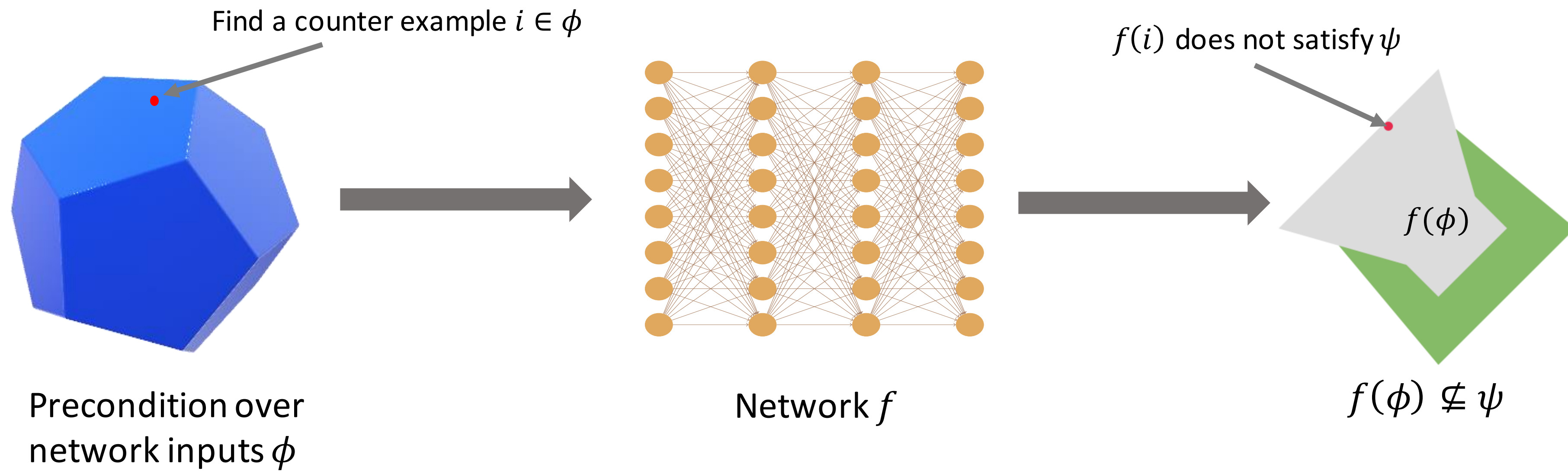


Postcondition over network outputs ψ

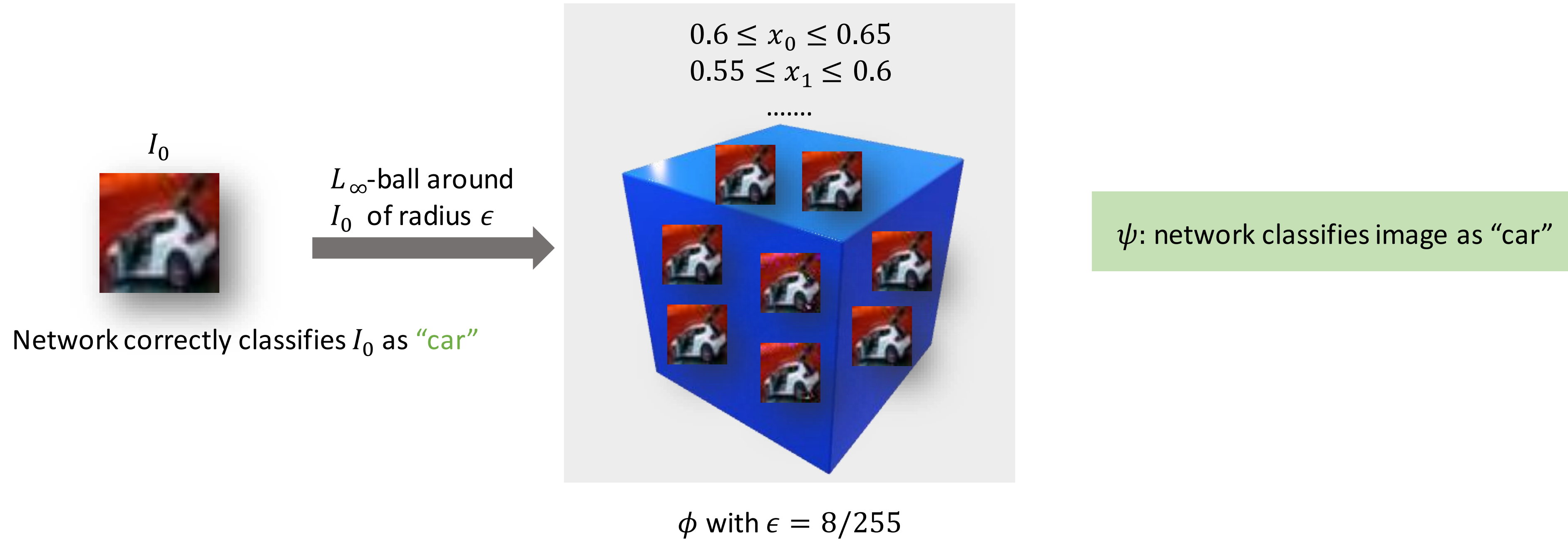
Neural network certification: problem statement

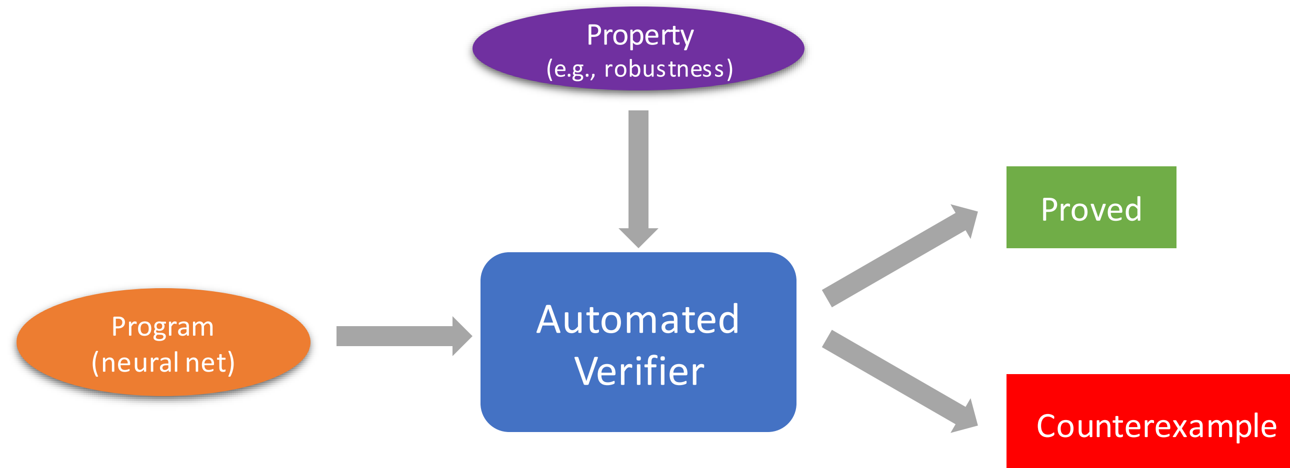


Neural network certification: problem statement



Robustness against adversarial perturbations





The general problem is computationally intractable, therefore we may need to provide an approximate answer

Soundness and Completeness for Certification

Sound	Complete	Guarantees
Yes	Yes	If the specification holds on the network, then the verifier proves it. The verifier does not prove any specification that does not hold
Yes	No	If the specification does not hold on the network, then the verifier does not prove it. Whenever the verifier proves a specification, it holds on the network
No	Yes	If the specification holds on the network, then the verifier proves it. The verifier may say that the specification holds even if it does not
No	No	Random Guessing

Desirable properties for certification:

- **Soundness**
- **Scalability**
- **Precision: ``as complete as possible''**

Certification of Neural Networks

Incomplete	Abstract interpretation: Box, Zonotope, DeepPoly
Complete	Mixed Integer Linear Programming (MILP) SMT solvers (Reluplex)

Active area of research with annual competition: VNNComp
Current winner: alpha-beta crown

Recap

- **Today : Robustness verification from a formal methods lens**
  - **Formalizing program verification: Pre/post conditions**
  - **Verification as constraint solving**
  - **Robustness checking as program verification**

- **Next: Verification techniques for neural networks**
  - **Specialized constraint solver ReluPlex for neural network verification**
  - **Verifying robustness by abstract interpretation (box and zonotopes)**