Generating and Checking DNN Verification Proofs

Hai Duong ThanhVu Nguyen Matthew Dwyer

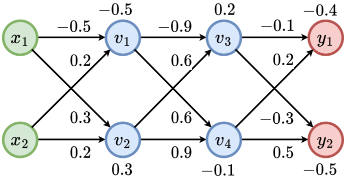
GEORGE MASON UNIVERSITY

UNIVERSITY of VIRGINIA

DNN Problems

DNN Verification

DNN Verification Example

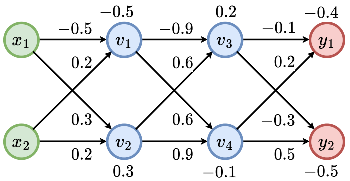


$$(x_1, x_2) \in [-2.0, 2.0] \times [-1.0, 1.0] \Rightarrow (y_1 > y_2)$$

DNN verifiers: return `unsat` for valid property and `sat` otherwise

DNN Verification

DNN Verification Example



$$(x_1, x_2) \in [-2.0, 2.0] \times [-1.0, 1.0] \Rightarrow (y_1 > y_2)$$

DNN verifiers: return `unsat` for valid property and `sat` otherwise

But, Could We Trust Verifiers?

- Currently, `sat` comes with counterexample; but `unsat` has no evidence!
- Bugs are inevitable (verification tools 20K+ lines of code)
- Multiple reports of soundness bugs in verifiers and literature
  - Claiming something is safe (`unsat`) when it is not

Needs Proof of "Proved" Results!

Challenges In Proof Generation and Checking

Challenge 1: Compatibility

Different verifiers use different algorithms and optimizations

Need a proof generation approach compatible with many verifiers

Challenges In Proof Generation and Checking

Challenge 1: Compatibility

Different verifiers use different algorithms and optimizations
Need a proof generation approach compatible with many verifiers

Challenge 2: Human-Readability

Proofs needs to be represented and stored efficiently
Need a standard, compact, and human-readable format

Challenges In Proof Generation and Checking

Challenge 1: Compatibility

Different verifiers use different algorithms and optimizations
Need a proof generation approach compatible with many verifiers

Challenge 2: Human-Readability

Proofs needs to be represented and stored efficiently
Need a standard, compact, and human-readable format

Challenge 3: Efficiency

Proofs can be *very* large!!
Need an efficient proof checking algorithm

Challenge 1: Compatibility

Initialization

- Input: a DNN N and a property ϕ
- An `ActPatterns` $\leftarrow \{\emptyset\}$ to record branches (patterns)

Branch-and-Bound Loop

- **Select** (pop) a branch σ from `ActPatterns`.
- **Deduce (bound)** checks feasibility of σ wrt (N, ϕ)
  - If *infeasible*:
    - Prune this σ branch (verified)
  - If feasible:
    - **Decide (branch)** selects a neuron v
    - Splitting $\sigma \wedge (v = \text{on})$ and $\sigma \wedge (v = \text{off})$
    - Add new branches to `ActPatterns`

Termination

- If a counterexample is found during search, return (**sat,** `cex`)
- If no counterexample exists, return (**unsat,**)

Challenge 1: Compatibility

Initialization

- Input: a DNN N and a property ϕ
- An `ActPatterns` $\leftarrow \{\emptyset\}$ to record branches (patterns)
- A proof tree `prooftree` $\leftarrow \{\ \}$ to record conflicts

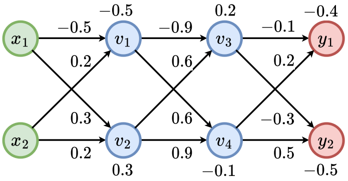
Branch-and-Bound Loop

- **Select** (pop) a branch σ from `ActPatterns`.
- **Deduce (bound)** checks feasibility of σ wrt (N, ϕ)
  - If *infeasible*:
    - Prune this σ branch (verified)
    - Record σ to `prooftree` $\rightarrow$ `prooftree` $\cup \{\sigma\}$
  - If feasible:
    - **Decide (branch)** selects a neuron v
    - Splitting $\sigma \wedge (v = \text{on})$ and $\sigma \wedge (v = \text{off})$
    - Add new branches to `ActPatterns`

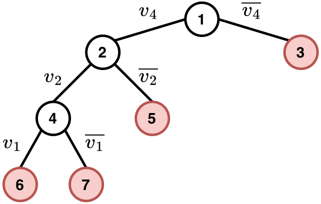
Termination

- If a counterexample is found during search, return (**sat,** `cex`)
- If no counterexample exists, return (**unsat**, `prooftree`)

Example: BaB + Proof Generation



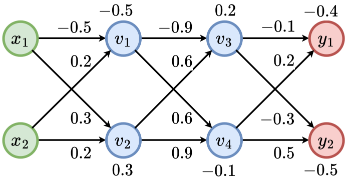
A simple DNN



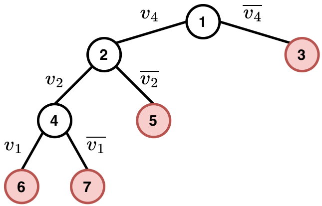
A proof tree

Verifying $(x_1, x_2) \in [-2.0, 2.0] \times [-1.0, 1.0] \Rightarrow (y_1 > y_2)$

Example: BaB + Proof Generation



A simple DNN



A proof tree

Verifying $(x_1, x_2) \in [-2.0, 2.0] \times [-1.0, 1.0] \Rightarrow (y_1 > y_2)$

Proof Tree

- Binary tree structure
  - each neuron decision branches into two children (representing on and off cases)
- Captures unsatisfiability reasoning of BaB
  - each leaf represents an unsatisfiable branch
  - $l_3 : \overline{v_4}$, $l_5 : v_4 \wedge \overline{v_2}$, $l_6 : v_4 \wedge v_2 \wedge v_1$, $l_7 : v_4 \wedge v_2 \wedge \overline{v_1}$

Challenge 2: Human-Readability

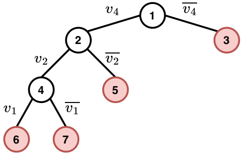
APTP Proof Language

```
⟨proof⟩ ::= ⟨declarations⟩ ⟨assertions⟩
⟨declarations⟩ ::= ⟨declaration⟩ | ⟨declaration⟩ ⟨declarations⟩
⟨declaration⟩ ::= (declare-const ⟨input-vars⟩ Real)
               | (declare-const ⟨output-vars⟩ Real)
               | (declare-pwl ⟨hidden-vars⟩ ⟨activation⟩)
⟨input-vars⟩ ::= ⟨input-var⟩ | ⟨input-var⟩ ⟨input-vars⟩
⟨output-vars⟩ ::= ⟨output-var⟩ | ⟨output-var⟩ ⟨output-vars⟩
⟨hidden-vars⟩ ::= ⟨hidden-var⟩ | ⟨hidden-var⟩ ⟨hidden-vars⟩
⟨activation⟩ ::= ReLU | Leaky ReLU | . . .
⟨assertions⟩ ::= ⟨assertion⟩ | ⟨assertion⟩ ⟨assertions⟩
⟨assertion⟩ ::= (assert ⟨formula⟩)
⟨formula⟩ ::= (⟨operator⟩ ⟨term⟩ ⟨term⟩)
           | (and ⟨formula⟩+) | (or ⟨formula⟩+)
⟨term⟩ ::= ⟨input-var⟩ | ⟨output-var⟩
        | ⟨hidden-var⟩ | ⟨constant⟩
⟨operator⟩ ::= < | ≤ | > | ≥
⟨input-var⟩ ::= X_⟨constant⟩
⟨output-var⟩ ::= Y_⟨constant⟩
⟨hidden-var⟩ ::= N_⟨constant⟩
⟨constant⟩ ::= Int | Real
```

Challenge 2: Human-Readability

APTP Proof Language

⟨*proof*⟩ ::= ⟨*declarations*⟩ ⟨*assertions*⟩
⟨*declarations*⟩ ::= ⟨*declaration*⟩ | ⟨*declaration*⟩ ⟨*declarations*⟩
⟨*declaration*⟩ ::= (**declare-const** ⟨*input-vars*⟩ **Real**)
| (**declare-const** ⟨*output-vars*⟩ **Real**)
| (**declare-pwl** ⟨*hidden-vars*⟩ ⟨*activation*⟩)
⟨*input-vars*⟩ ::= ⟨*input-var*⟩ | ⟨*input-var*⟩ ⟨*input-vars*⟩
⟨*output-vars*⟩ ::= ⟨*output-var*⟩ | ⟨*output-var*⟩ ⟨*output-vars*⟩
⟨*hidden-vars*⟩ ::= ⟨*hidden-var*⟩ | ⟨*hidden-var*⟩ ⟨*hidden-vars*⟩
⟨*activation*⟩ ::= ReLU | Leaky ReLU | . . .
⟨*assertions*⟩ ::= ⟨*assertion*⟩ | ⟨*assertion*⟩ ⟨*assertions*⟩
⟨*assertion*⟩ ::= (**assert** ⟨*formula*⟩)
⟨*formula*⟩ ::= (⟨*operator*⟩ ⟨*term*⟩ ⟨*term*⟩)
| (**and** ⟨*formula*⟩+) | (**or** ⟨*formula*⟩+)
⟨*term*⟩ ::= ⟨*input-var*⟩ | ⟨*output-var*⟩
| ⟨*hidden-var*⟩ | ⟨*constant*⟩
⟨*operator*⟩ ::= $<$ | $\leq$ | $>$ | $\geq$
⟨*input-var*⟩ ::= X_⟨*constant*⟩
⟨*output-var*⟩ ::= Y_⟨*constant*⟩
⟨*hidden-var*⟩ ::= N_⟨*constant*⟩
⟨*constant*⟩ ::= **Int** | **Real**



Example

```
; Declare variables
(declare-const X_0 Real)
(declare-const X_1 Real)
(declare-const Y_0 Real)
(declare-const Y_1 Real)
(declare-pwl N_1 ReLU)
(declare-pwl N_2 ReLU)
(declare-pwl N_3 ReLU)
(declare-pwl N_4 ReLU)
; Input constraints
(assert (>= X_0 -2.0))
(assert (<= X_0  2.0))
(assert (>= X_1 -1.0))
(assert (<= X_1  1.0))
; Output constraints
(assert (<= Y_0 Y_1))
; Hidden constraints
(assert (or
    (and (<  N_4 0))                            ; l_3
    (and (<  N_2 0) (>= N_4 0))                 ; l_5
    (and (>= N_2 0) (>= N_1 0) (>= N_4 0)) ; l_6
    (and (>= N_2 0) (<  N_1 0) (>= N_4 0)) ; l_7
))
```

Challenge 3: Efficiency

APTP Checker Algorithm

input : DNN $\mathcal{N}$, property $\phi_{in} \Rightarrow \phi_{out}$, *proof*
output : *certified* if proof is valid, otherwise *uncertified*

if ¬ `RepOK` (*proof*) **then** `RaiseError`(*Invalid proof tree*) ;
model ← `CreateMILP`($\mathcal{N}, \phi_{in}, \phi_{out}$) `// initialize MILP model with inputs`
while *proof* **do**
    node ← `Select`(*proof*) `// get node to check`
    model ← `AddConstrs`(*model*, *node*) `// add corresponding constraints`
    if `CheckFeasibility`(*model*) **then**
        return *uncertified* `// cannot certify`
return *certified*

Challenge 3: Efficiency

APTP Checker Algorithm

input : DNN $\mathcal{N}$, property $\phi_{in} \Rightarrow \phi_{out}$, *proof*
output : *certified* if proof is valid, otherwise *uncertified*

```
if ¬ RepOK (proof) then RaiseError(Invalid proof tree) ;
model ← CreateMILP(N, φ_in, φ_out) // initialize MILP model with inputs
while proof do
    node ← Select(proof) // get node to check
    model ← AddConstrs(model, node) // add corresponding constraints
    if CheckFeasibility(model) then
        return uncertified // cannot certify
return certified
```

Optimizations

1. **Stabilization:** fix stable ReLUs to reduce MILP constraints
2. **Pruning:** check parent nodes and if unsat then skip children
3. **Parallelization:** check multiple leaves in parallel

Evaluation

Name	Networks				Instances
	Num.	Layers	Neurons	Param.	Num.
CNN	8	1-2C;1F	320-3920	41K-180K	200
FNN	8	2-6F	64-3072	27K-1.7M	200

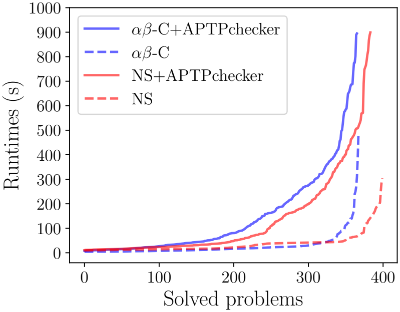
Benchmarks

8 CNNs, 8 FNNs; 25 properties; total 400 problem instances

Verifiers

Add APTP proof generation to $\alpha\beta$-CROWN, `NeuralSAT`, and Marabou

APTP Checker Performance



- Performance of verifiers (dashed) differ across configurations
    - They are able to verify between 337 and 400 problems
- APTP checker (solid) certified 93.7% and 99.4% of generated proofs
- This demonstrates that:
    - APTP proof format can encode proofs generated by different DNN verifiers
    - APTP checker can check them efficiently

Trade-offs Between Verifiers

Verifier	Num. Sub-Proofs		MILP Complexity	
	Mean	Median	Mean	Median
NeuralSAT	95	36	601	545
$\alpha\beta$-CROWN	230	180	414	179

- NeuralSAT: smaller proof trees, but with more complex MILP problems.
- $\alpha\beta$-CROWN: significantly larger proof trees, but with simpler MILP problems
- Strategies(?):
  - Generate larger proof trees with simpler MILPs for better parallelization
  - Adopting fast verification during development
  - Switching to proof-friendly strategies

APTP Checker vs. Marabou Checker

Checker	Proof checking time (seconds)		
	Mean	Median	Max
Marabou checker	4	204	785
APTP checker	3	9	38

- Extract APTP proofs from 54 problems that Marabou verified
- For simple problems, both checkers handle quickly (similar mean times)
- For challenging problems, APTP checker is over 20× speedup

Key Takeaways

1. Verifiers producing unsound results: UNACCEPTABLE!
2. Proof generation technique applicable to a wide-range of verifiers
3. Standard, compact, and human-readable proof format
4. Efficient, lightweight, and verifier-independent proof checker

Try APTP Checker:

`https://github.com/dynaroars/aptpchecker`

Thank You!