An Abstraction-Based Framework for Neural Network Verification

Guy Katz

SAT/SMT/AR/CP Summer School
August 2022
Acknowledgements

- Based on CAV 2020 and SEFM 2022 papers

- Joint work with:

Yizhak Elboher  Justin Gottschlich  Elazar Cohen

An Abstraction-Based Framework for Neural Network Verification, Elboher, Gottschlich & Katz, CAV 2020
Neural Network Verification using Residual Reasoning, Elboher, Cohen & Katz, SEFM 2022
Recall: DNN Verification

- Given: network $N$, properties $P$ and $Q$

- $Q$ is the *negation* of the desired property
  - SAT: the discovered point is a counterexample
  - UNSAT: property holds
DNN Verification Complexity

- Many approaches have been proposed
  - Complete and incomplete

- Everyone is struggling with network sizes

- Problem is NP complete:
  - $n$ neurons lead to $2^n$ operations

- So what can we do?
Abstraction

- Key idea: use a smaller network

Input Space

Output Space

$P$

$N$

$ar{N}$

$Q$
But why is this Sound?

- Network $\overline{N}$ is related to $N$
  - It is an over-approximation

- Common theme in verification

- If $\langle P, \overline{N}, Q \rangle$ is UNSAT, then $\langle P, N, Q \rangle$ is also UNSAT

- And what if $\langle P, \overline{N}, Q \rangle$ is SAT?
  - Ambiguous
  - We will handle this case later
Over-Approximations

Input Space

Output Space

$P$  $S$  $Q$

$\bar{R}$  $R$
Over-Approximations (cnt’d)

- Over-approximation has **all behaviors** of original system

- It is **simpler**, and easier to verify

- If over-approximation is safe, so is original
  - If over-approximation query is UNSAT, original query also UNSAT

- So how do we over-approximate neural networks?
Output Assumption

- Assume, without loss of generality:
  - Network $N$ has single output, $y$
  - Output property $Q: y > c$

- Over-approximation network $\overline{N}$:

$$\forall x. \quad \overline{N}(x) \geq N(x)$$

- If over-approximate query is UNSAT, $\forall x. \overline{N}(x) \leq c$
  - And hence, $N(x) \leq c$
  - So original query also UNSAT
Constructing $\overline{N}$

- **Idea:** merge two neurons into one
  - And repeat

- **Input weight:** \text{max}
- **Output weight:** \text{sum}
- **Example:** $N(1) = 17 \leq \overline{N}(1) = 32$
Constructing $\overline{N}$ (cnt’d)

- Why did it work?

\[
\bar{y} = (c + d) \cdot \text{ReLU}(\max(a, b) \cdot x)
\]

\[
= c \cdot \text{ReLU}(\max(a, b) \cdot x) + d \cdot \text{ReLU}(\max(a, b) \cdot x)
\]

- Works because:
  1. Weights $c$ and $d$ were positive
  2. We wanted $\bar{y}$ to increase
Constructing $\overline{N}$ (cnt’d)

- In order to merge two neurons:
  1. All outgoing edges must have same sign
  2. Next-layer neurons either need to increase, or decrease

- Four categories of neurons: $\{pos, neg\} \times \{inc, dec\}$

- Neurons from same category can be merged

- Input network not guaranteed to meet requirements
  - So we will preprocess them!
Preprocessing

- Part 1: all outgoing edges need to have same sign
  - We will double the network size
Preprocessing (cnt’d)

\[
\begin{align*}
  v_1 & \quad 1 \\
  v_1^+ & \quad 2 \\
  v_1^- & \quad -3 \\
  v_4 & \quad 1 \\
  v_5 & \quad 2 \\
  v_6 & \quad -3
\end{align*}
\]
Preprocessing (cnt’d)

- Assume all neurons are pos/neg

- Now classify as inc/dec
  - Inc: if neuron value increases, output neuron increases
  - Dec: if neuron value decreases, output neuron increases

- Start with output, work backwards

- Double neurons again, if needed
Preprocessing (cnt’d)

- All neurons classified as pos/neg, inc/dec
Preprocessing: Summary

- Mark output neuron as inc

- From the last-before-layer, **backwards:**
  - If neuron has positive and negative outgoing weights, double it (pos/neg)
  - If neuron is connected to inc and dec neurons, double it (inc/dec)

- Preprocessed network **completely equivalent** to original
  - Up to 4 times larger

- Wlog, assume input network is already preprocessed
Abstraction Operator

- Recall the pos/inc case:

$$
\bar{y} = (c + d) \cdot \text{ReLU}(\max(a, b) \cdot x)
$$

$$
= c \cdot \text{ReLU}(\max(a, b) \cdot x) + d \cdot \text{ReLU}(\max(a, b) \cdot x)
$$

$$
\geq c \cdot \text{ReLU}(a \cdot x) + d \cdot \text{ReLU}(b \cdot x)
$$

$$
= y
$$
Abstraction Operator (cnt’d)

- **Pos/dec** case:

\[
\begin{align*}
\bar{y} &= (c + d) \cdot \text{ReLU}(\min(a, b) \cdot x) \\
&= c \cdot \text{ReLU}(\min(a, b) \cdot x) + d \cdot \text{ReLU}(\min(a, b) \cdot x) \\
&\leq c \cdot \text{ReLU}(a \cdot x) + d \cdot \text{ReLU}(b \cdot x) \\
&= y
\end{align*}
\]
Current Abstraction Algorithm

- (Preprocess network)

- Apply abstraction to saturation
  - Each hidden layer has 4 nodes at most

- Verify abstract network $\overline{N}$
  - If UNSAT, original query is UNSAT
  - But what if SAT?
Coarse Abstractions

- Abstraction to saturation takes us to one extreme
  - Will likely be too coarse
Coarse Abstractions (cnt’d)

- Suppose we need to prove that $N(x) \geq 5$ is UNSAT

- We create abstract network, and get SAT
  - $\overline{N}(x_0) = 7$
  - But $N(x_0) = 3$
    - Spurious counter-example
    - Abstract network not suitable

- Create a slightly less abstract network, $\overline{N}'$
  - $\overline{N}'(x_0) = 4$
  - Maybe this is a better network to work with
Refinement

- The opposite of abstraction
  - Split previously-abstracted nodes

- Still an abstraction of the original network
  - But less abstract

- More formally:
  - Start with $N(x) \leq \bar{N}(x)$
  - Now: $N(x) \leq \bar{N}'(x) \leq \bar{N}(x)$
Refinement Operator

- Maintain a mapping from neuron to abstract neuron
- Edge weights re-computed

\[ v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow \tilde{v}_2 \rightarrow \tilde{v}_1 \]
Verification Algorithm

1. Generate initial abstraction $\overline{N}$
2. Verify $\overline{N}$
3. If UNSAT
   1) Stop and return UNSAT
4. If SAT
   1) Obtain counter example $x$
   2) Check whether $x$ is a counter example for the original $N$
   3) If yes, stop and return SAT
   4) Else, refine $\overline{N}$ and go to step 2

- The algorithm is guaranteed to converge
  - Sound and complete if underlying verifier is sound and complete
Which Node to Refine?

- In abstraction, we abstract to saturation

- What about refinement?
  - Want to refine as little as possible
  - But also want to rule out the spurious counter example $x$

- We will combine two criteria:
  - Weight-based
  - Counter-example guided
Weight-Based Refinement

- Look for coarse min/max abstractions
- Use refinement to make them more precise
Is Weight-Based Enough?

- It’s definitely a good start

- But, it doesn’t consider $x$

- Solution: maximize $|w_{max} - w_{min}| \cdot v(x)$

- Counter-example guided abstraction refinement
  - CEGAR
The ACAS Xu System

- Airborne Collision-Avoidance System for drones
- A new standard being developed by the FAA

Produce advisories:
1. Strong left (SL)
2. Weak left (L)
3. Strong right (SR)
4. Weak right (R)
5. Clear of conflict (COC)

- FAA considering an implementation that uses 45 deep neural networks
  - But wants to verify them!
The ACAS Xu System (cnt’d)

- Verified arbitrary properties of ACAS Xu
  - Similar to the ones specified by the FAA

![Graph showing sum query times for Marabou with abstraction]
The ACAS Xu System (cnt’d)

- 90 experiments total, 20 hour timeout

- Abstraction solved 58, vanilla just 35
  - Median query time for abstraction: 1045 seconds. Vanilla: 63671
  - Average final network size for abstraction: 385 nodes
    - Original networks: 310
    - But still faster
Adversarial Inputs

- Slight input perturbations cause misclassification

\[ \text{Image} + \epsilon \times \text{Noise} = \text{Image} \]

- Can use verification to prove the absence of such inputs

Goodfellow et al., 2015
Adversarial Robustness

- Verified adversarial robustness properties of ACAS Xu
Adversarial Robustness (cnt’d)

- 900 experiments total, 20 hour timeout

- Abstraction solved 805, vanilla solved 893
  - Median query time for abstraction: 0.026 seconds. Vanilla: 15.07
    - 99% reduction in time
  - Average final network size for abstraction: 104.4 nodes
    - Original networks: 310
    - Much smaller
Residual Reasoning

- Common CEGAR work-flow:
  - Generate initial abstraction
  - Verify
  - Obtain spurious counter-example
  - Refine

- **Key observation**: each verification call is **oblivious** of the past

- Idea: **re-use** some information to expedite the process
Recap: Case Splitting

- Case splitting approach:
  - Fix *each* ReLU to a linear segment (active or inactive)
  - Solve the resulting linear problem
  - If property is violated for this configuration, stop
  - But if property holds, backtrack and try other option

\[
a + b + c = 5, b > c
\]

- State explosion: 300 ReLUs $\rightarrow 2^{300}$ checks
Remembering “Bad” Splits

- Observe neurons $v_1, v_2$ in abstract network
  - Associate $v$ with Boolean variables $l_{v_1}, l_{v_2}$

- Consider the following scenario:
  - Verifier splits on $v_1, v_2$, sets both to inactive
    - $l_{v_1} \leftarrow F$, $l_{v_2} \leftarrow F$
  - Hits UNSAT
  - Then explores other splits, produces spurious SAT assignment
  - Performs refinement on $v_2$, splits it to $\bar{v}_2^1, \bar{v}_2^2$
Remembering “Bad” Splits (cnt’d)

- Already know: in $\overline{N}$, $\langle l_1 \lor l_2 \rangle$ is implied by the formula
  - Like learned clauses in SAT solving

- But what happens when we switch to $\overline{N'}$?
  - Can learn: $\langle l_1 \lor l_1^1 \lor l_2^2 \rangle$
  - Under some constraints…
Residual Reasoning Workflow

- Form of query: \( \langle P, N, Q, \Gamma \rangle \)
  - \( \Gamma \) is a **context**:
    - CNF formula, literals correspond to activation functions
    - \( \Gamma \) is implied by \( \langle P, N, Q \rangle \): satisfying assignments must satisfy \( \Gamma \)

- A solver can:
  - **Store information** in \( \Gamma \)
    - Record any list of splits that led to an UNSAT branch
  - **Read information** from \( \Gamma \) and perform unit-propagation
    - If \( (v_1 \lor v_2) \in \Gamma \) and \( v_1 \) already false, set \( v_2 \) to true without splitting
    - Can implement this efficiently with **watch literals**
High-level Workflow
Limitations and Overhead

- The goal is to prevent case-splitting, but there are costs
  - Actual overhead of populating and reading from $\Gamma$
  - Need to instrument the solver
    - Unlike in previous abstraction/refinement work
  - Extra book-keeping when performing refinement
    - E.g., rename $l_2$ in $\Gamma$ to $l_2^1 \lor l_2^2$

- Theoretical foundation: need to prove that clauses added to $\Gamma$ are sound
  - Depends on abstraction scheme in use
  - Quite complex – see paper
Experiments on ACAS Xu

- $AR^4$: Abstraction-Refinement with Residual Reasoning for Reluplex
  - Implemented on top of the Marabou tool
  - Evaluated on ACAS Xu

|                          | Adversarial | Safety | Total (Weighted) |
|--------------------------|-------------|--------|------------------|
|                          | $AR^4$     | $AR^4$| $AR^4$           |
| Timeouts                 | 95/900     | 7/180 | 102/1080         |
| Instances solved more quickly | 160        | 28    | 188              |
| Uniquely solved          | 26         | 2     | 28               |
| Visited tree states      | 6.078      | 3.569 | 5.634            |
| Avg. instrumentation time | 91.54      | 36.5  | 82.367           |
Next Steps

- Better initial abstraction (fewer refinement steps)
- Better refinement: split neuron into arbitrary subsets
- Better residual reasoning: populate $\Gamma$ with more clauses
Thank You!

Questions