Chapter 1

Introduction

Recent advancements in Deep Neural Networks (DNNs) have unlocked solutions for problems like image recognition and natural language processing which were once deemed intractable. Because of this progress, DNNs are indispensable across diverse fields, including high-stakes domains where precision and reliability are crucial. In particular, DNNs have driven major strides in safety-critical areas: in healthcare, they bolster diagnostic accuracy and predictive analytics [DCY21], while in autonomous vehicles, they enable advanced perception systems that support real-time object detection and decision-making [BTD+16].

While DNNs have achieved remarkable success, they remain vulnerable to adversarial [SZS+14, GSS15, MFF16, KGB17, CKBD18, CW17] and backdoor attacks [CLL+17], which limits their application in safety-critical settings. Formal analysis techniques, including verification [KBD+17, SGPV19a, ZWC+18, WZX+21, EGK20] and explainability techniques [MI22, BK23], are typically used to understand and evaluate DNN trustworthiness. However, for the scalability of these techniques, the size of the network continues to be a limiting factor. For instance, the DNN verification problem is known to be NP-complete [KBD+17].

Abstraction-refinement is a well-studied strategy that can help verification techniques scale better. Instead of verifying the original, typically much bigger, network $\mathcal{N}$, the idea is to work with a smaller, abstract, network $\mathcal{N}'$, which may be easier to verify. This abstraction is constructed such that if $\mathcal{N}'$ satisfies a given property, then so does $\mathcal{N}$ [EGK20]. Structural abstraction techniques [EGK20, CEBK23, ZZC+22] build this abstract network $\mathcal{N}'$ by merging groups of neurons in $\mathcal{N}$ into single neurons, based on certain features of the network such as the weights and the biases, while ensuring that the properties proven on $\mathcal{N}'$ would also hold for $\mathcal{N}$.

If $\mathcal{N}'$ fails to meet the specification, the verifier returns a counterexample, which may be a true counterexample, indicating a real violation in $\mathcal{N}$, or a spurious one arising from the *over-approximation*. In the latter case, $\mathcal{N}'$ needs to be refined, i.e. made more precise (and possibly bigger), and the verification check is repeated on the refined network. This process continues until the property is proven to hold for some abstraction $\mathcal{N}^*$, and therefore we

know that it holds on $\mathcal{N}$ as well, or a *real* counterexample is found to show that $\mathcal{N}$ violates the property. It is noteworthy that the iterative refinement may end up with the original network in the worst case, and if the verification technique fails to scale on the original network, we may not get a conclusive result in the end.

One way to guide the refinement process is to refine $\mathcal{N}'$ by using the spurious counterexample. This *counterexample-guided* abstraction-refinement (or, CEGAR, in short) is known to be an effective refinement approach in formal verification [CGJ+00]. The authors of [CGJ+03] applied this CEGAR approach specifically to neural network verification. Their method begins with a very coarse abstraction $\mathcal{N}_\top$, in which all neurons within each layer are initially merged. If $\mathcal{N}_\top$ generates a spurious counterexample—i.e., one that does not hold in $\mathcal{N}$ —it is used to selectively reverse certain merges, refining the abstraction until the counterexample is eliminated. This yields a more precise abstract network $\mathcal{N}'$ for re-verification. The process iterates until a network $\mathcal{N}^*$ is obtained on which the property can be conclusively proven or disproven.

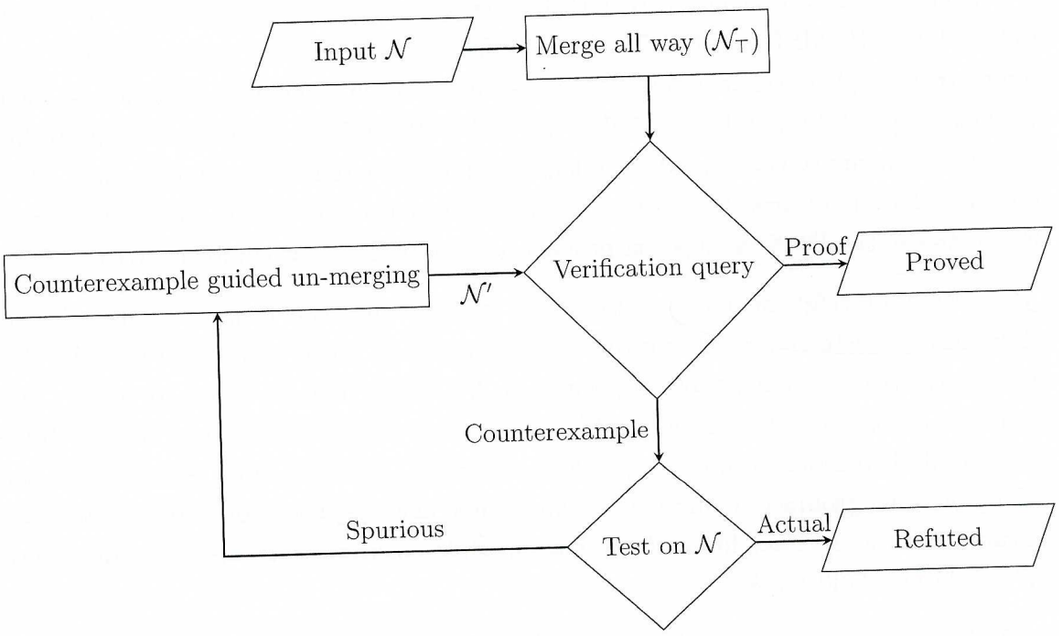


Figure 1.1: CEGAR flow

Although structural abstraction and counterexample-guided refinement techniques improve the scalability of neural network verification, they do not consider the network's *global*

semantic behavior, which can lead to needlessly large abstractions. Here, semantic behavior refers to the functional similarity of neurons—e.g., neurons that produce similar values across a wide range of inputs. Abstractions based purely on syntactic features such as weights and biases may overlook these similarities, resulting in suboptimal merging. Furthermore, since each refinement of $\mathcal{N}'$ is driven solely by a single spurious counterexample, the process lacks a global view of the network's behavior which can lead to a sequence of ineffective refinement choices, restricting the search to suboptimal abstraction regions.

Although neural network compression techniques [CWZZ17] and semantic abstraction methods [AHKM20, CKM23] can achieve substantial reductions in network size by taking into account the behavior of the network's *global semantic information*, compression methods [CWZZ17] are heuristic-based and lack formal guarantees, which means they do not maintain a rigorous connection with the original network $\mathcal{N}$. On the other hand, semantic abstraction methods [AHKM20, CKM23] do offer some formal guarantees, but these are limited. For instance, clustering-based approaches [AHKM20] are limited to lifting specific bound propagation proofs from $\mathcal{N}'$ to $\mathcal{N}$, while linear combination methods [CKM23] provide bounded guarantees on the difference in behavior between $\mathcal{N}'$ and $\mathcal{N}$ for a finite subset of the input space. Therefore, without strong formal guarantees connecting the behaviors of $\mathcal{N}'$ and $\mathcal{N}$, the results of property verification obtained on $\mathcal{N}'$ cannot be reliably extended to prove properties on $\mathcal{N}$.

To address these limitations, we adopt a global perspective to create better abstract networks by integrating the network's *semantic* information with merge operations from structural abstraction techniques. By splitting and labelling the neurons *inc* and *dec*, and by limiting merges to only those involving similarly labelled neurons, as outlined in [EGK20], we establish a clear formal relationship between the behaviors of $\mathcal{N}$ and $\mathcal{N}'$ through syntactic constraints. To account for the overall semantic behavior, we introduce a metric that quantifies the semantic closeness of neurons within the same layer. Using this metric, we construct a tree of merge operations that reflects the relative impact of each merge on the abstraction's quality. We propose a CEGAR framework that utilizes this tree to guide the refinement process to keep semantically similar neurons merged. This strategy allows us to preserve more merges, reduce over-approximation and generate a smaller $\mathcal{N}'$, all while maintaining robust formal guarantees with $\mathcal{N}$. To show how effective our framework is, we experimented with the ACASXu [OPM+19, KBD+17] networks. We discovered that we can create networks that are smaller than those that are created by current methods while still being strong enough to verify the desired properties.

Even though we adopt a more global perspective of the network's behavior to inform our CEGAR process, we found that our approach still relies on the same merge and unmerge operations as those in [CGJ+03, EGK20, CEBK23], which we found often lacked the expressiveness necessary to create more optimal networks. To address this issue, we parameterize the construction of the abstract network with continuous real-valued parameters. This allows us to utilize gradient descent to explore a larger landscape of potential abstractions, considering global information to avoid confining the search to suboptimal regions. Additionally, our parameterization enables the representation of more generalized abstractions than existing methods, allowing us to uncover better abstractions than were previously achievable.

1.1 Outline of the thesis

The rest of this thesis is organized as follows.

- **Chapter** 2 examines the limitations of structural and semantic abstraction techniques, which motivates our current work.
- **Chapter** 3 explains how we utilize semantic information from the network to guide the refinement process, resulting in improved abstract networks.
- **Chapter** 4 examines the limitations of CEGAR approaches and illustrates how parameterizing the construction of the abstract network with continuous real-valued parameters enables us to achieve better abstractions than previous methods.
- **Chapter** 5 discusses the closely related work.
- **Chapter** 6 summarizes the thesis work, discusses its limitations, and lists down possible directions of future work.

Chapter 2

Background

2.1 Neural Networks

An input layer, an output layer, and one or more intermediate layers—known as hidden layers—make up a neural network. A collection of computational units known as neurons make up each layer of the network (denoted by $\mathfrak{n}_{(\mathfrak{i},\mathfrak{l})}$, where $\mathfrak{l}$ refers to the layer number and $\mathfrak{i}$ indicates the index of a neuron within a layer). Neurons are interconnected through directed edges that carry weighted inputs.

Each neuron in a feedforward neural network computes the *weighted sum* of its inputs (also called pre-activation). It sends the outcome of this *pre-activation* to the following layer after applying a non-linear *activation function*. Formally, the pre-activation value $\mathfrak{z}_{(\mathfrak{i},\mathfrak{l})}(\boldsymbol{x})$ is calculated as follows for each neuron in the layer $\mathfrak{l}$ that receives an input vector $\boldsymbol{x} = [\mathfrak{x}_1, \mathfrak{x}_2, \ldots, \mathfrak{x}_n]^\top \in \mathbb{R}^n$, with a weight vector $\boldsymbol{w} = [\mathfrak{w}_1, \mathfrak{w}_2, \ldots, \mathfrak{w}_n]^\top \in \mathbb{R}^n$ and a bias term $\mathfrak{b} \in \mathbb{R}$:

$$\mathfrak{z}_{(\mathfrak{i},\mathfrak{l})}(\boldsymbol{x}) = \boldsymbol{w}^\top \boldsymbol{x} + \mathfrak{b} = \sum_{j=1}^{n} \mathfrak{w}_j \mathfrak{x}_j + \mathfrak{b}$$

We then apply the activation function σ, which is non-linear in its nature, such as sigmoid, ReLU, or tanh, on $\mathfrak{z}_{(\mathfrak{i},\mathfrak{l})}$ to produce the neuron's output:

$$\mathfrak{v}_{(\mathfrak{i},\mathfrak{l})}(\boldsymbol{x}) = \sigma(\mathfrak{z}_{(\mathfrak{i},\mathfrak{l})}(\boldsymbol{x}))$$

We use the Rectified Linear Unit (ReLU) as our activation function, defined by:

$$ReLU(x) = \max(0, x)$$

in our work. That is, if the input is positive, the output is equal to it; if not, it is zero. In this work, we only look at networks where the activation function is applied only to all

hidden layers, while the output layer remains linear (i.e., no activation is applied at the output).

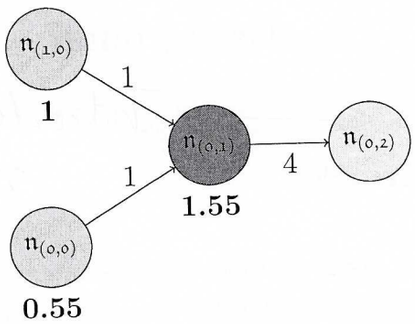


Figure 2.1: A Feedforward Neural Network

For example, consider an input vector $\boldsymbol{x} = [0.55, 1]^\top$ passed into a network where neuron $n_{(0,1)}$ receives inputs from both $n_{(0,0)}$ and $n_{(1,0)}$ with weights $w_1 = w_2 = 1$ and no bias. The pre-activation and activation are computed as follows:

$$z_{(0,1)}(\boldsymbol{x}) = 1 \cdot 0.55 + 1 \cdot 1 = 1.55 \Rightarrow v_{(0,1)}(\boldsymbol{x}) = ReLU(1.55) = 1.55$$

Similarly, for output neuron $n_{(0,2)}$ with weights $w_1 = 4$:

$$z_{(0,2)}(\boldsymbol{x}) = 4 \cdot 1.55 = 6.2$$

2.2 Formal Analysis of Neural Networks

Several techniques and methods have been explored to enhance the reliability and trustworthiness of DNNs which are used in safety-critical environments through formal analysis. These efforts include verifying DNNs against specific safety properties [KBD+17, EGK20, SGPV19a, CEBK23, ZZC+22, OBK22, AHKM20, CKM23], providing formal explanations for DNNs behavior [BK23, MI22], and defending against backdoor attacks [PS22]. To ensure formal guarantees for the analyses conducted, all of these techniques rely on making *neural network queries.*

These neural network queries take the form $(P, \mathcal{N}, Q)$ and inquire whether, given a precondition P on a neural network's inputs and a postcondition Q on the output, for

all inputs $\boldsymbol{x}$ to $\mathcal{N}$ that satisfy property P, does the output $\mathcal{N}(\boldsymbol{x})$ also satisfy Q. This relationship is expressed as follows:

$$\forall \mathbf{x}, P(\mathbf{x}) \rightarrow Q(\mathcal{N}(\mathbf{x})).$$

We adopt an assumption similar to [EGK20, KBD+17, RHK18] that the neural network has only one output neuron y, and Q is an upper bounding property of the form $y \leq c$ where c is some constant. This assumption is not particularly restrictive, as many commonly studied properties—including those involving complex Boolean logic and multiple neurons can be encoded into this framework by appending a small number of auxiliary neurons to the network. Under this formulation, the verification problem reduces to determining whether there exists an input x that violates the property Q; that is, we check the satisfiability of the formula $\exists x, P(x) \wedge (\mathcal{N}(x) > c)$.

2.3 Strong Formal Relationship between $\mathcal{N}$ and $\mathcal{N}'$

A common strategy in formal methods to tackle scalability is *abstraction*. Structural abstraction [EGK20, CEBK23, ZZC+22] creates a smaller, *abstract* DNN $\mathcal{N}'$ by *merging* sets of neurons in $\mathcal{N}$ into a single neuron in $\mathcal{N}'$, while ensuring that:

$$\forall x, P(x) \rightarrow (\mathcal{N}'(x) \geq \mathcal{N}(x))$$

Since Q is of the form $y \leq c$ (Section 2.2), we have:

$$\forall x, (P(x) \wedge \mathcal{N}'(x) \leq c) \rightarrow (\mathcal{N}(x) \leq c)$$

Therefore, this notion of formal connection allows one to abstract the larger $\mathcal{N}$ to a smaller $\mathcal{N}'$, dispatch the easier query $(P, \mathcal{N}', y \leq c)$ using a solver call, and argue that the original query holds [EGK20, CEBK23, ZZC+22]. This immediately makes such an abstract network $\mathcal{N}'$ useful for accelerating several formal analysis techniques. Therefore, in this work, we focus on developing a framework that produces abstract networks that maintain this formal connection.

2.4 Notation

Neurons within our network are denoted as $\mathfrak{n}_{(i,l)}$, where i indicates the neuron number in layer l. The function that calculates the neuronal value $\mathfrak{n}_{(i,l)}$ given an input $\boldsymbol{x}$ is represented as $\mathfrak{v}_{(i,l)}(\boldsymbol{x})$. Additionally, the function $o_{(i,l)}$ takes a list of input vectors $[\boldsymbol{x_1}, \ldots, \boldsymbol{x_N}]$ and returns a vector $[\mathfrak{v}_{(i,l)}(\boldsymbol{x_1}), \ldots, \mathfrak{v}_{(i,l)}(\boldsymbol{x_N})]$ for a specific neuron $\mathfrak{n}_{(i,l)}$. In our network, the abstract neurons are represented by $\mathfrak{a}_{(i,l)}$, where i represents the index of the neuron in layer l.

2.5 Syntactic Neural Network Splitting and Merging

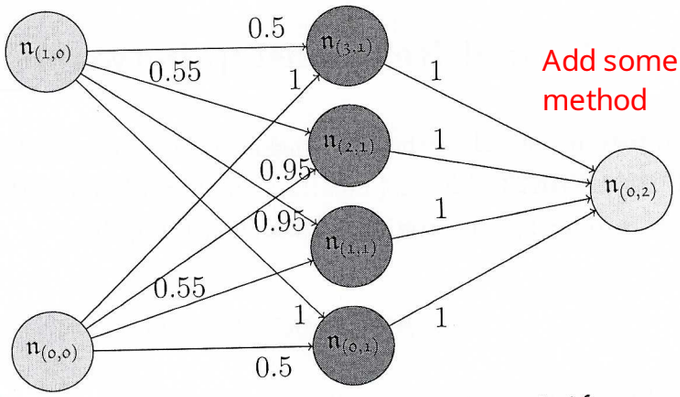


Figure 2.2: A Simple Neural Network $\mathcal{N}$

Given a neural network, its neurons can be partitioned into duplicate copies labelled as *inc* or *dec* [EGK20]. This partitioning is constructed such that an increase in the value of a neuron labelled *inc* leads to a corresponding increase in the network's output. Similarly, a decrease in the value of a neuron labelled *dec* also leads to an increase in the output (see Algorithm 1). While [EGK20] performs a *pos-neg* labelling before the *inc-dec* labelling, it is possible to forgo *pos-neg* and compute the *inc-dec* labels directly. This direct approach is more efficient as it avoids doubling of neurons [CAK+22, LXS+22, LXS+24]. The network that is produced after *inc-dec* labelling is equivalent to the original network.

Algorithm 1 split *inc dec*

Initialize M[1] = output and mark output as *inc*.
Let the nodes with successors in M be denoted by R.
Define sign(u,v) as the sign of the weight from the node u to v.
Define class(v) as 1 when v is marked as *inc*, and -1 otherwise.
while node $\notin M$ and node $\notin$ input_nodes **do**
    Pick a node u such that $u \in R$, $u \notin M$
    Let u be connected to $x_1, x_2, \ldots$ and $y_1, y_2, \ldots$, such that $\text{sign}(u, x_i) = \text{class}(x_i)$ for every x_i, and $\text{sign}(u, y_i) \neq \text{class}(y_i)$ for every y_i.
    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.
    Mark u_1 as *inc* and u_2 as *dec*
    Insert u_1 and u_2 into M and their predecessors into R
end while

Next, we *merge* neurons that share the same label. Specifically, for neurons labelled *inc* (*dec*), we consolidate their incoming edges from a common source neuron in the previous layer by retaining the edge with the maximum (minimum) weight. Similarly, outgoing edges to a common neuron in the subsequent layer are replaced with a single edge whose weight is the sum of the original edges in the *inc* (*dec*) case. Because we take the maximum (minimum) for the *inc* (*dec*) label, the value of the corresponding neuron increases (decreases), which in turn causes the output neuron's value to increase as well—thereby ensuring that the abstraction remains sound [EGK20].

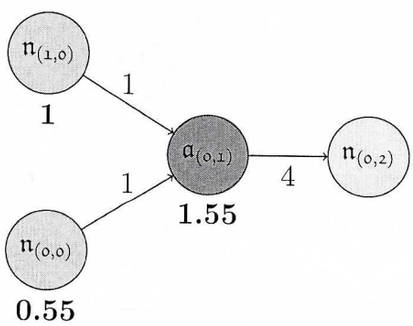
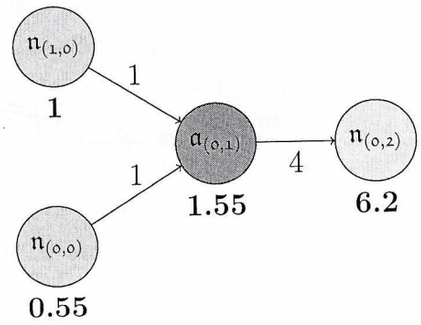


Figure 2.3: Fully Merged Network

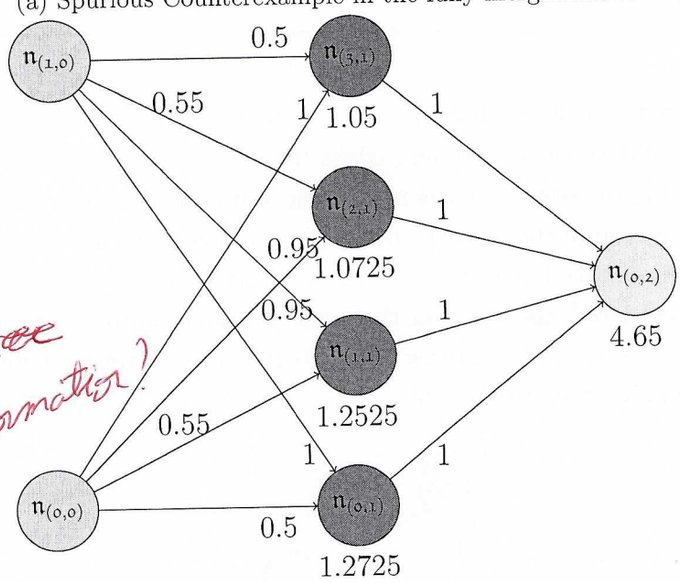
For example, in the network shown in Fig. 2.2, the output neuron is classified as *inc*.

[1]The set of nodes that are marked.

Consequently, the middle-layer neurons are also classified as *inc*, as increasing their values raises the value of the output neuron. Since all these neurons are classified as *inc*, they can be merged, producing the network illustrated in Fig. 2.3[2].



(a) Spurious Counterexample in the fully merged network



(b) Values of the Original Network for the spurious counterexample $n_{(0,1)} = 0.55$ and $n_{(1,1)} = 1$

Figure 2.4

Note that this merging process only takes into account the syntactic structure of the network, without incorporating any semantic information.

[2]Abstract neurons are depicted in red, while concrete neurons are shown in blue.

2.6 Syntactic Refinement

Original CEGAR approaches [EGK20, ZZC+22, CEBK23] begin with a *fully merged* network (see Fig. 2.3) to attempt to prove the property. However, this network might not be strong enough to prove or disprove the property, potentially resulting in a spurious counterexample β. For example, with input values $\mathfrak{n}_{(0,1)} = 0.55$ and $\mathfrak{n}_{(1,1)} = 1$, the original network produces an output of 4.65 (see Fig. 2.4b), while the fully merged network yields 6.2 (see Fig. 2.4a). This discrepancy suggests that the network requires refinement by reversing some merges. There are multiple options for unmerging nodes such as $\mathfrak{n}_{(0,1)}$, $\mathfrak{n}_{(1,1)}$, $\mathfrak{n}_{(2,1)}$, or $\mathfrak{n}_{(3,1)}$. In original CEGAR techniques [EGK20, ZZC+22, CEBK23], the refinement process typically involves restoring a single neuron from $\mathcal{N}$ that was previously merged in $\mathcal{N}'$, usually the one with the greatest estimated impact on the spurious counterexample β (the difference between the neuron's value in the concrete network and the corresponding abstract neuron's value in the abstract network is at its maximum, i.e., $|\mathfrak{v}_{(i,l)} - \mathfrak{a}_{(j,l)}|$ is maximized).

So, in the first refinement step, neuron $\mathfrak{n}_{(3,1)}$ is restored because it is estimated to have the greatest impact on the output, i.e., $|1.05 - 1.55|$, as illustrated in Fig. 2.4, resulting in the network shown in Fig. 2.5.

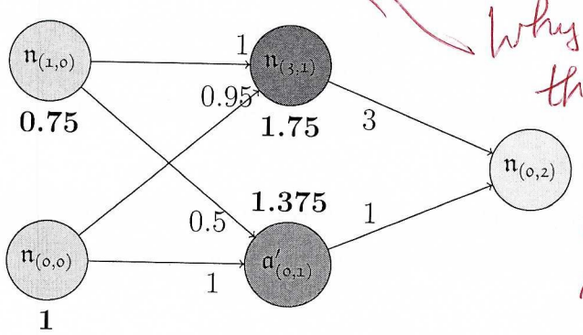


Figure 2.5: Refinement Step 1

However, this network still produces a spurious counterexample. In the next step, neuron $\mathfrak{n}_{(0,1)}$ is restored, resulting in the network in Fig. 2.6. In this new abstract network, neurons $\mathfrak{n}_{(1,1)}$ and $\mathfrak{n}_{(2,1)}$ remain merged despite their semantic dissimilarity (the values they produce differ significantly), while merges between more similar pairs—$\mathfrak{n}_{(2,1)}$ and $\mathfrak{n}_{(3,1)}$, and $\mathfrak{n}_{(0,1)}$ and $\mathfrak{n}_{(1,1)}$—are undone. However, this network also produces spurious counterexamples, necessitating further refinement back to the original network.

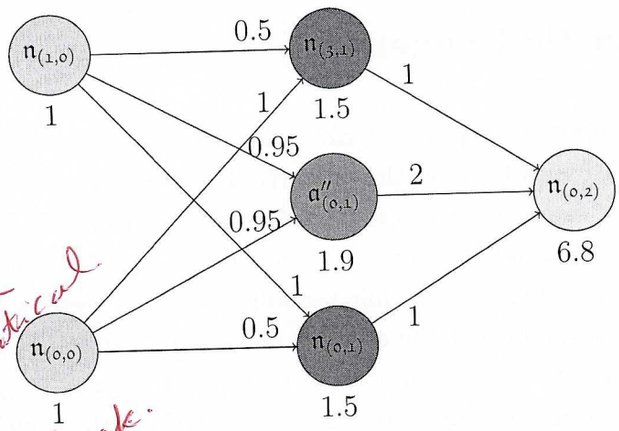


Figure 2.6: Refinement Step 2

This is a related work! Does this discussion provide mathematical foundation to describe your work.

2.7 Semantic Compressions and Abstractions with Empirical Guarantees

The following text reads like related work instead of prelim content

Since the refinement process does not incorporate the semantic behavior of the neurons for selecting which neurons to unmerge, it often results in many neurons being restored that are not merged with any other neurons. This proliferation of *singleton* neurons increases the size of $\mathcal{N}'$. Concurrently, groups of dissimilar neurons may remain merged in $\mathcal{N}'$, which degrades its quality.

Now, several other techniques like [CWZZ17, CKM23, AHKM20] try to leverage *semantic* information, typically extracted through the simulation of the DNNs, to create a smaller $\mathcal{N}'$.

Neural network compression techniques [CWZZ17] aim to construct a smaller network $\mathcal{N}'$ by reducing the size and complexity of the original network $\mathcal{N}$. These methods include parameter pruning and quantization, which eliminate redundant or low-sensitivity parameters [HS92]; low-rank factorization, which approximates weight matrices using decompositions to retain only the most informative components [SKS+13]; and knowledge distillation, where a smaller model is trained to imitate the behavior of a larger one [KOL+20]. However, the relationship between $\mathcal{N}'$ and $\mathcal{N}$ in these approaches is generally characterized empirically.

Similarly, some semantic abstraction techniques like [AHKM20] approximate a concrete

Not clear..

network $\mathcal{N}$ by constructing an abstract network $\mathcal{N}'$ through the merging of neurons with similar input-output behavior, identified using *k-means clustering*. To maintain soundness, they compute an upper bound on the error introduced by abstraction and incorporate it into the interval-bound propagation performed on $\mathcal{N}'$. This allows the propagated bounds to be soundly lifted back to $\mathcal{N}$. However, the resulting interval bounds are often too coarse to verify complex and practically significant neural network properties.

Other semantic abstraction techniques, such as [CKM23], construct $\mathcal{N}'$ by replacing groups of neurons with their linear combinations. These methods provide bi-simulation guarantees that bound the behavioral differences between $\mathcal{N}'$ and the original network $\mathcal{N}$, but only over a finite set of input points. As a result, the trust in the relationship between $\mathcal{N}$ and $\mathcal{N}'$ is inherently empirical, and thus cannot guarantee safety.

Chapter 3

Unifying Syntatic and Semantic Approaches

In Chapter 2, we discussed structural abstraction techniques [EGK20, CEBK23, ZZC+22, OBK22], which convert a large *concrete* DNN $\mathcal{N}$ into a smaller *abstract* DNN $\mathcal{N}'$ by *merging* groups of neurons in $\mathcal{N}$ into single neurons in $\mathcal{N}'$. This merging is performed in a manner that provides concrete guarantees, enabling properties to be proven on $\mathcal{N}'$ and the results to be soundly lifted for proving properties on $\mathcal{N}$. While such techniques have been shown to improve the scalability of verification, they ignore the *semantic* behavior of the network, often resulting in unnecessarily large abstract networks.

In contrast, neural network compression methods [CWZZ17] and semantic abstraction approaches [AHKM20, CKM23] incorporate global *semantic* information about the network's behavior. These approaches can significantly reduce the network size but produce an abstract network $\mathcal{N}'$ that is merely an *approximation* of $\mathcal{N}$, offering only weak or no formal guarantees.

In this work, we integrate semantic information from $\mathcal{N}$ into the structural abstraction process. By using the semantic information to refine the abstraction, we are able to construct a smaller $\mathcal{N}'$ that still maintains strong formal connections to the original network $\mathcal{N}$. To achieve this:

- We introduce a *semantic closeness metric* that quantifies the similarity in semantic behavior between two neurons (see Section 3.1). We use this semantic closeness metric to organize the merge operations into a tree structure, positioning lower-quality merges that involve semantically distant neurons higher up, allowing them to be prioritized for refinement (Section 3.2).
- Leveraging this merge-tree structure, we develop a framework where refinement can be done by making cuts of this merge-tree while still providing concrete soundness guarantees. We demonstrate that the merges preserved during this refinement process are optimal based on the semantic information (Section 3.3). This approach enables us to prevent the restoration of a large number of singleton neurons and allows us to retain higher-quality merge operations (see Section 2.5).
- Using these components, we propose a general CEGAR [CGJ+03] loop-based framework (Section 3.5) that combines syntactic merge operations with semantic behavior.

This framework is able to produce an $\mathcal{N}'$ strong enough not to have any spurious counterexamples while having a much smaller size.

3.1 Semantic Closeness Factor

To guide the semantic abstraction process, we define a *semantic closeness metric* $\mathcal{C}$: $\mathcal{C}(\mathfrak{n}_{(i_1,l)}, \mathfrak{n}_{(i_2,l)})$ is a function that takes as input two neurons $\mathfrak{n}_{(i_1,l)}$ and $\mathfrak{n}_{(i_2,l)}$ in the same layer l, and returns a real number that represents how semantically close the behavior of neurons $\mathfrak{n}_{(i_1,l)}$ and $\mathfrak{n}_{(i_2,l)}$ are. The value returned by $\mathcal{C}$ is smaller for neurons whose semantic behavior is closer. Intuitively, this metric would characterize the semantic behavior of the neurons in layer l relative to each other and prioritize certain merges over others.

The precise definition of this metric can be adapted to the specific application at hand. Notably, our framework is agnostic to the particular semantic metric chosen, as the concrete soundness guarantees hold for any such choice. Inspired by [AHKM20], we chose the semantic closeness metric as the difference between the functions computed by the two neurons: $||\mathfrak{v}_{(i_1,l)} - \mathfrak{v}_{(i_2,l)}||_2$, where $||\cdot||_2$ is the L_2 norm on the space of continuous functions on the input region $P \subset \mathbb{R}^i \to \mathbb{R}$. Generally, inputs to DNNs are typically bounded, and thus P is bounded as well. Since $\mathfrak{v}_{(i_1,l)}$ and $\mathfrak{v}_{(i_2,l)}$ are neuron evaluations, they are Lipschitz continuous [VS18], which guarantees that the L_2 norm is well-defined.

Calculating $||\mathfrak{v}_{(i_1,l)} - \mathfrak{v}_{(i_2,l)}||_2$ exactly, however, is often infeasible, so we approximate it using a sample set of inputs X: $||o_{(i_1,l)}(X) - o_{(i_2,l)}(X)||_2$. In general, X may be selected in various ways, and our framework is agnostic to the sampling method. In our experiments, we use uniform sampling over the input region where P holds. Since the input region is typically an interval, this sampling approach is quite straightforward to do.

We use $\mathcal{C}$ to construct a hierarchical merge-tree structure to prioritize merges, where each leaf node represents an original neuron, and each non-leaf node represents a merge operation. Specifically, a non-leaf node $\mathfrak{mg}_i$ symbolizes an operation merging all the neurons corresponding to the leaf nodes that are descendants of $\mathfrak{mg}_i$. For example, in the upper half of Fig. 3.1, $\mathfrak{mg}_4$ merges $\mathfrak{n}_{(0,1)}$ and $\mathfrak{n}_{(1,1)}$, while $\mathfrak{mg}_6$ merges $\mathfrak{n}_{(0,1)}$, $\mathfrak{n}_{(1,1)}$, $\mathfrak{n}_{(2,1)}$, and $\mathfrak{n}_{(3,1)}$.

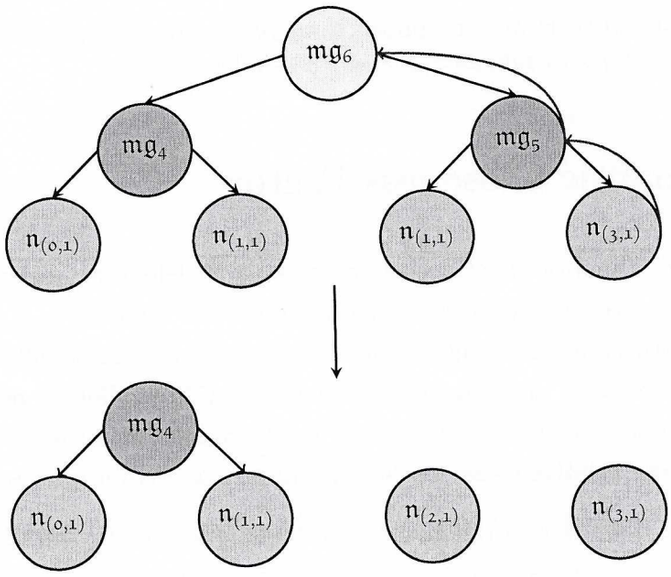


Figure 3.1: Merge-Tree and Cuts

3.2 Merge-Tree

The merge-tree is built using a bottom-up approach. It begins with individual neurons and then greedily performs merge operations that group the most similar neurons, thus deferring merges involving more dissimilar neurons. This process is outlined in Algorithm 2 [1].

We start with an initial structure consisting only of leaf nodes corresponding to original neurons (line 1). At this stage, no merges have been performed, and each node is available for merging, stored in *Cand* (line 2).

As long as there are at least two candidates available for merging, we continue to merge them in a greedy manner in the loop (line 11). At every iteration, we identify the two candidate nodes, $\mathfrak{mg}_i$ and $\mathfrak{mg}_j$, that are the most semantically similar (line 12), and then perform a merge operation on them (lines 13-15).

To determine the semantic closeness of a pair of candidates, $\mathfrak{mg}_1$ and $\mathfrak{mg}_2$, we examine the maximum semantic distance between any neuron involved in the merge process of $\mathfrak{mg}_1$

[1]For our choice of $\mathcal{C}$ (see Section 3.1), Algorithm 2 reduces to hierarchial clustering [Jr.63], enabling us to use existing efficient implementations. However, the general algorithm applies to any choice of $\mathcal{C}$.

and any neuron involved in the merge process of $\mathtt{mg}_2$. This measurement is performed recursively using the *PairwiseMax* function, as outlined in lines 3-10.

This resulting merge-tree captures an optimal sequence of merge operations. As we move upward in the merge-tree, the maximum value of $\mathcal{C}$ between any pair of merged neurons grows, indicating that the imprecision from merging increases. This property is formalized in Section 3.4.

Algorithm 2 Building the Merge-Tree

Input: Neurons $\{\mathtt{n}_{(i_1,l)}, \ldots, \mathtt{n}_{(i_\tau,l)}\}$ with *inc-dec* label, Closeness metric $\mathcal{C}$

1: Initialize G with nodes $\{\mathtt{n}_{(i_1,l)}, \ldots, \mathtt{n}_{(i_\tau,l)}\}$ and no edges.
2: Initialize $Cand = \{\mathtt{n}_{(i_1,l)}, \ldots, \mathtt{n}_{(i_\tau,l)}\}$
3: **function** PAIRWISEMAX($\mathtt{mg}_1, \mathtt{mg}_2$)
4: **if** $\mathtt{mg}_1$ or $\mathtt{mg}_2$ has children **then**
5: Without loss of generality, say $\mathtt{mg}_1$ has children c_1 and c_2.
6: **return** max(PairwiseMax($c_1, \mathtt{mg}_2$), PairwiseMax($c_2, \mathtt{mg}_2$))
7: **else**
8: $\mathtt{mg}_1$ and $\mathtt{mg}_2$ are neurons, **return** $\mathcal{C}(\mathtt{mg}_1, \mathtt{mg}_2)$.
9: **end if**
10: **end function**
11: **while** $|Cand| > 1$ **do**
12: $\mathtt{mg}_{j_1}, \mathtt{mg}_{j_2} = \arg\min_{\mathtt{mg}_1, \mathtt{mg}_2 \in Cand} \text{PairwiseMax}(\mathtt{mg}_1, \mathtt{mg}_2)$
13: Add new node $\mathtt{mg}_{j_3}$ to $\mathfrak{T}$
14: Make $\mathtt{mg}_{j_1}, \mathtt{mg}_{j_2}$ children of $\mathtt{mg}_{j_3}$ in $\mathfrak{T}$
15: Remove $\mathtt{mg}_{j_1}, \mathtt{mg}_{j_2}$ from $Cand$ and add $\mathtt{mg}_{j_3}$ to $Cand$.
16: **end while**
17: G now is a merge-tree, call it $\mathfrak{T}$

Output: Merge-Tree $\mathfrak{T}$

No use of the labels

For example, consider the middle layer in Fig. 2.2. Here, $\mathtt{n}_{(0,1)}$ and $\mathtt{n}_{(1,1)}$ are semantically closest. Thus, in the merge-tree (top half of Fig. 3.1), we first merge these two to get the node $\mathtt{mg}_4$, representing the merge group $\{\mathtt{n}_{(0,1)}, \mathtt{n}_{(1,1)}\}$. At this point, we have three choices for the next merge operation: $\mathtt{mg}_4$ and $\mathtt{n}_{(2,1)}$, $\mathtt{mg}_4$ and $\mathtt{n}_{(3,1)}$, or $\mathtt{n}_{(2,1)}$ and $\mathtt{n}_{(3,1)}$. Since $\mathtt{n}_{(2,1)}$ and $\mathtt{n}_{(3,1)}$ are semantically closer to each other than $\mathtt{n}_{(0,1)}$ or $\mathtt{n}_{(1,1)}$, the algorithm merges $\mathtt{n}_{(2,1)}$ and $\mathtt{n}_{(3,1)}$ to get $\mathtt{mg}_5$. This produces the merge group $\{\mathtt{n}_{(2,1)}, \mathtt{n}_{(3,1)}\}$, which has elements that are semantically closer than the groups $\{\mathtt{n}_{(0,1)}, \mathtt{n}_{(1,1)}\, \mathtt{n}_{(2,1)}\}$ or $\{\mathtt{n}_{(0,1)}, \mathtt{n}_{(1,1)}, \mathtt{n}_{(3,1)}\}$ obtained from following the other two choices. Finally, $\mathtt{mg}_4$ and $\mathtt{mg}_5$ get merged to $\mathtt{mg}_6$, giving us the complete merge-tree.

3.3 Cuts in Merge-Tree and Refinement

In our abstraction refinement loop, we begin with a fully merged network. When we receive a spurious counterexample $\boldsymbol{\beta}$, we aim to refine the network by determining neurons which should remain merged. This decision should be guided by two key factors: optimizing with respect to the network's semantic behavior and attempting to eliminate $\boldsymbol{\beta}$.

The merge-tree produced in the previous Section 3.2 captures the semantic behavior, and we use it to guide the refinement process as follows: Each cut of this merge-tree yields a set of merge-trees, where the groups of neurons we decide to keep merged correspond to the leaf nodes of these subtrees. Therefore, if we restrict ourselves to merging groups of neurons that are optimal with respect to the view of the semantic behavior of the network captured by $\mathcal{C}$, the task of refinement boils down to identifying appropriate cuts in the merge-tree.

To eliminate $\boldsymbol{\beta}$, we identify a *culprit neuron* γ, which is the primary contributor to the $\boldsymbol{\beta}$. The intuition behind this is that γ should remain unmerged with any other neuron, as any over-approximation of γ's behavior is likely to lead to the introduction of $\boldsymbol{\beta}$.

Consequently, our refinement process consists of two steps: first, we identify the culprit neuron γ, and then we determine a cut in the merge-tree that ensures γ is kept separate from all other neurons.

3.3.1 Finding γ

There are several strategies that can be employed to identify the culprit neuron γ, and our framework is agnostic to the specific strategy chosen. In our experiments, we adopted the strategy based on *gradient-guided refinement* as described in [CKM23]. For each potential γ, we calculated the following score and selected the one with the highest value:

$$\|v_\gamma^*(\boldsymbol{\beta}) - v_\gamma(\boldsymbol{\beta})\|_2 \cdot \left|\frac{\delta y(\boldsymbol{\beta})}{\delta v_\gamma}\right|$$

How do one compute this term?

In this equation, $v_\gamma(\boldsymbol{\beta})$ represents the value of neuron γ for the input $\boldsymbol{\beta}$ in the original $\mathcal{N}$, while $v_\gamma^*(\boldsymbol{\beta})$ denotes the value of the neuron into which γ has been merged in our current abstraction. The term $\frac{\delta y(\boldsymbol{\beta})}{\delta v_\gamma}$ refers to the partial derivative of the output y of $\mathcal{N}$ with respect to the value at γ for the input $\boldsymbol{\beta}$.

Prelim needs to discuss gradiant!

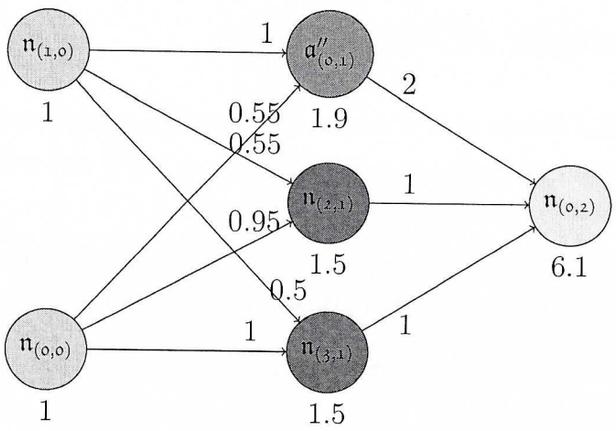


Figure 3.2: Refining by our method: Culprit Neuron is 3

3.3.2 Cutting the Merge-Tree

Our goal is to identify a cut in the merge-tree that keeps γ unmerged with any other neurons while maximizing the number of neurons that remain merged (therefore minimizing the increase in the size of the abstraction). To achieve this, we delete precisely those nodes that are dependent on γ, starting from γ's parent and progressing upward through the merge-tree along the parent links.

In our running example, the culprit neuron is $n_{(3,1)}$. We traverse the merge-tree along the blue edges in Fig. 3.1, reversing the merges mg_5 and mg_6. This results in three merge-trees, corresponding to leaving $n_{(0,1)}$ and $n_{(1,1)}$ merged while undoing the merge of $n_{(2,1)}$ and $n_{(3,1)}$. Consequently, we obtain the abstraction illustrated in Fig. 3.2.

Note that unlike the refinement process used by [EGK20] (see Section 2.5), we maintain merges of semantically similar neurons ($n_{(0,1)}$ and $n_{(1,1)}$) while prioritizing unmerging of semantically dissimilar neurons $n_{(2,1)}$ and $n_{(3,1)}$ (Section 3.3). This avoids the proliferation of singletons and gets a smaller abstraction that is sufficient to prove the property in fewer iterations.

Once we have cut the merge-tree and decided on which neurons to leave merged, the actual merge operation is the exact same as that followed by [EGK20] (Section 2.5). Therefore, we are able to retain concrete soundness guarantees.

3.4 Optimality of Merge-Tree

Notation: Given a (sub)tree $\mathfrak{T}$ of merge operations, we say a neuron $\mathfrak{n}_{(i,l)} \in \mathfrak{T}$ if and only if $\mathfrak{T}$ has a leaf node corresponding to $\mathfrak{n}_{(i,l)}$. That is, $\mathfrak{n}_{(i,l)} \in \mathfrak{T}$ if and only if the merge operations forming $\mathfrak{T}$ involve $\mathfrak{n}_{(i,l)}$ at any point.

The merge-tree $\mathfrak{T}$ produced in Section 3.2 captures an optimal ordering of merge operations with respect to the semantic information in the following sense:

Lemma 1. *Let $\mathfrak{T}_1$ and $\mathfrak{T}_2$ be two sub-trees of $\mathfrak{T}$. Then, we have:*

Are they disjoint?

Are they disjoint? $$\max_{\substack{\mathfrak{n}_{(i_1,l)} \in \mathfrak{T}_1 \\ \mathfrak{n}_{(i_1',l)} \in \mathfrak{T}_1}} \mathcal{C}(\mathfrak{n}_{(i_1,l)}, \mathfrak{n}_{(i_1',l)}) \leq \max_{\substack{\mathfrak{n}_{(i_1,l)} \in \mathfrak{T}_1 \\ \mathfrak{n}_{(i_2,l)} \in \mathfrak{T}_2}} \mathcal{C}(\mathfrak{n}_{(i_1,l)}, \mathfrak{n}_{(i_2,l)})$$

Proof. Using induction on the combined size of $\mathfrak{T}_1$ and $\mathfrak{T}_2$, this lemma can be proved. If a violation of the inequality exists, there may be two cases. In the first case, we have $\mathfrak{n}_{(i_1,l)}, \mathfrak{n}_{(i_1',l)} \in \mathfrak{T}_1'$ where $\mathfrak{T}_1'$ is a strict sub-tree of $\mathfrak{T}_1$. But $\mathfrak{T}_1'$ and $\mathfrak{T}_2$ would then form a violation of the induction hypothesis. The other case directly violates the pairwise maximum condition used in the construction of the merge-tree in Algorithm 2 line 3 in Section 3.2. □

Intuitively, the lemma shows that for any cut in the merge-tree, the maximum difference in the semantic behavior of neurons that have been left merged is less than the maximum difference in the semantic behavior of neurons that have been unmerged. In particular, this implies that, after each refinement step, the groups of neurons that remain merged together are optimal with respect to the semantic behavior of the network.

However, note that our semantic closeness metric fails to say anything about the networks' output for any given input. Thus, although we have optimality with respect to semantic behavior, we are unable to predict the result that making a cut would have on the output for the given spurious counterexamples. Nonetheless, providing some guarantees on the network output for the spurious input would be an interesting direction for future work.

3.5 CEGAR [EGK20, CGJ+03] Loop Framework

We integrate the components discussed so far into a CEGAR [EGK20, CGJ+03] loop. We begin with a fully merged network and then iteratively refine it using spurious counterexamples until we reach an abstraction that eliminates spurious counterexamples. This loop is parameterized by:

- The definition of the semantic closeness factor $\mathcal{C}$.
- The method for selecting the culprit neuron γ.

Note that even though we have suggested specific strategies for each of these elements, our framework is sufficiently flexible to allow for modifications or substitutions based on the application to achieve better performance.

3.6 Experiments

We have developed our framework in Python[2] using the NumPy library for linear algebra operations and the SciPy library for the hierarchical clustering implementation. Our approach employs a linkage-matrix-based data structure, similar to that in SciPy, to represent the merge-tree. Additionally, we have precomputed and cached several operations that are likely to be repeated in each refinement iteration. This design allows us to efficiently perform merge and split operations and compute the scores (see Section 3.3) without incurring the computational cost of merge-tree traversal during every iteration of the abstraction refinement loop.

Using this implementation, we conducted some experiments to test the usefulness of our abstraction technique for verifying neural network queries on the ACASXu [OPM+19, KBD+17] set of networks. We evaluated both the original safety properties from [KBD+17] and the ϵ-robustness properties introduced in [EGK20]. For this purpose, we set up a CEGAR [EGK20, CGJ+03] loop (see Section 3.5) that employed our abstraction technique, utilizing the NeuralSAT [DND24] solver as the underlying tool to handle verification queries on the abstraction.

[2]The complete code, along with the networks and datasets used in our evaluation, can be found at https://github.com/digumx/unified-merges.

We compared our abstraction framework against the existing CEGAR [EGK20, CGJ+03] framework proposed in [EGK20][3] Our timeout was set to 200 seconds for each benchmark instance, and we applied this limit to our technique and the existing approach. We conducted our experiments on a machine equipped with an Intel(R) Core(TM) i7-6700 CPU with 8 cores running at 3.40 GHz, 16 GB of RAM, and Ubuntu 22.04 LTS.

If $\mathcal{N}'$ contains multiple neurons in the same layer with identical incoming edges, these neurons compute the same function and are considered redundant. To optimize our method, we perform a safe *re-merging* of these neurons by adding up the weights of their outgoing edges. Note that this process does not alter the behavior of the abstraction.

Method	No. Safe	No. Unsafe	No. Timeout	Average Size
Ours	121	43	16	335.3
Existing [EGK20]	118	43	19	536.0

Table 3.1: Summary of ACASXu [OPM+19, KBD+17] on original safety properties

Method	Percentage Verified	Average Size
Ours	100%	27.9
Existing [EGK20]	100%	31.5

Table 3.2: Summary of ACASXu [OPM+19, KBD+17] on robustness properties

Tab. 3.1 and Tab. 3.2 summarize the results of these benchmarks. Our findings indicate that, with our framework, we outperform the existing CEGAR [EGK20, CGJ+03] approach on the original safety properties, verifying a greater number of networks as safe, while maintaining performance on the robustness properties.

[3]We used a faithful re-implementation of this framework that strictly follows the procedures outlined in the paper, with two exceptions: we are utilizing a two-class classification as described in [CAK+22, LXS+22, LXS+24], and the call to verify the $\mathcal{N}'$ obtained in each iteration is sent to an instance of the NeuralSAT [DND24] solver as opposed to Marabou [KBD+17, KHI+19, WIZ+24].

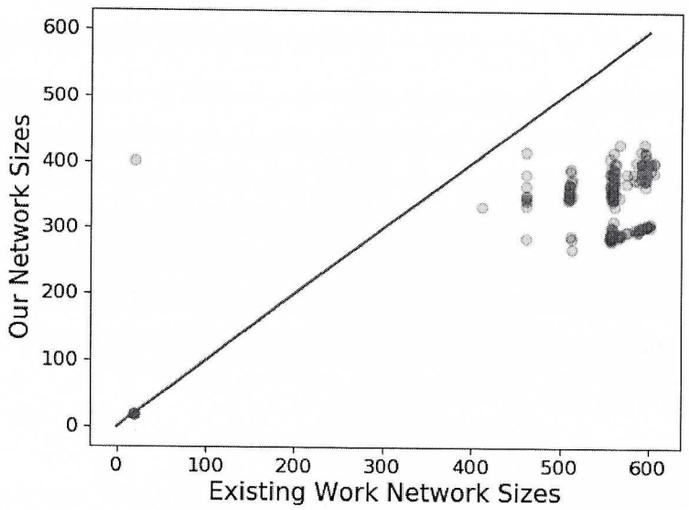


Figure 3.3: Scatter plot of network sizes produced by our framework vs existing work [EGK20] on ACASXu [OPM+19, KBD+17]

For each instance, we gathered the final $\mathcal{N}'$ at the conclusion of the CEGAR [EGK20, CGJ+03] iterations, in cases where either the property could be proven safe, the solver identified a counterexample or the solver timed out. Fig. 3.3 presents a scatter plot comparing the sizes of these final abstractions obtained through our framework and the existing work [EGK20] for each benchmark instance. A point located below the red diagonal line indicates an instance where our framework yields a smaller final abstraction than the existing approach, signifying better performance. The average sizes of these abstractions across all instances are reported in the "Average Size" columns of Tab. 3.1 and Tab. 3.2.

As shown in Fig. 3.3, Tab. 3.1, and Tab. 3.2, our approach explores smaller abstractions compared to existing techniques, effectively proving or disproving the property in question. This indicates that leveraging semantic information to guide the CEGAR process can lead to more efficient abstractions than those produced by current methods.

In our experiments, we found that both our CEGAR approach and the existing CEGAR method [EGK20] required more time than the NeuralSAT [DND24] solver for the ACASXu [OPM+19, KBD+17] benchmarks. However, we observed that solver call times scaled exponentially with the size of the network, whereas the abstraction procedure overhead did not exhibit such exponential growth. Consequently, for larger benchmarks, we believe that the

Back this claim by numbers!!

ability to generate smaller abstractions can lead to substantial reductions in execution time. Furthermore, a verified $\mathcal{N}'$ has value beyond verifying a single property—it can be utilized for other related queries or may serve as a safely deployable compressed network.

Furthermore, we found that the final solver times on $\mathcal{N}'$ are comparable to those obtained with the original unabstracted $\mathcal{N}$. Our experiments, along with findings from [EGK20], indicate that the effort required to verify a network depends on more than just its size. Notably, in [EGK20], smaller solver times were achieved on larger networks. While it is true that in general, the worst-case performance of neural network solvers will almost certainly remain exponential in the size of the network [KBD+17], exploring other factors that influence the performance of neural network solvers presents an intriguing avenue for future research.

Chapter 4

Learning DNN Abstractions using Gradient Descent

In Chapter 3, we adopted a global perspective for creating improved abstract networks by integrating the network's *semantic* information with traditional merge operations based on *syntax* from structural abstraction techniques. This information guided the CEGAR loop to undo merges of neurons that were most dissimilar in behavior and thus had the most negative impact on the abstraction. This approach allowed us to refine abstractions in a semantically optimal manner while preserving soundness guarantees.

However, while this work utilized a global perspective on the network's behavior to guide the CEGAR [EGK20, CGJ+03] process, CEGAR [EGK20, CGJ+03] search strategy itself is limited in its refinement process, as it only allows neurons to be unmerged at each step. There is no mechanism to backtrack and re-merge neurons, meaning that an early poor decision can lock the search into a sub-optimal region, potentially missing a better abstraction entirely. Consequently, the search could end up in a worst-case scenario, requiring unmerging back to the original network, wasting computational resources and missing better solutions.

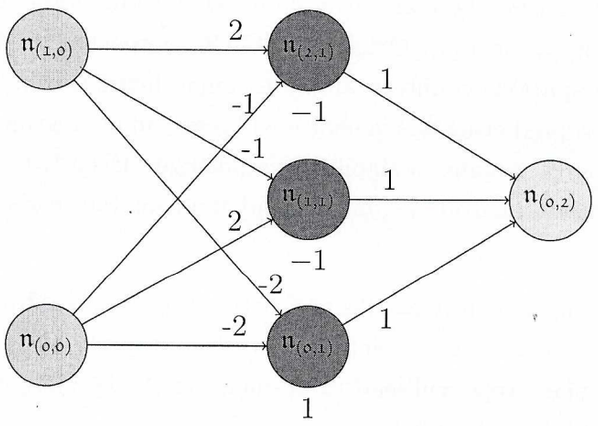


Figure 4.1: Original Network

Consider the network in Fig. 4.1, and the property P_1 which states $\forall \boldsymbol{x}, \boldsymbol{x} \in [0,1] \Rightarrow \mathcal{N}(\boldsymbol{x}) \leq 4$, where $\boldsymbol{x} = [n_{(0,0)}\ n_{(1,0)}]$. Notice that in this network, the nodes $n_{(0,1)}, n_{(1,1)}, n_{(2,1)}$ exhibit

the property that increasing the value produced at these nodes leads to an increase in the output value of the network (*inc* nodes). Consequently, following [EGK20], one can soundly substitute these nodes with another node that yields a greater output value through a *merge* operation, which takes the maximum of incoming weights and the sum of outgoing weights, as illustrated in Fig. 4.2.

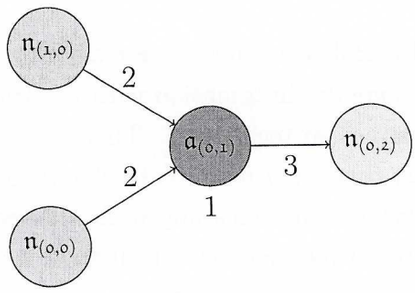


Figure 4.2: Fully Merged Network

Original CEGAR approaches [EGK20, ZZC+22, CEBK23] begins with this *fully merged* network, attempting to prove the property. However, observe that for the input $n_{(0,0)} = 0, n_{(1,0)} = 1$, the original network outputs 1, while the fully merged network outputs 6. Therefore, we have encountered a spurious counterexample and must refine the network by reversing some merges.

It is worth noting that there are several options for unmerging nodes—one could choose to unmerge $n_{(0,1)}$, $n_{(1,1)}$, or $n_{(2,1)}$. Original CEGAR approaches [EGK20, ZZC+22, CEBK23] typically uses the spurious counterexample to guide heuristics in deciding which nodes to unmerge. However, predicting which choice will result in an abstract network strong enough to verify the property remains a significant challenge. Even with a tree-guided approach, identifying the culprit neuron is difficult, and it ultimately ends up being unmerged out regardless.

If we unmerge $n_{(0,1)}$ then it results in a network with a maximum upper bound of 8 on the output $n_{(0,2)}$, which is insufficient to prove the property (Fig. 4.3). With this choice, subsequent refinement steps will lead to un-merging all the merges, ultimately reverting to the original network from Fig. 4.1.

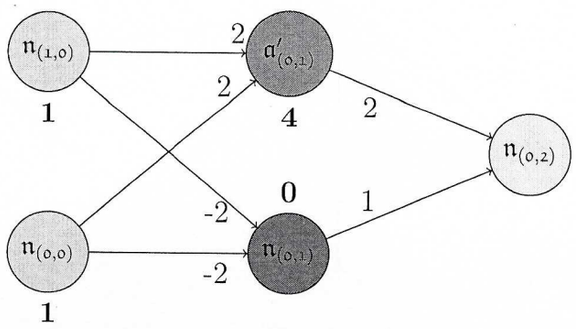


Figure 4.3: Refined network obtained by unmerging $n_{(0,1)}$

On the other hand, if we un-merge $n_{(2,1)}$, the resulting network achieves an output upper bounded by 4, thereby enabling us to prove the property P_1 (Fig. 4.4). Thus, a single incorrect decision to un-merge the neuron $n_{(0,1)}$ would have necessitated unmerging back to the original network $\mathcal{N}$.

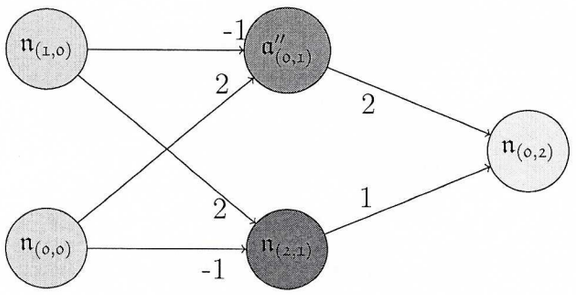


Figure 4.4: Refined network obtained by unmerging $n_{(2,1)}$

Furthermore, although we determined a locally optimal method for merging two nodes, this narrow focus may neglect the broader network dynamics, potentially resulting in a poorer overall abstraction when the impact on subsequent layers is considered.

Additionally, even though our work adopted a global perspective on the network's behavior to guide the CEGAR [EGK20, CGJ+03] process, we still relied on a fixed *syntactic* merge strategy from [CGJ+03]. However, there may exist numerous abstract networks that cannot be reached through the merge and unmerge operations employed by original CEGAR techniques [EGK20, ZZC+22, CEBK23].

Consider the same network as in Fig. 4.1 but with a different property P_2, which states that $\forall \boldsymbol{x}, \boldsymbol{x} \in [0,1] \Rightarrow \mathcal{N}(\boldsymbol{x}) \leq 2$, where $\boldsymbol{x} = [n_{(0,0)}\ n_{(1,0)}]$. In Fig. 4.5, the blue surface

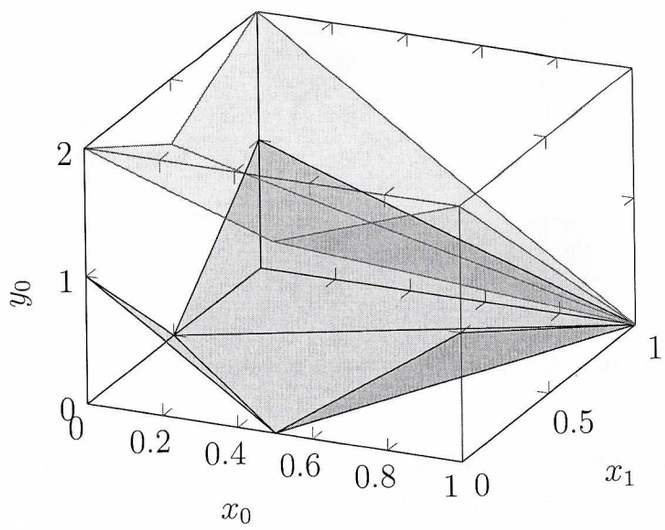


Figure 4.5: Plot of Output vs Input of $\mathcal{N}$ and $\mathcal{N}'$

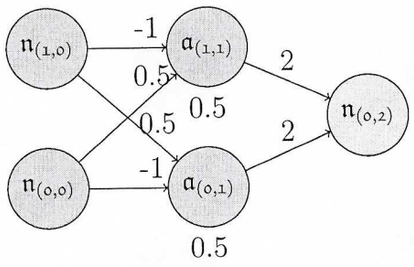


Figure 4.6: Optimal Abstract Network

represents the output of the original network in Fig. 4.1 plotted against the two inputs $n_{(0,0)}$ and $n_{(1,0)}$. Above this, the green surface is upper-bounded by 2 and is piecewise linear, consisting of three linear segments with two linear boundaries separating them.

Since such a surface can be emulated by a *ReLU* DNN with 2 neurons, the existence of this surface suggests that it may be possible to derive an abstract network with 2 neurons that is sufficiently strong enough to prove the required property. Indeed, Fig. 4.6 illustrates a 2-neuron DNN that generates the green surface shown in Fig. 4.5. However, this network cannot be reached through merge and unmerge operations starting from Fig. 4.1. Therefore, it becomes evident that existing CEGAR [EGK20, CGJ+03]-based methods may overlook more optimal abstract networks. Motivated by this, we attempt to find a process that can arrive at some of these missed abstract networks. In this running example, beginning with

the concrete network in Fig. 4.1, we can achieve the abstract network in Fig. 4.6 through the following steps.

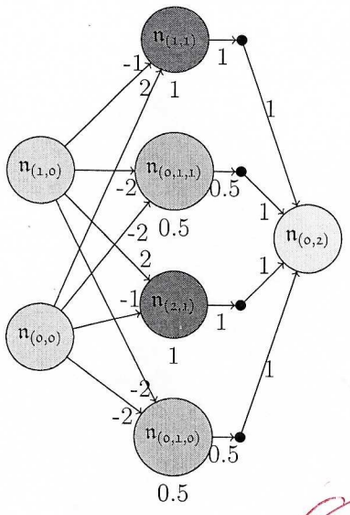


Figure 4.7: Equivalent Network after λ multiplication

First, we create two copies of $\mathfrak{n}_{(0,1)}$, labeled $\mathfrak{n}_{(0,1,0)}$ and $\mathfrak{n}_{(0,1,1)}$, and multiply the outgoing weights of each by 0.5. This modification does not alter the network's behavior because the combined contributions of these copies remain equivalent to c_0 (Fig. 4.7).

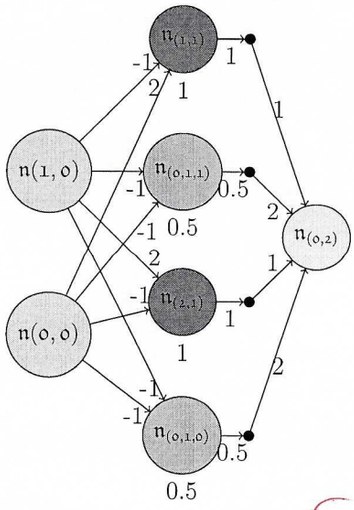


Figure 4.8: Equivalent Network after α multiplication

Next, we scale down the incoming weights and biases of $n_{(0,1,0)}$ and $n_{(0,1,1)}$ by a factor of 2 while scaling up the outgoing weights by a factor of 2. This adjustment also preserves the network's behavior since $\forall x, 2 \cdot ReLU(x) = ReLU(2x)$. The resulting network after these transformations is depicted in 4.8.

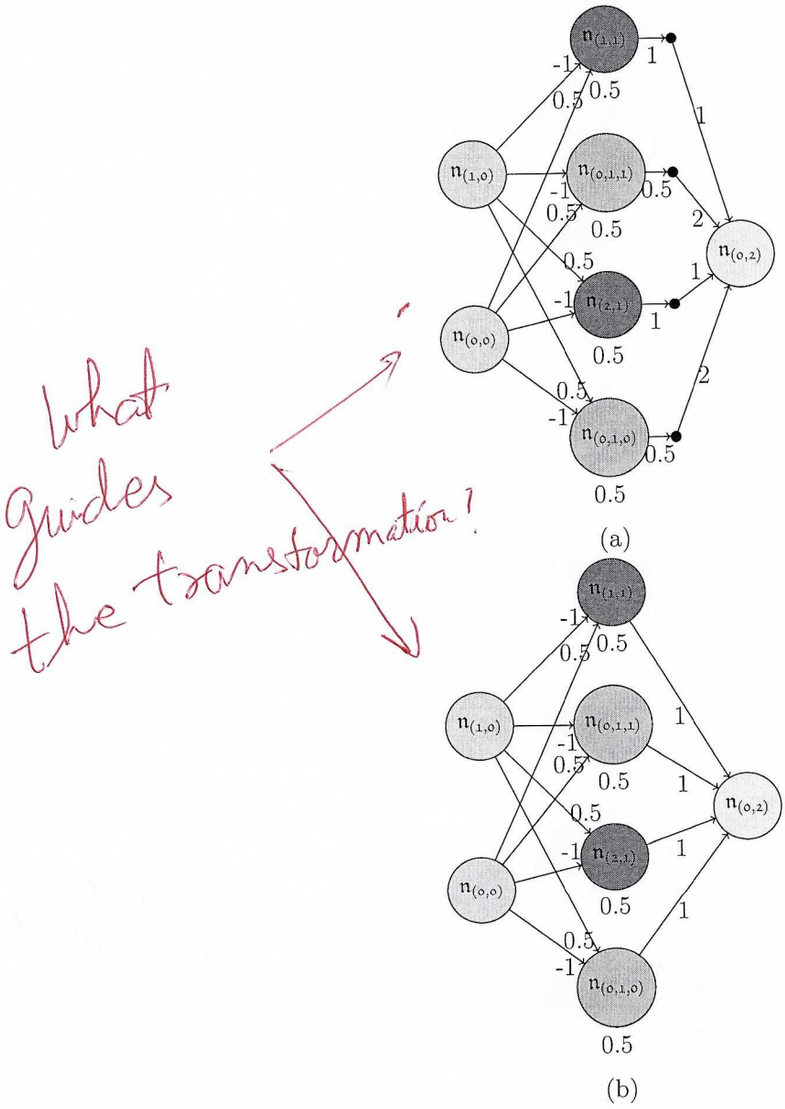


Figure 4.9: After Weight Replacement

Next, we observe that for the nodes $n_{(0,1,0)}$ and $n_{(2,1)}$, modifying the incoming weights to

$(-1, 0.5)$ and the bias to 0.5 would increase the output value of both $n_{(0,1,0)}$ and $n_{(2,1)}$ when $n_{(0,0)}$ and $n_{(1,0)}$ lie within $[0, 1]$, thus leading to an increase in the overall network output $n_{(0,2)}$. Similarly, adjusting the incoming weights and biases of $n_{(0,1,1)}$ and $n_{(1,1)}$ to $(-1, 0.5)$ and 0.5 also results in an increase in the output of the network.

Therefore, we can safely replace the incoming weights and biases with these values, obtaining the network shown in Fig. 4.9a. Since $n_{(0,1,0)}$, $n_{(2,1)}$ and $n_{(0,1,1)}$, $n_{(1,1)}$ represent pairs of neurons that perform the same function (Fig. 4.9b), we can merge them by summing their outgoing edges, resulting in the network illustrated in 4.6.

Thus, we have successfully found an abstraction that cannot be achieved through any sequence of merge and un-merge operations employed by existing techniques. Throughout this construction, we made several implicit choices, such as the values used to multiply the outputs of $n_{(0,1,0)}$ and $n_{(0,1,1)}$, the scaling factor, and more. Consequently, the process of generating the abstract network can be viewed as being parameterized by these values. Building on this idea, we propose a novel approach where:

- We introduce a *parameterized abstract network* ${\mathcal{N}'}_k(\boldsymbol{x}; \boldsymbol{\theta})$ that is continuous and differentiable almost everywhere, with parameters $\boldsymbol{\theta}$ representing the choices and values selected. This parameterization allows for encoding a broader spectrum of possible abstractions compared to those achievable through simple merge and un-merge operations.
- Next, we transform the search for an optimal $\mathcal{N}'$ into a search for the optimal $\boldsymbol{\theta}$ using *gradient descent*. This offers two key advantages:
  - We introduce a *parameterized abstract network* ${\mathcal{N}'}_k(\boldsymbol{x}; \boldsymbol{\theta})$ that is continuous and differentiable almost everywhere, with parameters $\boldsymbol{\theta}$ representing the choices and values selected. This parameterization allows encoding a broader spectrum of possible abstractions compared to those achievable through simple merge and unmerge operations.
  - Next, we transform the search for an optimal $\mathcal{N}'$ into a search for the optimal $\boldsymbol{\theta}$ using *gradient descent*. This offers two key advantages:
    * Throughout each step of the gradient descent process, the combined behavior of all layers in the network is considered, providing a more global perspective on the search for abstraction, increasing the chances of finding optimal ab.
    * The gradient descent-based approach also allows for small, incremental adjustments, enabling traversal in any direction, including the option to backtrack and re-merge neurons. This flexibility enhanced the likelihood of steering the search toward a more optimal and smaller network, even if it begins from a sub-optimal one.

* It enables us to leverage the significant engineering work invested in efficient implementations of gradient descent frameworks, such as PyTorch, as well as take advantage of specialized hardware like GPUs to accelerate the computation.

We formalize the construction of this $\mathcal{N}'_k(\boldsymbol{x};\boldsymbol{\theta})$ in Section 4.1 and, through a preliminary set of experiments (Section 4.2), demonstrate that our approach can indeed discover abstractions that are unattainable using existing methods.

4.1 Our Approach

Building on the example above, we introduce an approach centred around constructing a parameterized abstract network ($\mathcal{N}'_k(\boldsymbol{x};\boldsymbol{\theta})$), where the values of $\boldsymbol{\theta}$ capture the various choices made during the construction process. Conceptually, altering the value of $\boldsymbol{\theta}$ continuously results in a continuous change in the behavior of $\mathcal{N}'_k(\boldsymbol{x};\boldsymbol{\theta})$ as a function, much like how the behavior of DNN shifts with continuous changes to its weights and biases. Therefore, we can *train* the $\mathcal{N}'_k(\boldsymbol{x};\boldsymbol{\theta})$ using gradient descent similar to the manner in which a DNN is trained.

4.1.1 Construction of $\mathcal{N}'_k(\boldsymbol{x};\boldsymbol{\theta})$

As a pre-processing step, we apply the *inc-dec* splitting technique described in [EGK20], which transforms the original network into an equivalent one where the output behavior is monotonic with respect to every neuron in the hidden (non-input) layers. For the remainder of this discussion, we assume this transformation has already been applied, resulting in the network $\mathcal{N}$. We borrow terminology from [EGK20] to talk about *inc* and *dec* nodes in $\mathcal{N}$.

We start with a budget, k, which represents the number of abstract nodes of each type (*inc* and *dec*) allocated for each layer of $\mathcal{N}'_k(\boldsymbol{x};\boldsymbol{\theta})$. The construction of $\mathcal{N}'_k(\boldsymbol{x};\boldsymbol{\theta})$ can then be described in three stages:

Copying and Multiplying (called) multiplication in example)

In general, we do not know which abstract nodes in $\mathcal{N}'$ should be responsible for abstracting which nodes in $\mathcal{N}$. As demonstrated in the example in Fig. 4.7, multiple abstract nodes

may need to share the responsibility for abstracting a single concrete node, and vice versa. To account for all these possibilities, we create a copy $\mathfrak{n}_{(i,l,j)}$ of each concrete node $\mathfrak{n}_{(i,l)}$ for each abstract node $\mathfrak{a}_{(j,l)}$, representing a version of $\mathfrak{n}_{(i,l)}$ that will be abstracted by $\mathfrak{a}_{(j,l)}$.

We scale the output of $\mathfrak{n}_{(i,l,j)}$ by learnable parameters $\lambda_{(i,j,l)} \in [0, 1]$, which represent the degree of responsibility assumed by $\mathfrak{a}_{(j,l)}$ in abstracting $\mathfrak{n}_{(i,l)}$. By ensuring that $\sum_j \lambda_{(i,j,l)} = 1$, we guarantee that the behavior of the resulting network remains identical to $\mathcal{N}$ after this step. Thus, at this stage, we have not introduced any over-approximation.

In our example, this approach translated to creating two copies of the concrete neuron $\mathfrak{n}_{(0,1)}$, labelled $\mathfrak{n}_{(0,1,0)}$ and $\mathfrak{n}_{(0,1,1)}$, correlating to the two abstract neurons, $\mathfrak{a}_{(0,1)}$ and $\mathfrak{a}_{(1,1)}$. The outputs of these copies were scaled by learnable parameters $\lambda_{(0,0,1)} = 0.5$ and $\lambda_{(0,1,1)} = 0.5$, indicating that $\mathfrak{a}_{(0,1)}$ and $\mathfrak{a}_{(1,1)}$ shared equal responsibility for abstracting $\mathfrak{n}_{(0,1)}$. This setup ensured that both abstract neurons took on half the responsibility, maintaining the equivalence of the network without introducing over-approximation.

Our Technique Matches the Strength of Existing Methods

It is important to note that when all $\lambda_{(i,j,l)}$ values are either 0 or 1, the concrete nodes will be divided into disjoint sets, with each abstract node simply merging all nodes within the corresponding group.This is because if the values of $\lambda_{(i,j,l)}$ are either 0 or 1, then each $\mathfrak{n}_{(i,l,j)}$ contributes to exactly one $\mathfrak{a}_{(j,l)}$, being active only for that specific $\mathfrak{a}_{(j,l)}$. This is exactly what happens in original CEGAR techniques [EGK20, ZZC+22, CEBK23], where it is equivalent to $\mathfrak{n}_{(i,l)}$ contributing to $\mathfrak{a}_{(j,l)}$. The set of $\mathfrak{n}_{(i,l)}$ contributing to $\mathfrak{a}_{(j,l)}$ is called the *merge set* for the abstract neuron $\mathfrak{a}_{(j,l)}$ in original CEGAR techniques [EGK20, ZZC+22, CEBK23]. Therefore, our parameterization can express all abstractions achievable by existing techniques, and potentially more.

α Multiplication

Often, the weights of two concrete nodes that contribute similar values to later layers may differ significantly in scale. This can pose a challenge for abstraction, as any sound abstraction would need to compensate for the large disparity in weights, potentially leading to over-approximation. To address this, we introduce a learnable parameter, $\alpha > 0$, and allow the input and output weights of each copy of each concrete node to be multiplied and divided by α. This enables the re-balancing of the weight scales between concrete nodes.

Since $ReLU(\alpha x) = \alpha \cdot ReLU(x)$ for $\alpha > 0$, the network's behavior remains identical to $\mathcal{N}$ at this stage, ensuring that no over-approximation has been introduced.

In our example, we reduced the incoming weights and biases of $\mathfrak{n}_{(0,1,0)}$ and $\mathfrak{n}_{(0,1,1)}$ by a factor of 2, while simultaneously scaling up their outgoing weights by a factor of 2. This adjustment was made to avoid introducing unnecessary over-approximation and to help rebalance the scales effectively. This step corresponds to the α Multiplication phase in our example.

Sound Replacement by Abstract Nodes

Finally, we replace the $\mathfrak{n}_{(i,l,j)}$ with weights $\boldsymbol{w_i}$ by the abstract node $\mathfrak{a}_{(i,j,l)}$ with weights $\boldsymbol{w'_j}$ to obtain the abstract network. Each of these $\boldsymbol{w'_j}$ are learnable parameters, and their values are determined through gradient descent. For this replacement to be sound, the following condition must hold for each *inc* node:

$$\mathbf{S} : \forall \boldsymbol{x} \in I, ReLU(\boldsymbol{x}.\boldsymbol{w'_j}) \geq ReLU(\boldsymbol{x}.\boldsymbol{w_i})$$

and similarly for *dec* nodes. Here, I represents an interval covering all reachable values of $\boldsymbol{x}$, which can be computed via interval propagation. The condition $\mathbf{S}$ builds on the sufficient condition derived in [EGK20], adding the constraint that the output value at each node only needs to increase for reachable input values, rather than all possible input values. This relaxed condition allows for a broader set of potential choices for $\boldsymbol{w'_j}$, which were not considered by original CEGAR techniques [EGK20, ZZC+22, CEBK23], enabling the discovery of abstractions not found by existing methods.

However, $\mathbf{S}$ is difficult to enforce directly within a gradient descent framework, as gradient descent is not well-suited for constrained optimization. Therefore, we relax the condition to:

$$\mathbf{R} : \forall \boldsymbol{x} \in I, ReLU(\boldsymbol{x}.\boldsymbol{w'_j}) + \delta_{(i,j,l)} \geq ReLU(\boldsymbol{x}.\boldsymbol{w_i}),$$

where $\delta_{(i,j,l)}$ represents a *residual value* that can be computed by maximizing

$$ReLU(\boldsymbol{x}.\boldsymbol{w_i}) - ReLU(\boldsymbol{x}.\boldsymbol{w'_j})$$

within the interval I. Since this function is piecewise linear with four distinct linear regions, a case analysis provides a differentiable closed-form solution to the maximization problem. By leveraging the monotonicity properties of *inc-dec* split networks [EGK20], we propagate the contribution of these residuals, $\delta_{(i,j,l)}$, to the output layer in a manner similar to interval propagation, adding the resulting values to the output. This results in a ${\mathcal{N}'}_k(\boldsymbol{x};\boldsymbol{\theta})$ that is a sound abstraction [1].

In the running example from Fig. 4.7, we chose $\boldsymbol{w'_j}$ such that **S** holds, meaning the residuals were zero. While in practice, it is unlikely to find $\boldsymbol{w'_j}$ such that the residuals are exactly zero, even with non-zero residuals, it would still be possible to find a ${\mathcal{N}'}_k(\boldsymbol{x};\boldsymbol{\theta})$ that produces the green plot in 4.5.

Write a theorem that says that transformation is correct?

4.1.2 Learning Abstractions via Gradient Descent

The $\boldsymbol{\theta}$ in ${\mathcal{N}'}_k(\boldsymbol{x};\boldsymbol{\theta})$ will include all the learnable parameters described in the previous section, namely $\lambda_{(i,j,l)}$, $\alpha_{(i,j,l)}$, and $\boldsymbol{w'_j}$. These parameters can then be optimized using gradient descent to find a suitable $\mathcal{N}'$. Note that the values of these parameters at each layer can have a complex and non-linear impact on the network's output, due to the effects propagated through subsequent layers. It would be very hard to design CEGAR heuristics that take this global non-linearity into account. In contrast, gradient descent is well-equipped to handle such non-linearities. In fact, training *ReLU* DNNs itself is a problem where such non-linear dependencies between changes across layers are seen regularly. Therefore, we expect gradient descent to be effective in optimizing the search for $\boldsymbol{\theta}$.

Counterexample Loss

We wish to find an $\mathcal{N}'$ that is strong enough to prove the property. To achieve this, we construct a loss function that quantifies the strength of the abstraction ${\mathcal{N}'}_k(\boldsymbol{x};\boldsymbol{\theta})$. Intuitively, this is done by estimating the extent to which a random input will violate the property. Suppose our properties are of the form $\forall \boldsymbol{x}, \boldsymbol{x} \in I_i \Rightarrow \mathcal{N}(\boldsymbol{x}) \leq u_o$, where I_i represents the interval bounds for the input, and u_o is an upper bound for the output. It is worth noting that, as detailed in [EGK20], many general properties can be reduced to this form. The loss

[1] Soundness is ensured by the monotonicity properties, see [EGK20].

function is then defined as How does this loss function reduce size?

$$\mathop{\mathbb{E}}_{\boldsymbol{x}\in I_i} max(\mathcal{N'}_k(\boldsymbol{x};\boldsymbol{\theta}) - u_o, 0)$$

, which can be estimated by uniformly sampling a large number of inputs, $\boldsymbol{x}$, from I_i.

Once training is complete, we extract a $\mathcal{N}'$ from the $\mathcal{N'}_k(\boldsymbol{x};\boldsymbol{\theta})$ produced by simply taking a network with nodes $\mathfrak{a}_{(\mathrm{i,j,l})}$ and incoming weights $\boldsymbol{w'_j}$, and scale the output weights by $\lambda_{(i,j,l)}$. Note that this network is equivalent to $\mathcal{N'}_k(\boldsymbol{x};\boldsymbol{\theta})$, and thus, it is just as unlikely as $\mathcal{N'}_k(\boldsymbol{x};\boldsymbol{\theta})$ to produce a spurious counterexample. This $\mathcal{N}'$ only has k nodes in each layer, and thus the effort needed to check $\mathcal{N}' \vdash P$ will be less than $\mathcal{N} \vdash P$.

4.2 Experimental Evaluation

4.2.1 Set-up

To demonstrate that this $\mathcal{N'}_k(\boldsymbol{x};\boldsymbol{\theta})$ indeed represents tighter abstraction than those that can be found by CEGAR and that this can be obtained via gradient descent in a practically feasible manner, we have implemented a basic prototype in Python. We used PyTorch as the gradient descent framework. We ran preliminary experiments on a system with a 11th Gen Intel(R) Core(TM) i5-1145G7 @ 2.60GHz CPU and 16 GB of RAM.

4.2.2 Benchmarks

We chose a small (3x50) VNNComp MNIST [BMB+23, BBLJ23, Den12] classifier and three ϵ-robustness properties from the ERAN [MMS+22, MSS+21, RBBV21, BBS+19, SYZ+19, SGPV19a, SGM+18a] benchmarks. Then, we set a budget of 30 abstract neurons per layer (representing a 40% reduction in size) and ran our gradient descent framework for 2000 steps to obtain abstract networks. As a baseline to compare against, we also ran the existing CEGAR-based method from [EGK20], stopping the CEGAR loop once the budget of 30 abstract neurons per layer is hit.

4.2.3 Results

Table 4.1: Comparison of Counterexample Loss

Property	Existing Cex Loss	Our Cex Loss
Instance 1, $\epsilon = 0.008$	514.5	174.7
Instance 0, $\epsilon = 0.004$	217.8	110.9
Instance 0, $\epsilon = 0.031$	238.0	201.9

In Tab. 4.1, we compare the counterexample loss (from Section 4.1.2) achieved by our networks and those produced using CEGAR. We find that our approach is consistently able to achieve a lower loss than CEGAR, indicating that we are able to find abstractions that existing techniques are not able to find.

- No timing
- Number of abstract neurons