Title

Sanaa Siddiqui, Diganta Mukhopadhyay, Afzal Mohammed, Hrishikesh Karmarkar, Kumar Madhukar

ACM Reference Format:
Sanaa Siddiqui, Diganta Mukhopadhyay, Afzal Mohammed, Hrishikesh Karmarkar, Kumar Madhukar . 2024. Title. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

TODO: Add affiliations

1 Introduction

TODO: Re-work wrt new experiments section

Advances in Deep Neural Networks (DNNs) have enabled the scalable solution of several previously intractable problems like image recognition.TODO: cite Due to this, DNNs have been increasingly assumed a central role across various domains. These include several safety critical domains like healthcare [1], where they contribute significantly to medical diagnostics and predictive analysis [**b2**], and autonomous vehicles, where DNNs serve as the backbone for sophisticated perception systems, supporting tasks such as object recognition and decision-making [**b3**].

However, DNNs are well known to be vulnerable to adversarial attacks TODO: cite and can produce unexpected behaviours which in safety critical domains can lead to unsafe situations. Therefore, there is a critical need to guarantee the precise functionality and safety of these DNNs that are deployed in safety-critical environments, to build trust on the systems that utilize these DNNs.

A number of verification techniques have been proposed to validate the reliability of DNNs in safety-critical settings [**b4**, **b5**], including those based on abstract interpretation, branch and bound, etc. TODO: cite However, since DNN verification is NP-Hard TODO: cite, verification techniques for DNNs often face scalability issues. Our work is based on counterexample guided abstraction refinement [2] and aims to alleviate this issue by reducing the verification problem for a large *concrete* DNN $\mathcal{N}$ into an over-approximate query involving a smaller *abstract* DNN $\mathcal{N}'$. This is done by *merging* groups of neurons belonging to the same *merge group* in $\mathcal{N}$ into single neurons in $\mathcal{N}'$. TODO: appeal to later sections via ref. Then, this smaller query may be dispatched using any existing technique, thus extending the scalability of that technique.

While counterexample guided abstraction refinement based on structural abstraction has been proposed [2] TODO: cite others, in several cases these techniques are forced to refine all the way to the original network, resulting in a query that is as complicated as the original verification problem we started with. We posit that this is because the refinement process does not identify which merges should be prioritised based on the semantics of the network, instead removing a single node from a merge group, leaving potentially equally over-approximate merges in the network. This leads to situations where, after several refinement steps, $\mathcal{N}'$ has a large number of singleton merge groups, leading to a large yet over-approximate abstract network. TODO: refer to later sections with eg demoing this.

In this work we utilise measures of similarities between neurons explored in TODO: cite, check Jan Kretinsky's and others works to build a partial order of possible merges in each layer capturing the priority each merge should get during the refinement process. This allows us to use the information from a spurious counterexample to cut a merge group in a manner that eliminates the merge operation that has been estimated to contribute most to the over-approximation. This avoids the proliferation of singleton merge groups in $\mathcal{N}'$, and leads to a smaller and a tighter abstraction.

We describe a way to efficiently implement this partial order based refinement procedure to be able to iterate through potential refinements with minimal computational overheads. We do this by creating tree-based data structures that allows us to pre-compute a significant portion of the relevant information at the beginning of the refinement loop. Then, refinement operations can be performed via very quick index manipulations. This allows us to iterate through and screen several possible refinements fast before committing to a solver call.

2 Background

An example of Elboher's Refinement where we get a lot of singletons

2.1 Neural Network

We conceptualize neural networks as directed graphs comprising three types of layers: the input layer, intermediate hidden layers, and the output layer. Our focus is on verifying feed forward networks, where neuron values are computed

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

based on preceding layer neuron values. Neurons in our network are denoted as n_i^j, where 'i' signifies the neuron number in layer 'j'. The weight matrix between layers '$j-1$' and 'j' is denoted as '$W_{j-1,j}$', and the bias matrix for layer 'i' is denoted as 'B_i'. The value of the 'i^{th}' neuron in layer 'j' is represented by 'v_i^j', and 'V_j' encompasses all such 'v_i^j' values for layer 'j'.

The computation of 'V_j' involves applying an "activation function (ϕ)" to the "weighted sum":

$$V_j = \phi(W_{j-1,j} \cdot V_{j-1} + B_j).$$

We confine our activation function to the Rectified Linear Unit (ReLU), which can be expressed as $V_j = \max(W_{j-1,j} \cdot V_{j-1} + B_j, 0)$.

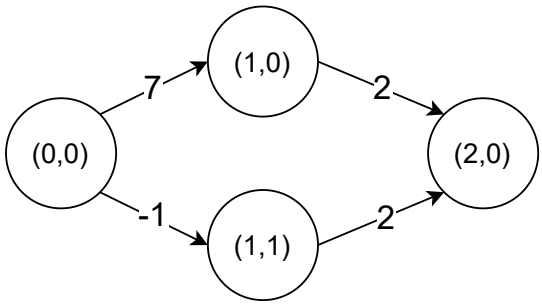


Figure 1. A Simple Neural Network

Figure 1 depicts a feed forward neural network comprising four neurons: n_0^0, n_0^1, n_1^1, and n_0^2. The weight matrix between layer 0 and layer 1, denoted by $W_{0,1}$, is $[7,-1]^T$, and the bias matrix for layer 1, denoted by B_1, is $[0,0]^T$. If we assign a value of 1 to the neuron n_0^0 (i.e., $v_0^0 = 1$), then the value of the neuron n_0^1 is $v_0^1 = 7$, as determined by $V_1 = \phi([7,-1]^T \cdot [1] + [0,0]^T) = [7,0]^T$.

2.2 Neural Network Verification

In the context of neural network verification, we typically engage with a neural network, posing a "*satisfiablility*" query to validate or refute a property. The query $\mathcal{P}$ is structured as a triple, $\mathcal{P} = \langle \mathcal{N}, \kappa, \lambda \rangle$. Here, $\mathcal{N}$ denotes the neural network as previously mentioned, while κ and λ represent properties on the input and output neurons, respectively.

κ is defined as a conjunction of linear constraints applied to the input neurons. Correspondingly, λ is also expressed as a conjunction of linear constraints targeting the output neurons. Our objective is to verify boolean properties formulated as $\forall x \in X,\ y < c$. However, for the purpose of assessing the validity of these properties, we transform our $\lambda(y)$—the conjunction of constraints on our output variables—into the negation of $y < c$, represented as $y \geq c$, where y denotes the output of the neural network. To achieve this transformation, we introduce additional neurons into the network, resulting in a modified network structure with a singular output neuron. This transformation of the Neural Network $\mathcal{N}$ proves advantageous for subsequent Inc/Dec classification, which is described in the subsequent section 2.3.1.

Our query is constructed as a boolean formula $\mathcal{P}(x) \equiv \kappa(x) \wedge \lambda(\mathcal{N}(x))$. Since we are looking for validity we use the negation of the output property and seek an 'UNSAT'. The solver is presented with the formula $\exists x \in X, \mathcal{P}(x)$, where X is the domain of the input. If our solver responds with 'SAT' to this query, it indicates that we have identified a counterexample, and our property does not hold. Conversely, an 'UNSAT' result signifies the validity of our original property.

2.3 Counter-example guided abstraction and refinement

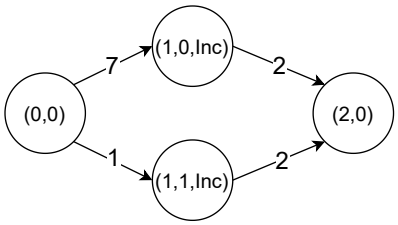
To aid and expedite the verification queries, we employ a strategy to simplify our neural network by reducing the number of neurons. Our approach involves constructing an over-approximate network, denoted as $\mathcal{N}''$, where the output value, $\mathcal{N}''(x)$, consistently exceeds the value computed by the original network, $\mathcal{N}(x)$, i.e ($\forall x \in X,\ \mathcal{N}''(x) \geq \mathcal{N}(x)$), where X is our input domain. This simplification is undertaken because our property which is expressed as $\forall x \in X,\ \mathcal{N}(x) < c$ can then be proven by determining the unsatisfiability of $\exists x \in X,\ \mathcal{N}''(x) \geq c$. If the answer to this question is 'Yes,' then it implies that $\forall x \in X,\ \mathcal{N}(x) < c$ holds, as evidenced by ($\forall x \in X,\ \mathcal{N}''(x) \geq \mathcal{N}(x)$).

2.3.1 Increment Decrement Splitting.

To aid in the process of constructing such an over-approximate network we begin with the creation of an equivalent network, denoted as $\mathcal{N}'$, ensuring that $\mathcal{N}'(x)$ equals $\mathcal{N}(x)$ for all x. To build this equivalent network, we perform an 'Increment/Decrement' splitting of neurons in our original network $\mathcal{N}$. Upon reflection, it became apparent to us that a two-class classification was more fitting for our needs, deviating from the initially recommended four-class classification outlined in the original work by [**b2**]. Neurons are categorized as 'Increment (Inc)' if increasing their values increases the output neuron's value, and as 'Decrement (Dec)' if decreasing their values achieves the same result.

Algorithm 1 split_Inc_Dec

1: Initialize M, the set of nodes that are marked, to {out_node} and mark out_node as **Inc**.
2: Let R be the set of nodes with successors in M.
3: Define sign(u,v) as the sign of the weight from the node u to v.
4: Define Class(v) as 1 when v is marked as **Inc**, and -1 otherwise.
5: **while** node $\notin M$ and node $\notin$ input_nodes **do**
6: Pick a node u such that $u \in R - M$
7: Suppose u feeds into $x_1, x_2, x_3, ..$ and $y_1, y_2, y_3..$ where $\text{sign}(u, x_i) = \text{Class}(x_i)$ for all x_i, $\text{sign}(u, y_i) \neq \text{Class}(y_i)$ for all y_i.
8: Split u to u_1 and u_2, where u_1 feeds into all the x_i and u_2 feeds into all the y_i.
9: Mark u_1 as **Inc** and u_2 as **Dec**
10: Insert u_1 and u_2 into M and their predecessors into R
11: **end while**



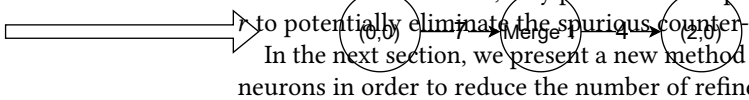
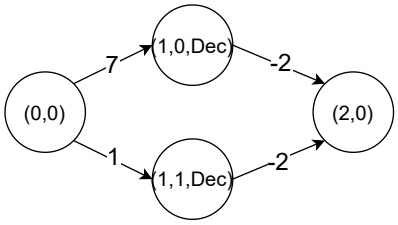


Figure 2. Merging of Increment Neurons



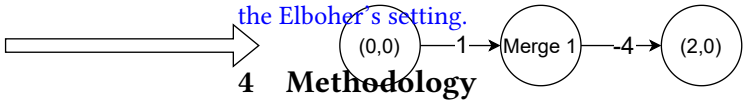


Figure 3. Merging of Decrement Neurons

2.3.2 Abstraction. The increment/decrement splitting of the neural network $\mathcal{N}$ to the new network $\mathcal{N}'$ changes the structure of the neural network but does not change the value of the output neuron. If we want to construct a neural network $\mathcal{N}''$ which over-approximates the value of the original network $\mathcal{N}$ i.e $\forall x \in X,\ \mathcal{N}''(x) \geq \mathcal{N}(x)$, then we perform the following steps:

1. When merging two *Increment Neurons*, discard one of those neurons and replace the incoming weight by maximum of all the incoming weights and the outgoing weight by summation of all the outgoing weights.
2. When merging two *Decrement Neurons*, discard one of those neurons and replace the incoming weight by minimum of all the incoming weights and the outgoing weight by summation of all the outgoing weights.

2.4 Refinement

After merging the neurons, we introduce an over-approximation into the new network $\mathcal{N}''$. And since $\mathcal{N}''(x) \geq \mathcal{N}(x)$, there might exist a x, for which, $\exists x \in X,\ \mathcal{N}''(x) \geq c$ but $\nexists x \in X,\ \mathcal{N}(x) \geq c$. Therefore, if a counter-example 'β' returned is not a valid counter-example, which means, $\mathcal{N}''(\beta) \geq c > \mathcal{N}(\beta)$, then, we must reverse some of the merges performed previously to get a new network which mitigates the extent of the over-approximation for us to get a valid counter-example or to prove that the original property is valid.

In [**b2**], the authors employed a Counterexample-Guided Abstraction Refinement (CEGAR) approach to eliminate the spurious counter-examples. They utilized a counter-example (β) to identify a neuron n_i^j that had been merged into the node 'r' for refinement. The authors then computed the value ω which was equal to $|v_i^j(\beta) - r(\beta)| \cdot |w_{n_{i-1,k},n_{i,j}} - w_{n_{i-1,k},r}|$, where $v_i^j(\beta)$ denotes the value of $n_{i,j}$ at the counter-example β, $r(\beta)$ denotes the value of the representative neuron r at the counter-example β, $w_{n_{i-1,k},n_{i,j}}$ denotes the weight between a neuron $n_{i-1,k}$ and a neuron $n_{i,j}$. If this value ω was over a recommended amount ϵ, they proceeded to separate n_i^j from r to potentially eliminate the spurious counter-example β.

In the next section, we present a new method for merging neurons in order to reduce the number of refinement steps, thereby expediting the verification process.

3 Our Method

In same example, we immediately arrive at a better picture. This picture is equivalent to refining and merging back in the Elboher's setting.

4 Methodology

Our methodology involves two broad steps:

1. Finding a tree structure that represents the order in which neurons should be merged.
2. Using that structure to guide a CEGAR approach in order to help reduce number of refinement steps.

4.1 Tree and Merges:

To establish the merging order of neurons, we create a tree structure wherein leaf nodes represent the original neurons, and non-leaf nodes represent merge groups. The construction of the tree follows a bottom-up approach, prioritizing the merging of similar neurons and delaying the merging of dissimilar ones. Similarity is ascertained through observation vectors. Consider simulating the network with distinct input values and observing the fluctuating values of a particular neuron corresponding to those inputs. The collection of these observed values constitutes an observation vector.

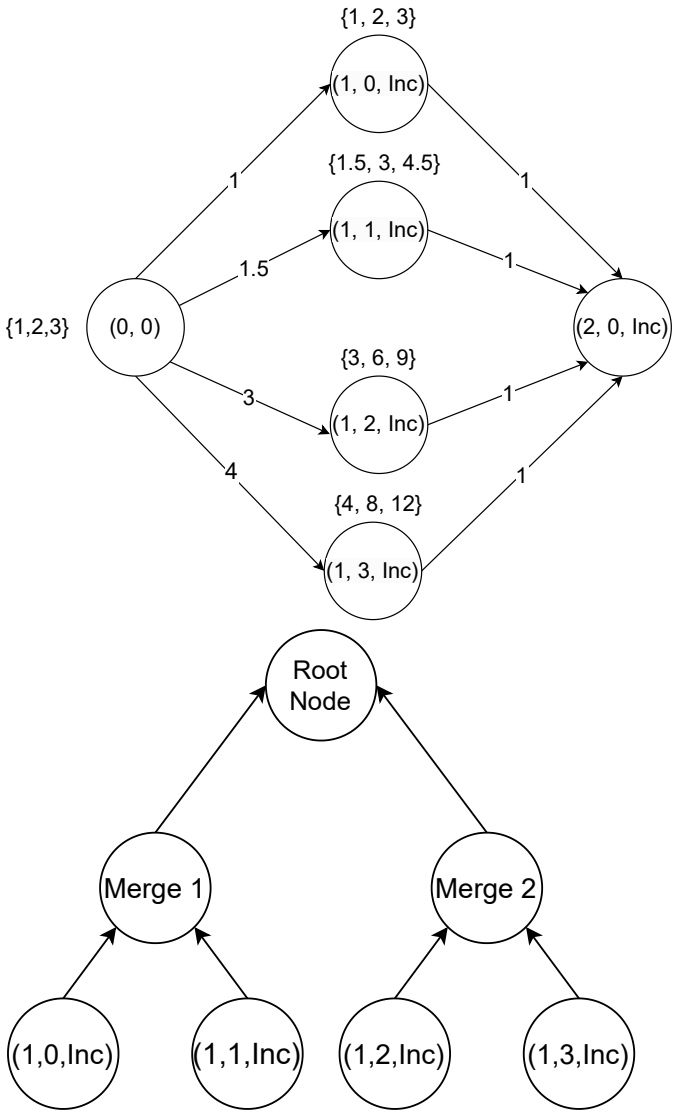


Figure 4. Order Of Merging

In Figure 4, when we input values {1, 2, 3} to the neuron n_0^0 at different time instances, we obtain corresponding values {1.5, 3, 4.5}, forming the observation vector at neuron (n_1^1, Inc). For instance, considering three neurons n_i^w, n_j^w, and n_k^w with observation vectors $v(n_i^w)$, $v(n_j^w)$, and $v(n_k^w)$, the merging sequence adheres to the following conditions:

If $||v(n_i^w) - v(n_j^w)||_2 \leq ||v(n_i^w) - v(n_k^w)||_2$ and $||v(n_i^w) - v(n_j^w)||_2 \leq ||v(n_j^w) - v(n_k^w)||_2$, then n_i^w and n_j^w are initially merged into a representative neuron α, followed by the merging of α and n_k. Here $||v(n_i^w) - v(n_j^w)||_2$ computes the "*Euclidean Distance*" between the observation vectors $v(n_i^w)$ and $v(n_j^w)$.

In Figure 4, for the first layer, the initial merge would involve the neuron (n_0^1, Inc) with (n_1^1, Inc), forming *Merge 1*, then we would combine (n_2^1, Inc) and (n_3^1, Inc), forming *Merge 2*. Finally we combine *Merge 1* and *Merge 2* into the root node.

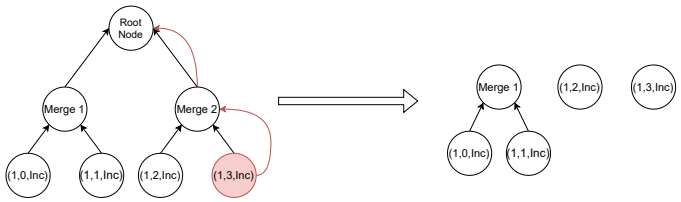


Figure 5. Trees and Cuts

As we progress up the tree, the degree of over-approximation rises. This is due to the increasing difference between observation vectors as we ascend. Therefore, the sub-trees closer to the root are indicative of coarser merges, whereas the ones farther from the root represent finer merges.

4.2 Using Counter-Examples to make cuts in the Tree

We are guided by this tree as a prospective refinement method. Starting with the entire tree where everything is merged. When the solver detects a spurious counterexample 'β', we leverage it to refine the network. This process commences by identifying the "culprit neuron γ" selected for refinement. A "culprit neuron" in a merge group is selected on the basis of how much the neuron contributed to the output. If change in output of neuron changes the value of the output neuron significantly then that neuron is a good candidate for "culprit neuron".

Following this, we reverse all merges dependent on the culprit neuron γ. Therefore, refinement essentially involves finding a cut-point in the tree, precisely where all merges dependent on the culprit neuron γ are undone. Each cut produces a set of trees, the merge groups then consist of neurons in the leaf nodes of the these trees. Therefore finding new merge groups for refinement is therefore just finding a cuts in the tree.

Consider Figure 5, illustrating the merging sequence of neurons (n_0^1, Inc), (n_1^1, Inc), (n_2^1, Inc), and (n_3^1, Inc). If, for instance, the neuron (n_3^1, Inc) is identified as the problematic neuron based on a counter-example, we will reverse all the merges dependent on the (n_3^1, Inc) neuron, including *Merge 2* and the *Root Node* merge. Consequently, after implementing this reversal indicated in Figure 5, our refinement phase will yield three distinct merge groups. The first merge group comprises two neurons, namely (n_0^1, Inc) and (n_1^1, Inc). The second merge group and the third merge have single neurons (n_2^1, Inc) and (n_3^1, Inc), respectively.

4.3 Culprit Neurons

A neuron, denoted as γ, is designated as a culprit neuron within a specific layer when absolute value of the product

of the difference between ($v_{Rep(\gamma)}$ and v_γ) and the effective weight is maximized.

$||(v_{Rep(\gamma)} - v_\gamma)||_2 \cdot |(\textit{effective\_weight})|$

In this context, *Rep* signifies the representative neuron for neuron γ, v_γ represents the value of the neuron γ at counter-example β and *effective_weight* represents the how much does the value of output neuron changes with respect to change in the value of the neuron under consideration, essentially corresponding to the "*gradient*" at that particular counter-example "β".

Algorithm 2 Finding Cuts in the Tree (find_new_merge_groups)

1: $\gamma = \arg\max_i \|v_{Rep(i)} - v_i\|_2 \cdot |\textit{effective\_weight}|$
2: Find a sequence of nodes, $t_1, t_2, t_3, .., t_k$ representing a path from t_1 =root to $t_k = \gamma$.
3: Remove the nodes $t_1, t_2, .., t_{k-1}$ denoting the merges dependent on γ through this path, leading to our connected tree being split into a collection of disconnected subtrees.
4: New merge groups are the leaf nodes in our disconnected graph.

Output New Merge Groups

Algorithm 3 Cluster Merging Algorithm (find_abstraction_tree)

1: Initialize every simulated distance vector as a singleton cluster.
2: Initialize $C = \{v_1, v_2, v_3, ..\}$ as the set of singleton clusters.
3: Initialize a Binary Tree T with leaves as $\{(n_1), (n_2), (n_3), ..\}$ corresponding to $\{v_1, v_2, v_3, ..\}$.
4: Initialize V as a set of visited nodes, empty at first.
5: **function** MergeFunction(u, v)
6: **if** All nodes are classified as **Inc then**
return $\max(u, v)$
7: **else**
return $\min(u, v)$
8: **end if**
9: **end function**
10: **while** $|C| > 1$ **do**
11: $v_j, v_j = \arg\min_{a,b \in C} \|a - b\|_2$
12: Set w = MergeFunction(v_i, v_j)
13: Let nodes from T not in V corresponding to v_i, v_j be m_i and m_j
14: Remove v_i, v_j from C and add w to C.
15: Make $(m_i \cup m_j)$ the parent of (m_i) and (m_j) in tree T
16: Add m_i and m_j to V.
17: **end while**

4.4 Optimality of the Trees

Our objective is to determine the most efficient order for merging neurons, minimizing the introduction of over-approximation at each step. This approach aims to avoid creating networks with excessive over-approximation, which could lead to the generation of spurious counter-examples in response to queries. Opting not to mitigate over-approximation at each step would result in an increased number of refinement steps. This essentially entails making additional solver calls, incurring significant costs to eliminate the spurious counter-examples.

Nevertheless, during the initial merging process (until saturation is reached), the root node "ρ" will exhibit the same level of over-approximation across all conceivable merging scenarios—for all possible tree sequences. Nevertheless, when we descend one level down the tree to explore the children nodes of our original root node ρ for the purpose of identifying a cut for refinement, we discover varying levels of over-approximation manifesting in the root nodes of the resultant sub-trees. These differences are a result of the different merging scenarios pursued to construct those individual trees.

While the optimal tree, representing the optimal merging sequence, can aid in the refinement process by guiding the reversal of merges, finding such an optimal tree poses is extremely challenging. Even when dealing with only 'n' Increment (Inc) neurons that have been merged to saturation, the total number of possible trees is given by $(2n-3)!!$, making the task of determining the truly optimal tree from these options extremely challenging.

Since finding this ideal tree is a challenging task, we employ hierarchical clustering (Algorithm 3) as an approach to approximate and derive such a tree. Initially, we simulate our network using a set of 'k' inputs. Subsequently, we employ cluster analysis on these 'k' points to construct a hierarchical arrangement of clusters. This process initiates with data points corresponding to simulated values (observation values in the observation vector) of a neuron forming their own cluster. The clusters are then systematically combined based on their similarity, thereby generating a hierarchy of clusters. The choice of similarity measure is the "*distance metric*" between clusters. We have used "*Euclidean Distance*" as our distance metric. Given that the data points to perform this hierarchical clustering originate from the values of the simulated neurons, this hierarchical clustering effectively reflects the methodology we employ to merge the neurons.

For example, in Figure 4, we conducted a simulation of our network on three data points. Subsequently, we examined

the observation vectors corresponding to these points. Utilizing the hierarchical clustering algorithm, the initial selection for merging will involve (n_0^1, Inc) and (n_1^1, Inc) because of the fact that their Euclidean distance is minimum. This forms *Merge 1* in Figure 4. The observation vector for *Merge 1* (v_{Merge1}) is the max of the $v((n_0^1, Inc))$ and $v((n_1^1, Inc))$ which is {1.5, 3, 4.5}. For decrement nodes the observation vector would be minimum of the observation vector of the corresponding decrement nodes. The next merging step involved selecting $v((n_2^1, Inc))$ and $v((n_3^1, Inc))$ and merging these two neurons, representing *Merge 2* in Figure 4. The observation vector for *Merge 2* is now {4, 8, 12}. Ultimately, the Merge1 merge group is merged with the *Merge 2* merge group to create the Root Node in our network.

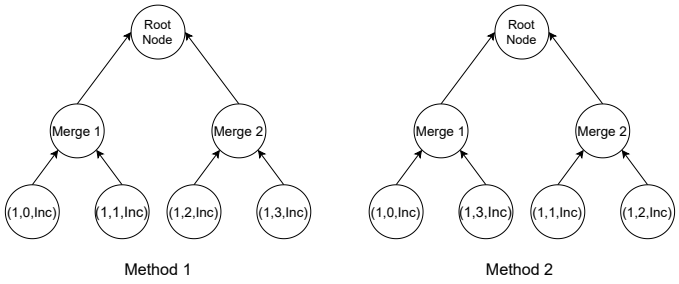


Figure 6. Ways of Merging

This approach of merging neurons based on similarity proves advantageous as it helps in reducing number of refinement steps. For instance, consider the task of checking whether $\forall v_0^0 \in [0, 1]$ implies $v_0^2 < 10$. If we had the neuron (n_3^1, Inc) as the culprit neuron and then if we follow the second merging approach depicted in Figure 6, then we would have been compelled to reverse both *Merge 1* and *Merge 2*. However, by employing the first merging approach, undoing only the *Merge 2* becomes sufficient, resulting in a reduction in the number of refinement steps required.

4.5 Overall Algorithm

Algorithm 4 Overall Algorithm

```
1:  N' = split_Inc_Dec(N)
2:  N'' = abstract_network(N')
3:  simulation_dict = simulate_network(N')
4:  T = find_abstraction_tree(N', simulation_dict)
5:  if verify(N'', κ, λ) is UNSAT then
        return Property Holds
6:  else
7:      Extract counter-example β
8:      if β is not a spurious counter-example then
          return (β, Property Violated)
9:      else
10:         Find culprit neuron γ
11:         merge_groups = find_new_merge_groups(T, γ)
12:         N'' = get_abstract_network(merge_groups)
13:         goto step 5
14:     end if
15: end if
```

5 Experiments

We have implemented our method in python, using a linkage-matrixTODO: cite, maybe from the scipy docs based data structure to store the tree, and have precomputed and cached several operations that may need to be repeated every refinement iteration. This allows us to quickly perform the merge and split operations and calculate the scores, TODO ref earlier sections without having to do (relatively) expensive tree traversal operations in each iteration of the abstraction refinement loop.

Using this implementation, we have performed three sets of experiments to demonstrate the usefulness of our benchmarks. We first use our abstractions to attempt to verify safety properties of the ACAS Xu TODO: citenetworks, where we show that we are able to obtain smaller abstract networks that are sufficiently strong to verify safety properties. However, abstractions may be useful in other domains as well, What? Check Jan Kretinsky, cite, in particular to obtain compressed networks with certain safety guarantees behind them. To demonstrate this, we use our implementation to compress networks trained on the MNIST networks in the second and third set of experiments. This needs to intro our story better

TODO add gh link and exp machine details

5.1 ACAS Xu TODO: cite

These set of experiments still needs to be rerun

In these set of experiments, we have taken the ACAS Xu TODO: citeset of networks TODO: add detailed table and properties and attempt to verify them using a CEGAR TODO: citeapproach. We use our technique to generate the

Benchmarks	% Verified	Avg Reduction	Avg Time
safe	40.404	4.4875	14.198
unsafe	100	55.6071	12.7916

Table 1. Summary of ACAS Xu TODO: citeResultsTODO: Fix formatting

abstraction, and attempt to verify the property on the abstract network using an existing neural network verification solver. If the solver returns a counterexample, and if that counterexample is not spurious, we use that counterexample as a reference point for score calculation TODO: Ref prev sec to select the culprit neuron. TODO: Ref prev section With the culprit neuron selected, we then refine the network.

The results are summarised in table 1. The solver we used was $\alpha\beta - CROWN$ TODO: cite. Note, avg time and network size here doesn't include timeouts. Also, for reduction rate, the original network size is taken to be 600, since the re-merging optimization is not there in this data. With that optimization added, we should compare against 300. Also, the full verification rate is 100% because its not a full set of experiments, this will potentially be a lot lower.

We notice that for the unsafe cases, we are able to refute the properties via a counterexample obtained on a significantly smaller network. But what about the safe cases, what do we say?. In fact, for most networks, we are able to refute based on a counterexample from the fully abstracted network of size 18. This is not there in this small summary table, but is clear from the full data table. Maybe link to that somewhere in the appendix? Note that since we use only two classes in our fully abstracted network as opposed to 4 in [2], the size of this network is significantly smaller. In other networks, we are are able to get a counterexample on a network with size very close to the original network.

For the safe cases, we see that the size of the abstracted networks are large for instances that did not time out. This is in line with what was observed in [2] Can we claim this? It's not immediately clear from their paper, maybe we need to show our recreation of their experiments? Also, this split is not clear from the table. For other instances, the timeout happened due to the solver timing out on the abstract query. While the sizes of the networks on which this to happened are small, does that not say something negative about our abstraction?

Figure 7 shows the network sizes and times of the ACAS Xu TODO: citenetworks as a scatter plot. This includes to and non-to. Note that the time taken here is dominated by the solver call time Should we measure and report this precisely? We see that the time taken by the solver call on these networks is not at all correlated with the network sizes. For instances, for the safe cases, there are several networks of sizes ranging from 350 to 600 $ReLU$nodes all of which time out on the solver call. Similarly, for the unsafe networks,

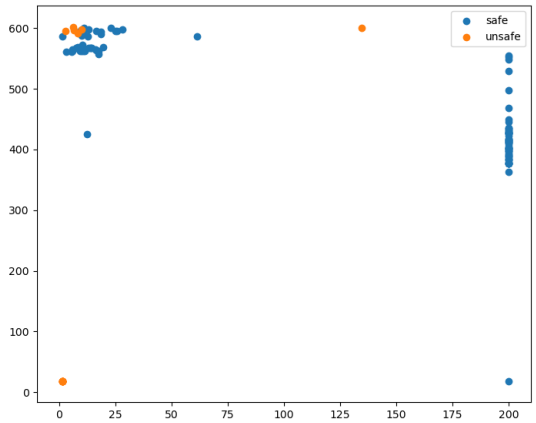


Figure 7. Scatter plot of abstract network sizes vs time in seconds for ACAS Xu TODO: cite

several 600 size networks take less than 25 seconds. In fact, we found that the time taken by the solver call on a particular network is more dependent on the solver (and configuration of the solver) used than on the size of the network. Due to this, in the following experiments, we measure the accuracy of the networks obtained via abstraction instead of making solver calls to determine the effectiveness of the abstraction. Is this okay? Can it be instead argued that the variance is due to the networks themselves, so the abstraction is not necessarily useful? Maybe trying with other configs and showing the variance?

5.2 Robustness on VNNComp MNIST TODO: cite

What about doing these experiments on ACAS? Do pgd then samples

In this section, we start with our abstraction and progressively refine it using our refinement technique, measuring the number of spurious counterexamples at each step. To perform the refinement at each step, we try four methods to perform the culprit neuron selection. The first method 'random' simply chooses a culprit at random, and serves as a baseline. For the other three methods, we choose a number of reference points to calculate scores, and choose the culprit neuron with the highest score. For 'samples', the reference points are random generated input points that satisfy the precondition. For 'pgd', the samples are generated via a PGD TODO: cite attack on the abstract network at each step. For 'samples-pgd', we do some initial refinement steps using the randomly generated samples, when all the samples have been exhausted, we then perform 'pgd'. We use the implementation of 'pgd' inside $\alpha\beta - CROWN$ TODO: citefor our experiments.

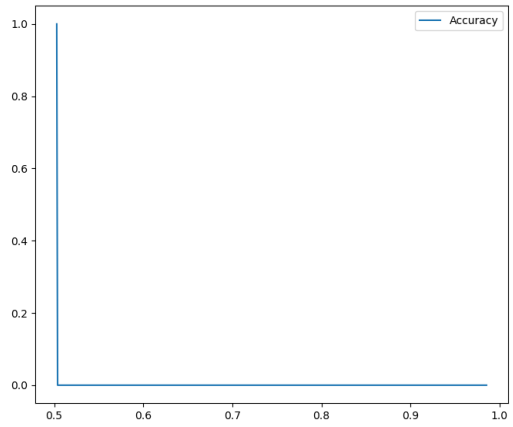


Figure 8. Accuracy vs Reduction Rate plot for VNNComp MNIST TODO: citeof size 2×256 with ϵ-Robustness property. Refinement done via the 'samples' method.

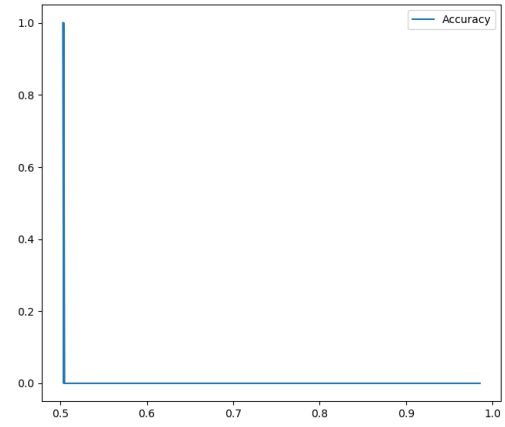


Figure 9. Accuracy vs Reduction Rate plot for VNNComp MNIST TODO: citeof size 2×256 with ϵ-Robustness property. Refinement done via the 'pgd' method.

Figure 8 9 shows a typical plot for these experiments. Here accuracy measures the number of true counterexamples within a random sample drawn from a region satisfying the pre-condition. Should I plot and talk in terms or spurious counterexample rate instead? We see that as we refine the network and the reduction rate reduces (moving right to left along the graph), the accuracy remains close to zero. At some point however, the accuracy jumps up sharply and becomes close to 100%. This indicates that there are a few critical refinement steps that remove almost all spurious counterexamples. This may be due the fact that with epsilon robustness properties that are being explored here, the pre-condition region may be very small, so a single refinement step may change the behavior of the network on that small region in a way that eliminates all possible spurious counterexamples from that region. Is this expl okay? BaB and abstract interp based methods like $\alpha\beta - CROWN$ TODO: citeare successful on such props because of this, discuss this? Also, other graphs are present. For eg, some random sampling graphs show peaks in the middle (2..random), and some pgd graphs are flat lines (4,6..pgd). Discuss these?

Net Size	Cex Method	Reduction	Accuracy	No. Steps
2 × 256	random	49.1%	100%	809
4 × 256	random	49.6%	100%	1778
6 × 256	random	49.7%	100%	2602
2 × 256	samples	50.3%	100%	27
4 × 256	samples	49.9%	100%	648
6 × 256	samples	47.9%	100%	1199
2 × 256	pgd	50.4%	100%	28
4 × 256	pgd	88.6%	0%	14
6 × 256	pgd	83.3%	0%	82
2 × 256	samples-pgd	50.3%	100%	27
4 × 256	samples-pgd	49.9%	100%	648
6 × 256	samples-pgd	47.9%	100%	1209

Table 2. Summary of VNNComp MNIST TODO: citeResults on a single robustness property TODO: Fix formatting Include Times?

Table 2 summarizes these results for multiple VNNComp MNIST TODO: citenetworks and multiple refinement methods. Accuracy here refers to the final accuracy of the best network found when the refinement process stops due to lack of counterexamples. Firstly, we find that 'pgd' performs poorly on the 4 and 6 layer networks, where while the number of steps are relatively low and the reduction rate is high, the accuracy is very low. This is because the $\alpha\beta - CROWN$ TODO: citeimplementation of PGD-attack TODO: citefails to find a spurious counterexample early on in the refinement process for these instances, leading to early termination without ever hitting the accuracy jump.

Apart from these exceptions, the other instances and methods seem to perform comparably on the same network in terms of reduction rate. This is evidence for the fact that, within the limited search space of abstractions defined by our tree, there exists a strong enough abstract network with a good reduction rate, and that it is possible to find this network via a CEGAR TODO: cite-like approach.

Further, we see significant difference in the number of steps taken by 'random' and the other methods in finding the abstract network. This indicates that our heuristics for finding a culprit neuron to base the refinement on indeed shortens the search for the refined network. We also observe that 'samples','samples-pgd' and for the 2 layer network 'pgd'

take similar number of refinement steps, indicating that each method may be taking close to the optimal set of refinement steps. So, although the 'pgd' method incurs a significant time overhead Measure? it does not necessarily provide any benefits.Last two paragraphs okay? Refer to this in intro of experiments section? Also, pgd does not produce improvement (unlike in cleverest). This is prob because global view, and cegarette is about doing an eager attack before solver call. Discuss this? So two potential angles on this, what Ive written vs global view angle.

5.3 Compression with Guarantees on Critical Classes

In several safety-critical applications of DNNas classifiers, there are certain 'critical' classes for which a false negative classification is far more dangerous than a false positive one. For example, for medical diagnosis and collision detection, a false negative is far more dangerous than a false positive.

Safety critical analysis of neural networks is highly sensitive to the size of the network. While several neural network compression techniques exist TODO: cite, they provide limited formal guarantees Talk about Jan? How? connecting the behavior of the compressed network and the original network. Thus, any analysis, including, for example, formal explainability or verification, done on the compressed network cannot easily be lifted to the original network. This lack of guarantees also limits the usefulness of these compression techniques as optimisation steps when deploying a network to a resource-constrained environment.

Our theoretical framework for abstraction allows us to produce compressed networks with the formal guarantee that the abstraction process will not introduce any new false negative. We do this by marking the output neuron n_c corresponding to the critical class in the output layer as inc, and all other neurons in that layer as dec. Then, for any input that the concrete network classified into the critical class, n_c will have the largest value in the output layer, and as it has been marked as inc, it will continue to have the larges value in the output layer for the abstract network as well. Thus, the abstract network will also classify this input as critical.

We demonstrate the effectiveness of our abstraction method as such a compression with guarantees technique via some experiments on VNNComp MNIST TODO: citeand show the results in figure 10. We set the class '0' as the critical class, and use our abstraction technique to obtain a compressed network. Then, we use the 'samples' method described previously to progressively refine the network (moving right to left) , plotting the accuracy, false/true positive/negative rates as we go along.

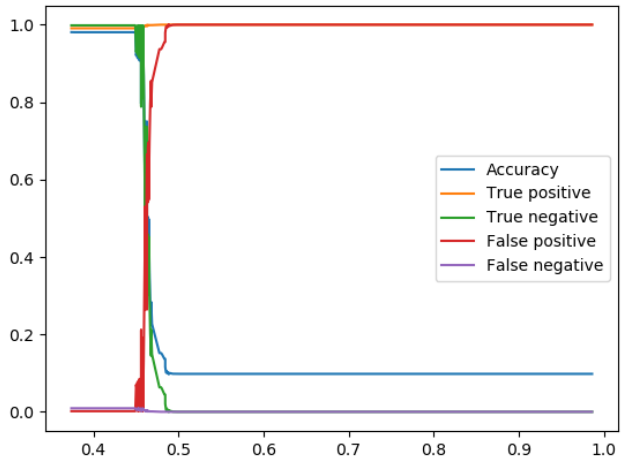


Figure 10. Accuracy and True/False Positive/Negative Rates vs Reduction Rate plot for VNNComp MNIST TODO: citeof size 2×256 with critical class 0. Refinement done via the 'samples' method.

References

[1] G. Reza Djavanshir, Xinrui Chen, and Wenhao Yang. "A Review of Artificial Intelligence's Neural Networks (Deep Learning) Applications in Medical Diagnosis and Prediction". In: *IT Professional* 23.3 (2021), pp. 58–62. DOI: 10.1109/MITP.2021.3073665.

[2] Yizhak Yisrael Elboher, Justin Gottschlich, and Guy Katz. "An Abstraction-Based Framework for Neural Network Verification". In: *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I*. Ed. by Shuvendu K. Lahiri and Chao Wang. Vol. 12224. Lecture Notes in Computer Science. Springer, 2020, pp. 43–65. DOI: 10.1007/978-3-030-53288-8_3. URL: https://doi.org/10.1007/978-3-030-53288-8%5C_3.