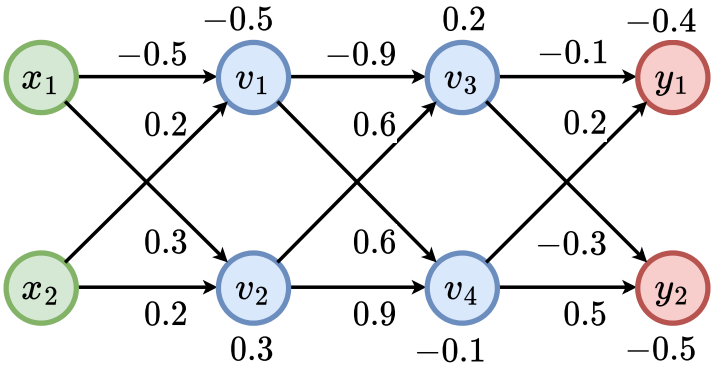
Compositional Neural Network Verification via Assume-Guarantee Reasoning

Hai Duong David Shriver ThanhVu Nguyen Matthew Dwyer

DNN Problems

DNN Verification

DNN Verification Example



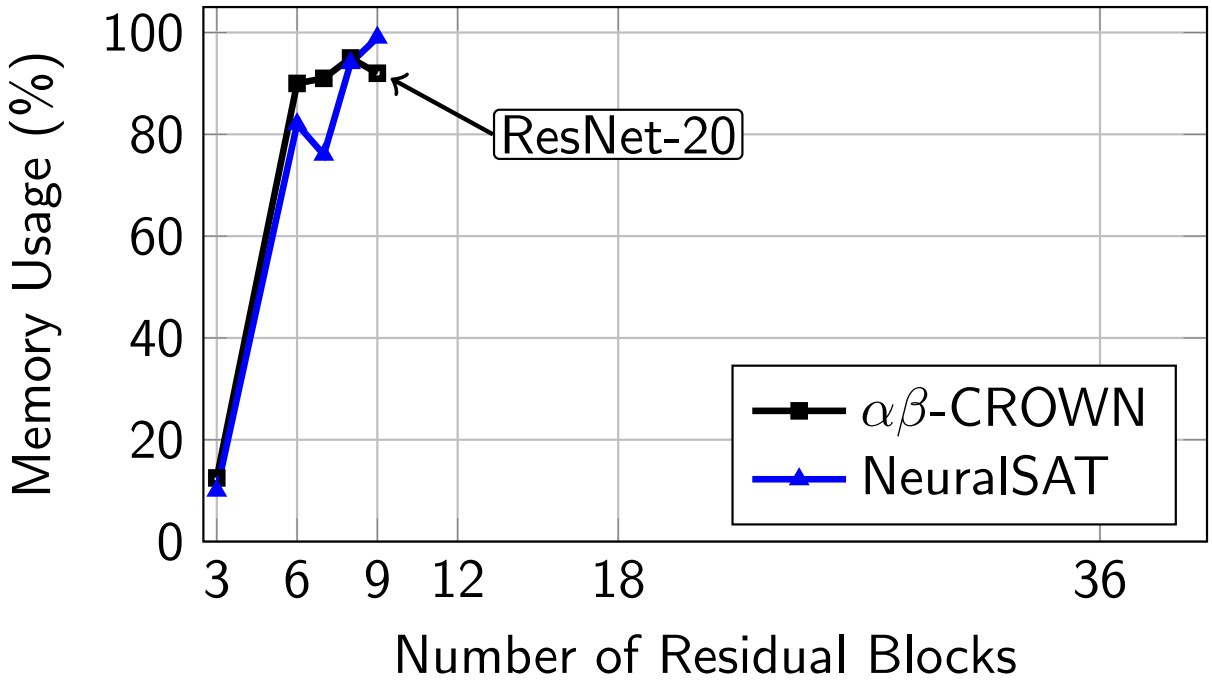
$$(x_1, x_2) \in [-2.0, 2.0] \times [-1.0, 1.0] \Rightarrow (y_1 > y_2)$$

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

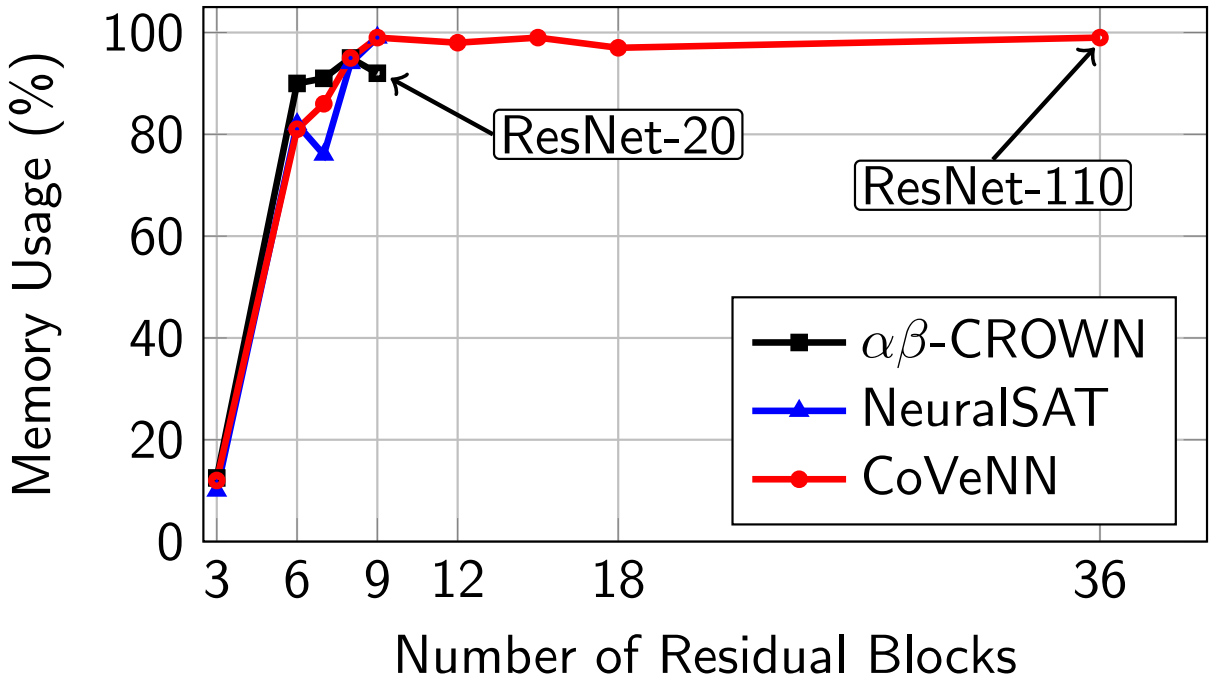
Scalability Issue

- Currently, verifiers can handle small DNNs, e.g., VGG
- DNN verifiers rely heavily on GPUs
- DNN sizes grow faster than GPU memory

Scalability Issue



Scalability Issue



Compositional verification can help!

CoVeNN: Compositional Verification

Decomposing DNNs

Decompose a DNN into smaller sub-networks $N \equiv N_1 \circ N_2 \circ \cdots \circ N_k$

Each sub-network N_i can be verified independently

CoVeNN: Compositional Verification

Decomposing DNNs

Decompose a DNN into smaller sub-networks $N \equiv N_1 \circ N_2 \circ \cdots \circ N_k$
Each sub-network N_i can be verified independently

Generating Assumptions

Generate assumptions (properties) to connect sub-networks N_i and N_{i+1}
Assume N_i has certain properties, not formally verified yet

CoVeNN: Compositional Verification

Decomposing DNNs

Decompose a DNN into smaller sub-networks $N \equiv N_1 \circ N_2 \circ \cdots \circ N_k$
Each sub-network N_i can be verified independently

Generating Assumptions

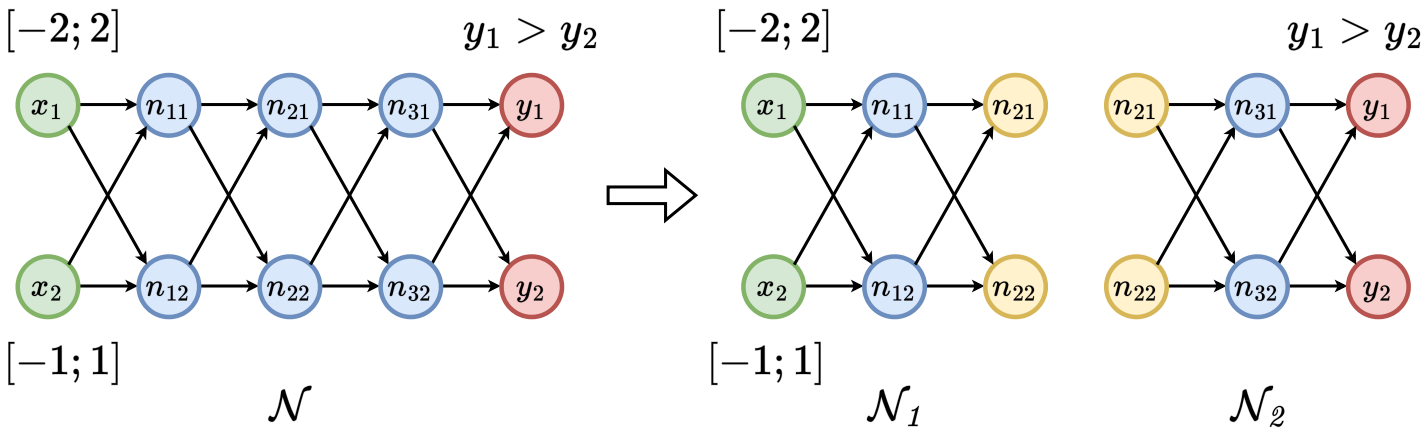
Generate assumptions (properties) to connect sub-networks N_i and N_{i+1}
Assume N_i has certain properties, not formally verified yet

Refining Assumptions

Verify assumptions, then refine them based on the verification results
Making the assumptions valid and more precise

Motivating Example

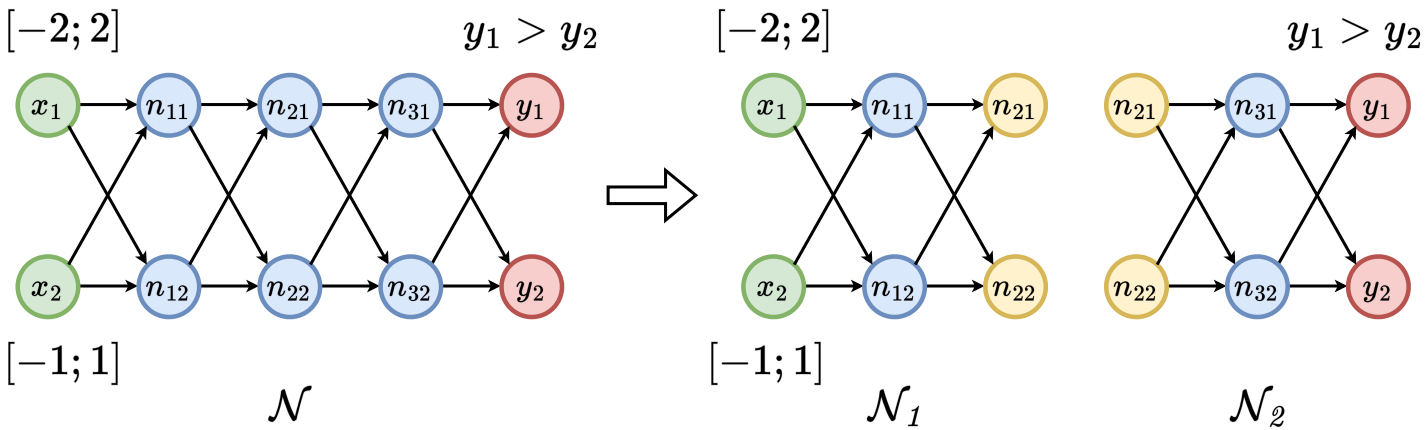
Decomposition



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

Motivating Example

Decomposition



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

Verification Chain

$$\underbrace{\mathcal{N}_1 : (x_1, x_2) \in [-2.0, 2.0] \times [-1.0, 1.0] \Rightarrow \gamma}_{(1)} \qquad \underbrace{\mathcal{N}_2 : \gamma \Rightarrow (y_1 > y_2)}_{(2)}$$

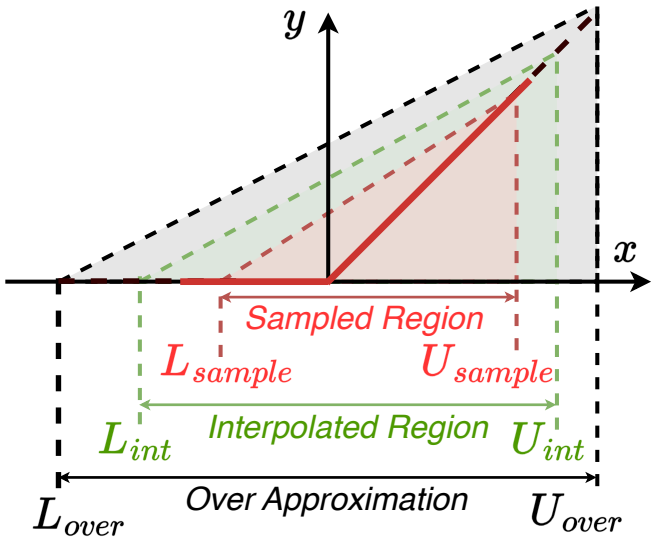
Generating and Refining Assumptions

Verification Chain

$$\mathcal{N}_1 : (x_1, x_2) \Rightarrow \gamma \quad (1)$$
$$\mathcal{N}_2 : \gamma \Rightarrow (y_1 > y_2) \quad (2)$$

Over-approximation $\gamma_o = \mathit{OverApprox}(\mathcal{N}_1, x)$

$\gamma_o = -5 \leq n_{21} \leq 5 \wedge -10 \leq n_{22} \leq 10$ (1)✓ (2)✗

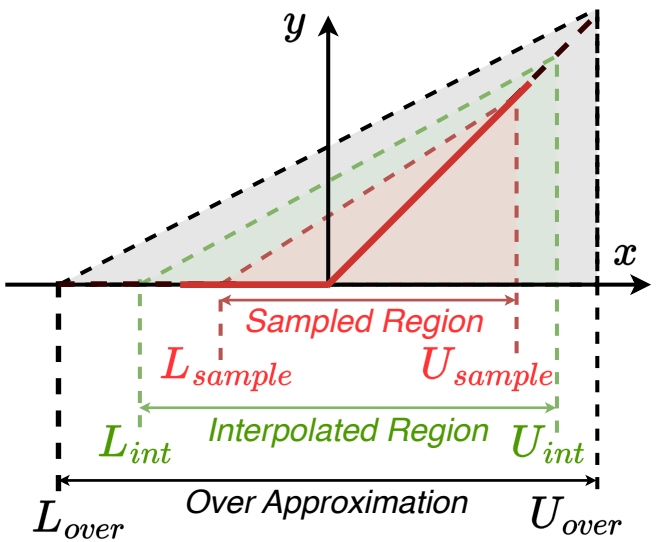


Regions of a ReLU

Generating and Refining Assumptions

Verification Chain

$$\mathcal{N}_1 : (x_1, x_2) \Rightarrow \gamma \qquad (1)$$
$$\mathcal{N}_2 : \gamma \Rightarrow (y_1 > y_2) \qquad (2)$$



Regions of a ReLU

Over-approximation $\gamma_o = OverApprox(\mathcal{N}_1, x)$

$\gamma_o = -5 \leq n_{21} \leq 5 \wedge -10 \leq n_{22} \leq 10$ (1)✓ (2)✗

Sampling $\gamma_s = Sample(\mathcal{N}_1, x)$

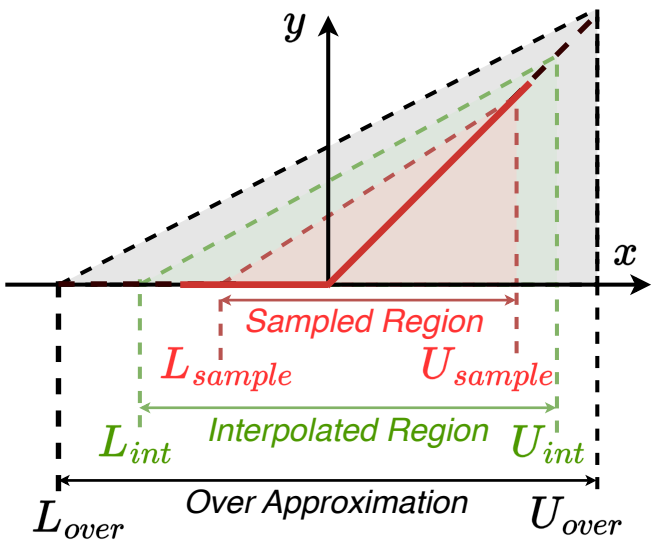
$\gamma_s = -2 \leq n_{21} \leq 2 \wedge -4 \leq n_{22} \leq 4$ (1)✗ (2)✗

- (1) returns sat with counterexamples

Generating and Refining Assumptions

Verification Chain

$$\mathcal{N}_1 : (x_1, x_2) \Rightarrow \gamma \qquad (1)$$
$$\mathcal{N}_2 : \gamma \Rightarrow (y_1 > y_2) \qquad (2)$$



Regions of a ReLU

Over-approximation $\gamma_o = OverApprox(\mathcal{N}_1, x)$

$\gamma_o = -5 \leq n_{21} \leq 5 \wedge -10 \leq n_{22} \leq 10$ (1)✓ (2)✗

Sampling $\gamma_s = Sample(\mathcal{N}_1, x)$

$\gamma_s = -2 \leq n_{21} \leq 2 \wedge -4 \leq n_{22} \leq 4$ (1)✗ (2)✗

- (1) returns sat with counterexamples

Interpolation $\gamma_o \supseteq \gamma \supseteq \gamma_s$

$\gamma = -3 \leq n_{21} \leq 3 \wedge -5 \leq n_{22} \leq 5$ (1)✗ (2)✗

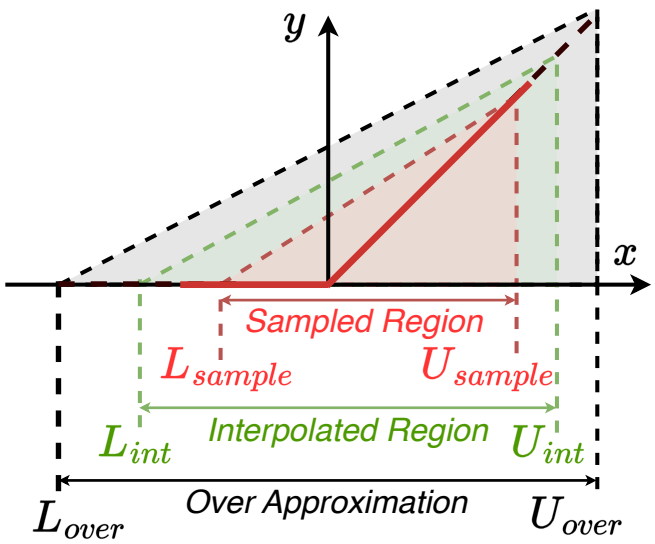
- If (1) returns unsat, use γ to verify (2)
- Otherwise, (1) returns unknown, use (1) result to refine γ

$\gamma = -4 \leq n_{21} \leq 4 \wedge -7 \leq n_{22} \leq 7$ (1)✓ (2)✓

Generating and Refining Assumptions

Verification Chain

$$\mathcal{N}_1 : (x_1, x_2) \Rightarrow \gamma \quad (1)$$
$$\mathcal{N}_2 : \gamma \Rightarrow (y_1 > y_2) \quad (2)$$



Regions of a ReLU

Over-approximation $\gamma_o = OverApprox(\mathcal{N}_1, x)$

$\gamma_o = -5 \leq n_{21} \leq 5 \wedge -10 \leq n_{22} \leq 10$ (1)✓ (2)✗

Sampling $\gamma_s = Sample(\mathcal{N}_1, x)$

$\gamma_s = -2 \leq n_{21} \leq 2 \wedge -4 \leq n_{22} \leq 4$ (1)✗ (2)✗

- (1) returns sat with counterexamples

Interpolation $\gamma_o \supseteq \gamma \supseteq \gamma_s$

$\gamma = -3 \leq n_{21} \leq 3 \wedge -5 \leq n_{22} \leq 5$ (1)✗ (2)✗

- If (1) returns unsat, use γ to verify (2)
- Otherwise, (1) returns unknown, use (1) result to refine γ

$\gamma = -4 \leq n_{21} \leq 4 \wedge -7 \leq n_{22} \leq 7$ (1)✓ (2)✓

Using **Intervals** ⇒ Support any activation functions, not limited to ReLU

CoVeNN Algorithm

input : Verifier $\mathcal{V}$, DNN $\mathcal{N}$, property $\phi_{in} \Rightarrow \phi_{out}$, rounds r
output : *unsat* if the property is valid and *unknown* otherwise

1 $(\mathcal{N}_1, ..., \mathcal{N}_K) \leftarrow$ `decomposeNetwork`$(\mathcal{N})$ `// split` $\mathcal{N}$ `into` K `subnetworks`
2 **for** $i \in [1, ..., K]$ **do** `// initialize coarse overapproximation`
3 $\quad \gamma_{over}.append($`overApproximate`$(\mathcal{N}_i, \gamma_{over}[i-1]))$
4 **if** `isVerified`$(\gamma_{over}[K], \phi_{out})$ **then** `//` $\text{CoVeNN}_{\cancel{\text{Refine}}}$ `terminates here`
5 $\quad$ **return** *unsat*

13 **return** *unknown*

CoVeNN Algorithm

input : Verifier $\mathcal{V}$, DNN $\mathcal{N}$, property $\phi_{in} \Rightarrow \phi_{out}$, rounds r
output : *unsat* if the property is valid and *unknown* otherwise

1 $(\mathcal{N}_1, ..., \mathcal{N}_K) \leftarrow$ `decomposeNetwork`$(\mathcal{N})$ // split $\mathcal{N}$ into K subnetworks
2 **for** $i \in [1, ..., K]$ **do** // initialize coarse overapproximation
3 $\quad \gamma_{over}.append($`overApproximate`$(\mathcal{N}_i, \gamma_{over}[i-1]))$
4 **if** `isVerified`$(\gamma_{over}[K], \phi_{out})$ **then** // CoVeNN$_{\text{~~Refine~~}}$ terminates here
5 $\quad$ **return** *unsat*
6 **for** $r_i \in [1, ..., r]$ **do** // iterate r times
7 $\quad$ **for** $i \in [1, ..., K-1]$ **do** // analyze first $K-1$ subnetworks
8 $\quad\quad \gamma_{assume} \leftarrow$ `generateAssumption`$(\mathcal{N}_i, \gamma_{over}[i])$
9 $\quad\quad info \leftarrow \mathcal{V}.$`verify`$(\mathcal{N}_i, \gamma_{over}[i], \gamma_{assume})$
10 $\quad\quad \gamma_{over}[i+1] \leftarrow$ `refineAssumption`$(\mathcal{N}_i, \gamma_{assume}, info)$
11 $\quad$ **if** $\mathcal{V}.$`verify`$(\mathcal{N}_K, \gamma_{over}[K-1], \phi_{out})$ **then**
12 $\quad\quad$ **return** *unsat* // property is valid
13 **return** *unknown*

Evaluation

Benchmark	Layers	Neurons	Parameters	Properties
VAE_BASE	20	43K	10K	20
VAE_WIDE	20	86K	39K	20
VAE_DEEP	28	44K	15K	20
RESNET6	20	283K	113K	20
RESNET12	38	627K	230K	20
RESNET18	56	700K	348K	20
RESNET36	110	1032K	706K	20
Total				**140**

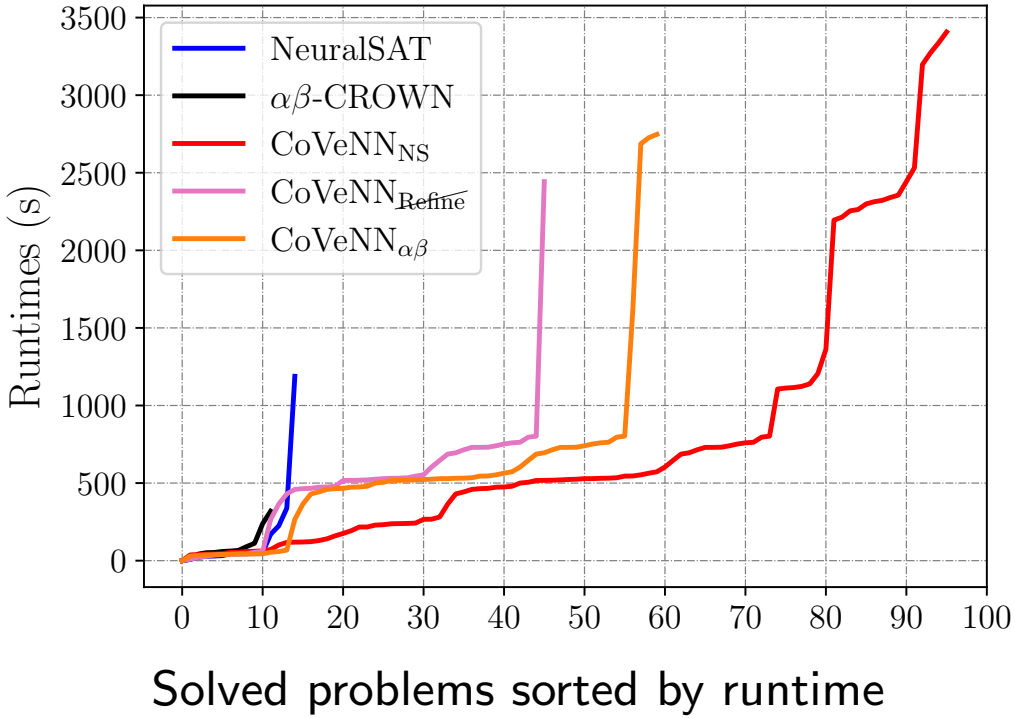
Benchmarks

7 networks; 2 architectures; total 140 problem instances

Verifiers

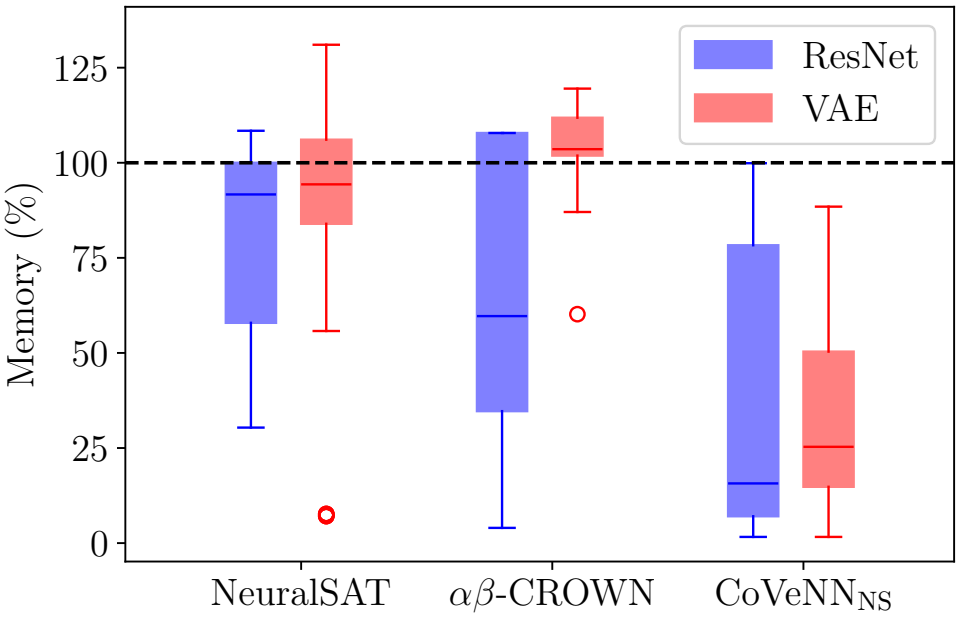
$\alpha\beta$-CROWN, NeuralSAT, CoVeNN$_{\not{\text{Refine}}}$, CoVeNN$_{\alpha\beta}$, and CoVeNN$_{\text{NS}}$

Performances Compared to SoTA Verifiers



- SoTA verifiers (e.g., NeuralSAT) verified only 14 (10%) problems
- CoVeNN is verifier-agnostic and significantly improves the performances
  - CoVeNN$_{\not{\text{Refine}}}$ verified 45 (32%) problems
  - CoVeNN$_{\text{NS}}$ verified 95 (68%) problems

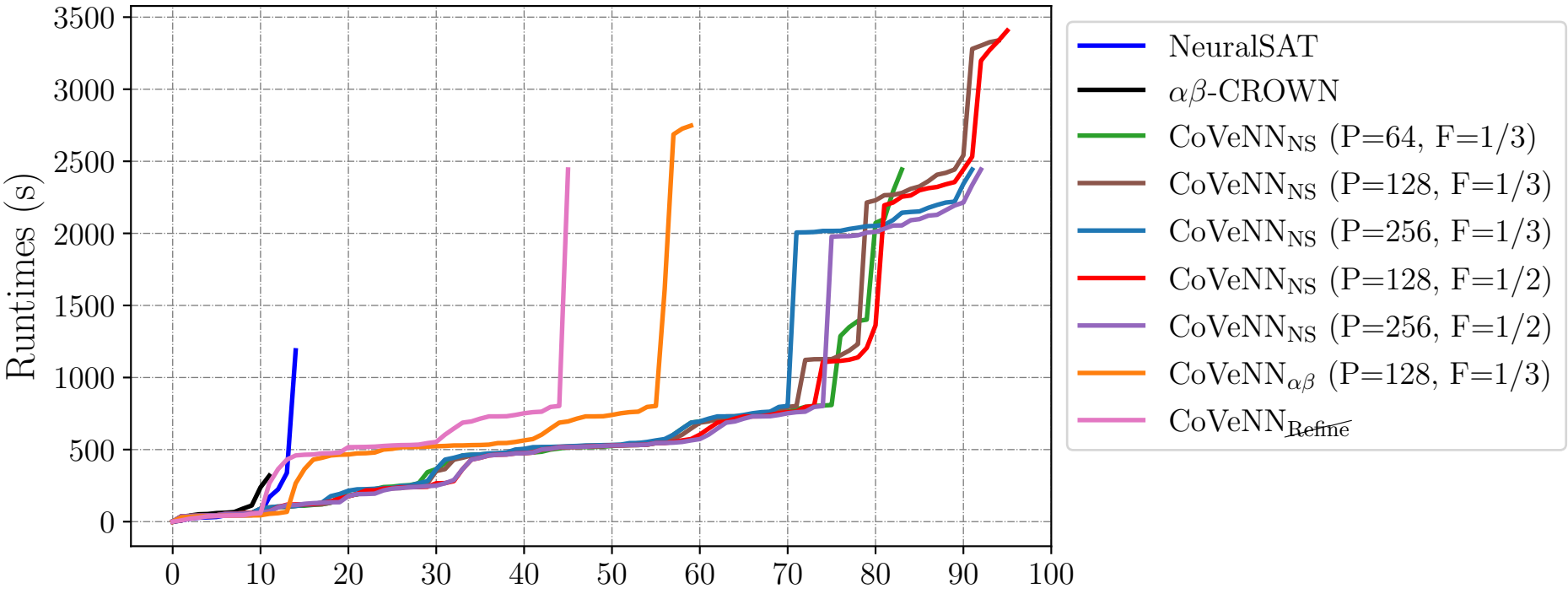
Memory Usage per Verifier



$$mem_{usage} = \frac{mem_{cur} + mem_{req} - mem_{free}}{mem_{total}}$$

- $\alpha\beta$-CROWN and NeuralSAT often encounter OOM errors ($> 100\%$)
- CoVeNN significantly reduces the memory usage with the limit

Performance with Parameter Tuning



- CoVeNN's performance depends on number of assumptions generated
- CoVeNN is robust to how assumptions are interpolated.
  - CoVeNN compensates assumptions through refinement

Key Takeaways

1. CoVeNN significantly improves the performances and reduces the memory usage
2. CoVeNN is activation-agnostic and verifier-agnostic

CoVeNN is part of NeuralSAT project:

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

Thank You!