Syntactic vs Semantic Linear Abstraction and Refinement of Neural Networks

Calvin Chau[0000−0002−3437−0240], Jan Křetínský[0000−0002−8122−2881], and Stefanie Mohr[0000−0002−8630−3218]

Abstract. Abstraction is a key verification technique to improve scalability. However, its use for neural networks is so far extremely limited. Previous approaches for abstracting classification networks replace several neurons with one of them that is similar enough. We can classify the similarity as defined either syntactically (using quantities on the connections between neurons) or semantically (on the activation values of neurons for various inputs). Unfortunately, the previous approaches only achieve moderate reductions, when implemented at all. In this work, we provide a more flexible framework, where a neuron can be replaced with a *linear combination* of other neurons, improving the reduction. We apply this approach both on syntactic and semantic abstractions, and implement and evaluate them experimentally. Further, we introduce a refinement method for our abstractions, allowing for finding a better balance between reduction and precision.

1 Introduction

Neural Network Abstractions Abstraction is a key instrument for understanding complex systems and analyzing complex problems across all disciplines, including computer science. Abstraction of complex systems, such as neural networks (NN), results in smaller systems, which are not only producing equivalent outputs (such as in distillation [10]), but additionally can be mapped to the original system, providing a strong link between the individual parts of the two systems. Consequently, abstraction find various applications. For instance, the smaller (abstract) networks are more understandable and the strong link between the behaviours of the abstract and the original network allows for better explainability of the original behaviour, too; smaller networks are more efficient in resource usage during runtime; smaller networks are easier to verify. Again, with no formal link between the original network and, say, a distilled or pruned one, verifying the smaller one is of no use to verifying the original one. In contrast, for abstractions, the verification guarantee can be in principle transfered to the original network, be it via lifting a counterexample or a proof of correctness.

Altogether, abstractions of neural networks are a key concept worth investigating *eo ipso*, subsequently offering various applications. However, currently it is still very under-developed. For defining an abstraction, we need a transformation linking the original neurons to those in the abstraction. Equivalently, we need a notion of the *similarity of neurons*, to identify a good representative of a group of neurons. The difficulty in contrast to, e.g., predicate abstraction of programs is that neurons have no inner structure such as values of variables stored in a

state. On the one hand, approaches based on bisimilarity [19] offer a solution focusing on the *"syntax"* of neurons: the weights of the incoming connections. The quantities give rise to an equivalence akin to probabilistic bisimulation. On the other hand, in search of a stronger tool, approaches such as [2] try to identify *"semantics"* of the neurons. For instance, given a vector of inputs to the network, the *I/O semantics* of a neuron [2] is the vector of activation values of this neuron obtained on these inputs. This represents a finite-dimensional approximation of the actual semantics of a neuron as a computational device. Either way, replacing several neurons with one that is very similar yields only moderate savings on size if the abstract network is supposed to be similar, i.e., yield mostly the same predictions and ensure a tight connection between the similar neurons.

Our Contribution We focus on studying abstraction irrespective of the use case (verification, smaller networks, explainability), to establish a better principal understanding of this crucial, yet in this context underdeveloped technique. First, we explore a richer abstraction scheme, where a group of neurons can be represented not only by a chosen neuron but also by a *linear combination of neurons*. Thus instead of keeping exactly one representative per group, we can "reuse" the chosen representatives in many linear combinations; in other words, the representatives can attain many roles, partially representing many groups, which reduces their required count. We provide several algorithms to do so, ranging from resource-intensive algorithms aiming to show the limits of the approach to efficient heuristics approximating the former ones quite closely. We apply these algorithms to the semantic approach of [2] as well as to the syntactic, bisimulation-like approach similar to [19] not implemented previously. Experimental results confirm the *greater power of this linear-combination* approach; further, they provide insight into the *advantages of semantic similarity over the syntactic* one, pointing out the more advantageous future research directions.

Further, we provide a formal link between the concrete and abstract neurons by proving an error bound induced by the abstraction, showing the abstraction is valid and (approximately) simulates the original network. We show the bound is better than the one based on bisimulation. While still not very practical, the experiments show that even on unseen data, the error is always closely bounded by the error on the data used for generating the abstraction, and mostly even a lot smaller. This empirical version of the concept of error could thus enable the transfer of reasoning about the abstraction to the original network in a yet much tighter way.

In addition, we suggest *abstraction-refinement* procedures to better fine-tune the trade-off between the precision and the size of the abstraction. The experiments reveal that a more aggressive abstraction followed by a refinement provides better results than a direct, moderate abstraction. Hence involving our refinement in the abstraction process improves the resulting quality, opening new lines of attack on efficient neural network abstractions.

Summary Our contribution can be summarized as follows:

- We define abstractions of neural networks with (approximate) equivalences being linear equations over semantics of neurons. We provide a theoretical

bound on the induced error, see Thm. 1. We reflect this idea also on the syntactic, bisimulation-based abstraction.
- We implement both approaches and compare them mutually as well as to their previous, special cases with equivalences being (approximate) identities. We perform the experiments on a number of standard benchmarks, such as MNIST, CIFAR, or FashionMNIST, concluding advantages of semantic over syntactic approaches and of linear over identity-based ones.
- We introduce an abstraction-refinement procedure and also evaluate its benefits experimentally.

Related Work There are various approaches for verification of NN, however, we are **not** presenting another verifier. Instead, we introduce an approach that is **orthogonal to verification** and could be integrated with an existing verifier. Therefore, we do not compare our approach to any verification tool and refer the interested reader to the Verification of Neural Networks Competition [4] for an overview of existing approaches [12, 22, 30, 28].

Network compression techniques share many similarities with abstraction [5] and either focus on reducing the memory footprint [31, 11] or computation time of the model [8], but in contrast, do not provide any formal relation to the original network, rendering them inappropriate for understanding redundancies or verification. Knowledge distillation is a prominent technique, which can reduce networks by a significant amount, but completely loses any connection to the original network [10], and can thus not be used in verification. There is some progress in using abstract domains for scalable verification, like [22, 23, 25], but they do not produce an abstracted NN for verification. Instead, they apply abstraction only tightly entangled together with the verification algorithm. These approaches also try to generate a more scalable verification, however, the key difference is that they do not return an actual abstracted network that could be reused or manually inspected. Katz et al. [6] introduce an abstraction scheme for NN, in which they decompose neurons into several parts, before merging them again to obtain an over-approximation of the original network. However, their approach is limited to networks with one output neuron. For networks with more output neurons, the property to be verified needs to be baked into the network, making the approach significantly less flexible. Additionally, this tight entanglement of specification and neural network does not allow for retrieving the abstraction later and reusing it for anything else than to verify that specific property. This strongly contrasts our generic and usage-agnostic abstraction and their property-restricted abstractions.

Some other works use abstraction after representing a neural network as an interval neural network [20], or more generally, by using more complex abstract domains [24]. While theoretically interesting, the practicality of these works has not been investigated. There are two approaches that we consider to be the closest to our work: a bisimulation-based approach [19], and *DeepAbstract* [2], which we will more closely introduce in the preliminaries, and compare to in the experiments.

2 Preliminaries

In this work, we focus on classification feedforward neural networks. Such a neural network N consists of several layers $1, 2, \ldots, L$, with 1 being the *input layer*, L being the *output layer* and $2, \ldots, L-1$ being the *hidden layers*. Each layer ℓ contains n_ℓ *neurons*. Neurons of one layer are connected to neurons of the previous and next layers by means of weighted connections. Associated with every layer ℓ that is not an output layer is a *weight matrix* $W^{(\ell)} = (w^{(\ell)}(i,j)) \in \mathbb{R}^{n_{\ell+1} \times n_\ell}$ where $w^{(\ell)}(i,j)$ gives the weights of the connections to the i^{th} neuron in layer $\ell+1$ from the j^{th} neuron in layer ℓ. We use the notation $W^{(\ell)}_{i,*} = [w^{(\ell)}(i,1), \ldots, w^{(\ell)}(i,n_\ell)]$ to denote the incoming weights of neuron i in layer $\ell+1$ and

$W^{(\ell)}_{*,j} = [w^{(\ell)}(1,j), \ldots, w^{(\ell)}(n_{\ell+1},j)]^\intercal$ to denote the outgoing weights of neuron j in layer ℓ. Note that $W^{(\ell)}_{i,*}$ and $W^{(\ell)}_{*,j}$ correspond to the i^{th} row and j^{th} column of $W^{(\ell)}$ respectively. A vector $\mathbf{b}^{(\ell)} = [b^{(\ell)}_1, \ldots, b^{(\ell)}_{n_\ell}] \in \mathbb{R}^{n_\ell}$ called *bias* is also associated with each hidden layer ℓ. The input and output of a neuron i in layer ℓ is denoted by $h^{(\ell)}_i$ and $z^{(\ell)}_i$ respectively. We call $\mathbf{h}^\ell = [h^{(\ell)}_1, \ldots, h^{(\ell)}_{n_\ell}]^\intercal$ the vector of *pre-activations* and $\mathbf{z}^\ell = [z^{(\ell)}_1, \ldots, z^{(\ell)}_{n_\ell}]^\intercal$ the vector of *activations* of layer ℓ. The neuron takes the input $\mathbf{h}^\ell$, and applies an *activation function* $\phi : \mathbb{R} \to \mathbb{R}$ element-wise on it. The output is then calculated as $\mathbf{z}^\ell = \phi(\mathbf{h}^\ell)$, where standard activation functions include tanh, sigmoid, or ReLU [17]. We assume that the activation function is Lipschitz continuous, which in particular holds for the aforementioned functions [26]. In a feedforward neural network, information flows strictly in one direction: from layer ℓ_m to layer ℓ_n where $\ell_m < \ell_n$. For an n_1-dimensional input $\mathbf{x} \in \mathcal{X}$ from some input space $\mathcal{X} \subseteq \mathbb{R}^{n_1}$, the output $\mathbf{y} \in \mathbb{R}^{n_L}$ of the neural network N, also written as $\mathbf{y} = N(\mathbf{x})$ is iteratively computed as:

$$\begin{aligned} \mathbf{h}^{(0)} &= \mathbf{z}^{(0)} = \mathbf{x} \\ \mathbf{h}^{(\ell+1)} &= W^{(\ell)}\mathbf{z}^{(\ell)} + \mathbf{b}^{(\ell+1)} && (1) \\ \mathbf{z}^{(\ell+1)} &= \phi(\mathbf{h}^{(\ell+1)}) && (2) \\ \mathbf{y} &= \mathbf{z}^{(L)} \end{aligned}$$

where $\phi(\mathbf{x})$ is the column vector obtained by applying ϕ component-wise to $\mathbf{x}$. We abuse the notation and write $\mathbf{z}^{(\ell)}(\mathbf{x})$, when we want to specify that the output of layer ℓ is computed by starting with $\mathbf{x}$ as input to the network.

2.1 Syntactic and Semantic Abstractions

We are interested in a general abstraction scheme that is not only useful for verification, but also for revealing redundancies, while keeping a formal link to the original network. We distinguish between two types of abstraction: semantic and syntactic. Syntactic abstraction makes use of the weights of the network, the syntactic information, and allows for overapproximation guarantees that are not restricted to specific inputs. However, as we shall see in the experiments, the semantic abstraction can capture the behavior of the original network on typical input data much more accurately than its syntactic counterpart. This comes at the cost of a more challenging error analysis.

Semantic Information In line with *DeepAbstract* [2], we will create the semantic information based on a set of inputs, the *I/O set*, $X = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\} \subseteq \mathcal{X}$, which is typically a subset of the training dataset. We use the inputs $\mathbf{x}_j \in X$, feed them to the network and store the output values $\{\mathbf{z}^{(\ell)}(\mathbf{x}_j)\}_{\mathbf{x}_j \in X}$ of a layer ℓ in a matrix $\mathbf{Z}^{(\ell)} = (z_i^{(\ell)}(\mathbf{x}_j))_{i,j}$. Note that the columns are the $\mathbf{z}^{(\ell)}(\mathbf{x}_j)$ and the rows, denoted as $\mathbf{Z}^{(\ell)}_{i,*}$, correspond to the values one neuron i produces for all inputs $\mathbf{x}_j$. We refer to the vector $\mathbf{Z}^{(\ell)}_{j,*}$ as the *semantics* of neuron i. This collection of matrices $\mathbf{Z}^{(\ell)}$ for all layers contains the semantic information of the network.

DeepAbstract Since we will compare our approach to *DeepAbstract* [2], we will give a concise description of the idea of their work. First, it generates the semantic information $\mathbf{Z}$. For one layer ℓ, it clusters the rows of the matrix by using standard clustering techniques, e.g. k-means clustering [3]. Each cluster is considered to be a group of neurons that have similar semantics and similar behavior. Thus, only one group representative is chosen to remain and the rest is replaced by the representatives.

Bisimulation The idea of [19] is to apply the notion of bisimulation to NN. A bisimulation declares two neurons as equivalent if they agree on their incoming weights, biases, and activation functions. Additionally, the paper introduces a δ-bisimulation that allows neurons to be equivalent only up to δ, i.e. two neurons i, j of layer ℓ with the same activation function are considered to be δ-bisimilar, if for all $k : |w^{(\ell-1)}(i,k) - w^{(\ell-1)}(j,k)| \leq \delta$ and $|b_i^{(\ell)} - b_j^{(\ell)}| \leq \delta$.

3 Linear Abstraction

Our abstraction of a NN is based on the idea that huge NN in their practical application are usually trained with more neurons than necessary. Since there are techniques to avoid "overfitting", users of machine learning tend to use NN that are bigger than necessary for their task [15]. Intuitively, such networks thus contain redundancies. We want to remove these redundancies to decrease the size of the network and make it more scalable for verification.

Existing approaches group together similar neurons, and then choose a representative. Instead, we propose to replace a neuron with a linear combination of other neurons. More specifically, we want to replace a neuron i of layer ℓ, not by one single neuron j, but rather by a clever combination of several neurons, called the *basis*, $B^{(\ell)} \subset \{1, \ldots, n_\ell\} \backslash \{i\}$, which is a subset of all neurons of this layer and in this case given as their indices. We assume that the behavior of a neuron can be simulated by a linear combination of the behavior of the basis neurons, i.e. by $\sum_{j \in B^{(\ell)}} \alpha_{i,j}^{(\ell)} \cdot \mathbf{Z}^{(\ell)}_{j,*}$ for some $\alpha_{i,j}^{(\ell)} \in \mathbb{R}$.

Example Consider the neural network in Fig. 1. It has an input layer with two neurons n_1^0, n_2^0, one hidden layer with three neurons n_1^1, n_2^1, n_3^1, and an output layer with two neurons n_1^2, n_2^2. We assume that we are given the basis $B^{(1)} = \{n_2^1, n_3^1\}$, marked with blue color in the figure, and the linear coefficients $\alpha_{1,1}^{(1)}, \alpha_{1,2}^{(1)}$. That is, we assume that n_1^1 can be simulated by the linear combination $\alpha_{1,1}^{(1)} \cdot n_2^1 + \alpha_{1,2}^{(1)} \cdot n_3^1$. We can remove neuron n_1^1 and its outgoing

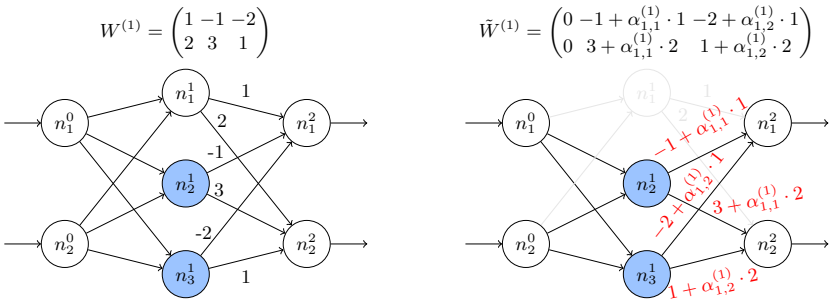


Fig. 1: Linear Abstraction - On the left, the original network with the basis B in blue. On the right, the abstracted network with the removed neuron n^1_1 and the changed output weights of the basis neurons n^1_2, n^1_3, where we assume that n^1_1 can be simulated by $\alpha^{(1)}_{1,1} \cdot n^1_2 + \alpha^{(1)}_{1,2} \cdot n^1_3$.

weights $[1,2]^\intercal$, and add the outgoing weights scaled by the linear coefficients to the basis neurons instead. We add $\alpha^{(1)}_{1,1} \cdot [1,2]^\intercal$ to the outgoing weights of neuron n^1_2, so we get $[-1,3]^\intercal + \alpha^{(1)}_{1,1} \cdot [1,2]^\intercal = [-1+\alpha^{(1)}_{1,1}\cdot 1, 3+\alpha^{(1)}_{1,1}\cdot 2]^\intercal$, and respectively, we get $[-2+\alpha^{(1)}_{1,2}\cdot 1, 1+\alpha^{(1)}_{1,2}\cdot 2]^\intercal$ as the outgoing weights of neuron n^1_3.

The computational overhead to compute a linear combination compared to finding a representative is negligible, as we will see in our experiments (see Section 5.2). On the other hand, they provide more expressive power, subsuming the aforementioned clustering-based approach [2]. In particular, we can detect scaled weights that previous approaches failed to identify.

Please note that although it is possible to replace a neuron with a linear combination of any other neurons in the network, we will only use neurons from the same layer due to more efficient support by modern neural network frameworks.

In the following sections, we will answer three questions: How can one find a set of neurons that serves as a basis (3.1)? How to find the coefficients for the linear combination (3.2)? How to replace a neuron, once its representation as a linear combination is given (3.3)?

3.1 Finding the Basis

Our approach is meant to find a sufficient smaller subset of neurons in one layer, which is enough to represent the behavior of the whole layer. We will make use of the semantic information of a layer ℓ, given as $\mathbf{Z}^{(\ell)} = (z^{(\ell)}_i(\mathbf{x}_j))_{i,j}$ (see 2.1). Based on this, we try to find a set of indices for neurons in this layer $B^{(\ell)} \subset \{i, \dots, n_\ell\}$, which can represent the whole space as well as possible. To this end we want to find a subset of size $k = |B^{(\ell)}|$ such that $\| \sum_{j \in B^{(\ell)}} \alpha^{(\ell)}_{i,j} \cdot \mathbf{Z}^{(\ell)}_{j,*} - \mathbf{Z}_{i,*} \|$ is minimized.

Algorithm 1 Greedy algorithm over all layers

Given: k neurons to be removed
$B_j \leftarrow \{1, \cdot, n_j\}$
$error_{min} \leftarrow \infty$, $l_{best} \leftarrow -1$, $n_{best} \leftarrow -1$
for $i \in 1, \ldots, k$ **do**
    for $l \in 1, \ldots, L$ **do**
        Choose j such that $A_{B \setminus \{j\}}$ minimizes the projection error $error_j$
        if $error_j < error_{min}$ **then**
            $l_{best} \leftarrow l$
            $n_{best} \leftarrow j$
            $error_{min} \leftarrow error_j$
    $B_{l_{best}} \leftarrow B_{l_{best}} \setminus \{n_{best}\}$
return B_j

Greedy Algorithm The problem of finding an optimal basis of size k w.r.t. L_2 distance can be seen as a variation of the *column subset selection problem* which is known to be NP-complete [21]. As a consequence, we use a variant of the greedy algorithm as presented in [1]. While it does not always yield the optimal solution, it has been observed to work reasonably well in practice.

It has already been observed that layers closer to the output usually contain more condensed information and more redundancies, and can, thus, be compressed more aggressively [2]. We present a greedy algorithm that chooses which layer contains more information and needs a larger basis instead of decreasing the basis sizes equally fast in each layer.

In Algorithm 1, we can see that the procedure evaluates all neurons in all layers $l \in \{1, \ldots, L\}$ in the network. It calculates the projection error that is defined as $\|A - \Pi_{A_B} A\|$, where A is the matrix in Eq. (3) with all neurons instead of just the ones in the basis, and $\Pi_{A_B} A$ is the projection matrix on the columns of A_B applied to A. It greedily evaluates all neurons in all layers and selects the best neuron of the best layer to be removed. After checking every layer, the algorithm decides on the best layer and neuron to be removed , i.e. the one with the smallest error.

Since the approach thoroughly evaluates all possibilities, its runtime depends on both the number of layers and neurons. A natural alternative would be a heuristic that guides us similarly well through the search space. We provide our choice of heuristic below.

Variance-based Heuristic Instead of a step-wise decision that takes a lot of computation time, we propose to use a variance-based heuristic. We define the variance of a vector $\mathbf{v} \in \mathbb{R}^n$ in the usual way by $\mathrm{Var}(\mathbf{v}) = \sum_{i=0}^{n} (v_i - \mathrm{Mean}(\mathbf{v}))^2$ where $\mathrm{Mean}(\mathbf{v})$ is the mean of the vector values. W.l.o.g. let the neurons be numbered in such a way that $\mathrm{Var}(\mathbf{z}_1^{(\ell)}) \geq \ldots \geq \mathrm{Var}(\mathbf{z}_{n_\ell}^{(\ell)})$. We then choose the basis to contain the neurons with the k_ℓ largest variances, i.e. $B = \{1, \ldots, k\}$. We assume that neurons with a higher variance in their output values carry more information, and are, therefore, more relevant. Indeed, we can see in our

experiments, i.e. Fig. 2, that the heuristic-based approach can achieve similar results, but in far less time.

3.2 Finding the Coefficients

Suppose we have determined a basis $B^{(\ell)}$ for some layer ℓ with the beforementioned approaches. The task is to find the coefficients that can be used to replace the remaining neurons which are not part of the basis. We fix a neuron i in layer ℓ that we want to replace and whose values are stored in $\mathbf{Z}^{(\ell)}_{i,*}$, and we want to minimize $\| \sum_{j \in B^{(\ell)}} \alpha^{(\ell)}_{i,j} \cdot \mathbf{Z}^{(\ell)}_{j,*} - \mathbf{Z}_{i,*} \|$ for the $\alpha^{(\ell)}_{i,j}$.

Since we want to find a linear combination of vectors, a natural choice is **linear programming**. The linear program is straightforward and can be found in Appendix C. Note that with the linear program, we are minimizing the L_1-distance between the neuron's values and its replacement, i.e. $\| \sum_{j \in B^{(\ell)}} \alpha^{(\ell)}_{i,j} \cdot \mathbf{Z}^{(\ell)}_{j,*} - \mathbf{Z}_{i,*} \|_1$.

Note that we can also consider the vectors $\mathbf{Z}^{(\ell)}_{j,*}$ for $j \in B^{(\ell)}$ to span a vector space. If we are given a subset $\{\mathbf{Z}^{(\ell)}_{j,*} | j \in B^{(\ell)} \subset \{1, \dots, n_\ell\}\}$ that forms a basis for this space, i.e. $\text{span}((\mathbf{Z}^{(\ell)}_{j,*})_{j \in B^{(\ell)}}) = \text{span}((\mathbf{Z}^{(\ell)}_{j,*})_{j \in \{1,\dots,n_\ell\}})$, we can represent any other vector $\mathbf{z}^{(\ell)}_i$ in terms of this basis. However, we usually cannot represent one neuron perfectly by a linear combination of other neurons. **Orthogonal projection** gives us the closest point in the subspace $\text{span}((\mathbf{Z}^{(\ell)}_{j,*})_{j \in B^{(\ell)}})$ for any vector, in terms of L_2-distance. Let the basis be given by $B = \{j_1, \dots j_{k_\ell}\}$, then

$$A_B = \begin{bmatrix} | & & | \\ \mathbf{Z}^{(\ell)}_{j_1,*} & \dots & \mathbf{Z}^{(\ell)}_{j_{k_\ell},*} \\ | & & | \end{bmatrix} \tag{3}$$

is the matrix containing the activations $\mathbf{Z}^{(\ell)}_{j,*}$ of the neurons in the basis as columns. Then, $\boldsymbol{\alpha} = [\alpha^{(\ell)}_{i,j_1}, \dots, \alpha^{(\ell)}_{i,j_{k_\ell}}]^\intercal \coloneqq (A_B^\top A_B)^{-1} A_B^\top \mathbf{Z}^{(\ell)}_{i,*}$ gives us the coefficients for the orthogonal projection of $\mathbf{Z}^{(\ell)}_{i,*}$ on the linear space spanned by the columns of A_B. For a more detailed description of orthogonal projection see e.g. [13, Chapter 6.8]. Note that we assume that the columns of A_B are linearly independent. If not we can simply replace the respective neurons directly.

3.3 Replacement

Assuming, we have a basis $B^{(\ell)}$ of this layer and we already know the coefficients $\alpha^{(\ell)}_{i,j} \in \mathbb{R}$ for $j \in B^{(\ell)}$ that we need to simulate the behavior of neuron i. This means, we have a linear combination $\sum_{j \in B^{(\ell)}} \alpha^{(\ell)}_{i,j} \cdot \mathbf{Z}^{(\ell)}_{j,*}$, which we want to use instead of neuron i itself. We will replace the outgoing weights $W^{(\ell)}$ of this layer, such that for all $j \in B^{(\ell)}$

$$\tilde{W}^{(\ell)}_{*,j} = [w^{(\ell)}(1,j) + \alpha^{(\ell)}_{i,j} w^{(\ell)}(1,i), \dots, w^{(\ell)}(n_{\ell+1},j) + \alpha^{(\ell)}_{i,j} w^{(\ell)}(n_{\ell+1},i)]^\intercal \tag{4}$$

$$= W^{(\ell)}_{*,j} + \alpha^{(\ell)}_{i,j} W^{(\ell)}_{*,i} \tag{5}$$

Furthermore, we set $\tilde{W}^{(\ell)}_{*,i} = [0, \dots, 0]^\intercal$, and $\tilde{W}^{(\ell)}_{i,*} = [0, \dots, 0]^\intercal$. This means that we will not use the output of neuron i anymore, but rather a weighted sum of the outputs of neurons in $B^{(\ell)}$, and that we will not even compute the value of i.

Additionally, we keep track of the changes we apply to the different neurons with a matrix $D^{(\ell)} = (d^{(\ell)}_{j,i}) \in \mathbb{R}^{n_\ell \times n_{\ell+1}}$. Initially, $D^{(\ell)}$ is 0 and after each replacement, we add $\alpha^{(\ell)}_{i,j} \cdot w^{(\ell)}(i, i')$ to $d^{(\ell)}_{j,i'}$ for $j \in B^{(\ell)}$ and $i' \in \{1, ..., n_{\ell+1}\}$. This is necessary for restoring neurons at a later point.

In the optimal case, the replacement will not change the overall behavior of the neural network. We can derive a the same semantic equivalence from [19] incorporated into our setting:

Proposition 1 (Semantic Equivalence). *Let N be a neural network with L layers, ℓ a layer of N, i a neuron of this layer, and $B^{(\ell)} \subset \{1, \ldots, n_\ell\} \backslash \{i\}$ a chosen basis. Let $\tilde{N}$ be the NN after replacing neuron i by a linear combination of basis vectors with coefficients $\alpha^{(\ell)}_{i,j}$, with the procedure as described above.*

If for all inputs $\mathbf{x} \in X \subset \mathcal{X}$, $z^{(\ell)}_i(\mathbf{x}) = \sum_{j \in B^{(\ell)}} \alpha^{(\ell)}_{i,j} z^{(\ell)}_j(\mathbf{x})$, then N and $\tilde{N}$ are semantically equal, i.e. for all inputs $\mathbf{x} \in X$, $\tilde{N}(\mathbf{x}) = N(\mathbf{x})$.

It is easy to see that this proposition is true, for a full proof see Appendix A. However, the proposition assumes equality of $z^{(\ell)}_i(\mathbf{x})$ and $\sum_{j \in B^{(\ell)}} \alpha^{(\ell)}_{i,j} z^{(\ell)}_j(\mathbf{x})$ for $\mathbf{x} \in X$, which virtually never holds for real-world neural networks. Therefore, we want to minimize the difference $|z^{(\ell)}_i(\mathbf{x}) - \sum_{j \in B^{(\ell)}} \alpha^{(\ell)}_{i,j} z^{(\ell)}_j(\mathbf{x})|$, which will not yield a semantically equivalent abstraction, but an abstraction with very similar behavior. We can then **quantify the difference** between the output of the original network and the abstraction, i.e. the *induced error* with the following Theorem.

Theorem 1 (Over-approximation Guarantee). *Let N be an NN with L layers. For each layer ℓ, we have a basis of neurons $B^{(\ell)}$, and a set of replaced neurons $I^{(\ell)}$. Then, let $\tilde{N}$ be the network after replacing neurons in $I^{(\ell)}$ as described above.*

We can over-approximate the error *between the output of the original network N^L and the output of the abstraction $\tilde{N}^L$ for $\mathbf{x} \in X \subset \mathcal{X}$ by*

$$\|\tilde{N}^L(\mathbf{x}) - N^L(\mathbf{x})\| \leq b(1 - a^{L-1})/(1 - a)$$

with $a = \lambda(\|W\| + \eta)$, $b = \lambda\|W\|\epsilon$, with $\lambda^{(\ell)}$ being the Lipschitz-constant of the activation function in layer ℓ, $\lambda = \max_\ell \lambda^{(\ell)}$, $\|W\| = \max_\ell \|W^{(\ell)}\|_1$, $\eta = \max_\ell \eta^{(\ell)}$, and $\epsilon = \max_\ell \epsilon^{(\ell)}$, assuming that for all layers $\ell \in \{1, ..., L\}$ and for all inputs $\mathbf{x} \in X$, we have

- *for $i \in I^{(\ell)}$: $|z^{(\ell)}_i(\mathbf{x}) - \sum_{j \in B^{(\ell)}} \alpha^{(\ell)}_{i,j} z^{(\ell)}_j(\mathbf{x})| \leq \epsilon^{(\ell)}$*
- *$|\sum_{i \in I^{(\ell)}} W^{(\ell)}_{*,i} \sum_{t \in B^{(\ell)}} \alpha^{(\ell)}_{i,t}| \leq \eta^{(\ell)}$*

In other words, we can over-approximate the difference in the output of the original and the abstracted network by a value that depends on the weight matrices, the activation function and the tightness of the abstracted neurons to their replacements. The full proof can be found in Appendix B. This Theorem provides us with the **theoretical guarantee** that, given our abstraction, we can provide a valid over-approximation of the output of the original network.

Comparison to the δ-bisimulation Let us recap the error definition from [19]. The difference of the bisimulation and the original network is bounded by $[(2a)^k - 1]b/(2a - 1)$, where $a = \lambda|S|\|W\|$ and $b = \lambda(|P|L(\mathcal{N})\|x\| + 1)\delta$[1]. In this notation, $|S|$ is the maximum number of neurons per layer in the whole network, $|P|$ the maximum number of neurons in the bisimulation (can be understood as the number of neurons in an abstraction), $L(\mathcal{N})$ is the maximum Lipschitz-constant of all layers, and δ is the maximum absolute difference of the bias and sum of the incoming weights.

The drawbacks of that approach are twofold: (i) the error is based on one specific input, and (ii) it makes use of the Lipschitz-constant of the whole network. Calculating the Lipschitz constant of an NN is still part of ongoing research [7] and not a trivial problem. In contrast, we improve on both. Our error calculation generalizes over a set of inputs. Additionally, we use local information, stored in the weight-matrices, to circumvent using the Lipschitz-constant of the NN.

4 Refinement

For certain inputs the abstraction might not reflect the behavior of the original network. For these inputs, so-called *counterexamples*, we may want to *refine* the abstraction, as opposed to starting the abstraction from the original network again. We consider an input to be a counterexample whenever the abstraction assigns it a different label than the original network. However, a counterexample can be any input that does not align with the specifications.

We propose to refine the abstraction by restoring some replaced neurons. To do this, we need to know which neurons should be replaced and how. We first briefly mention three heuristics to choose a neuron for restoration. Afterward, we explain how to restore a neuron. Note that the refinement offers more than a "roll-back" of the most recent step of the abstraction since it picks the step-to-be-rolled-back in retrospect reflecting all other steps, leading to a more informed choice. This could in principle be done directly in the abstraction phase, but at an infeasible cost of a huge look-ahead.

Refinement Heuristics We propose three different heuristics: difference-guided, gradient-guided, and look-ahead.

- The *difference-guided* refinement looks at the difference of a neuron in the original and its representation as a linear combination in the abstraction. It replaces the neuron with the largest difference.
- The *gradient-guided* refinement additionally takes the gradient of the NN into account, that is computed as in the training phase of the NN. This takes into account how the whole network would need to change to fix the counterexample.
- The *look-ahead* is the most greedy method and would try out every replaced neuron. It would check how much the network would improve if the neuron was replaced and then chooses the neuron with the highest improvement.

More details on the approaches can be found in Appendix D.

[1] Please note that this statement is slightly different from the paper ($(2a)^k$ instead of $(2/a)^k$), which we believe to be a typo in the paper.

Restoration of a Neuron The restoration principle can be seen as the counterpart of the replacement. Let $\tilde{\tilde{N}}$ be the network obtained by replacing several neurons in the original network N, where we want to restore a deleted neuron i of layer ℓ. To do this, we need not only to get the original neuron back, including its incoming and outgoing weights but also to remove the additional outgoing weights from the basis neurons. Intuitively, the restoration removes the linear combination, ensures that the original outgoing weights for the neuron are used, and adjusts the incoming weights of the neuron. We may have changed layer $\ell - 1$, and thus we cannot restore the original incoming weights of neuron i, but we have to adapt it to changes in the basis $B^{(\ell-1)}$. This can be done with the following changes:

- $\forall j \in B^{(\ell)}$: $\tilde{W}^{(\ell)}_{*,j} = \tilde{\tilde{W}}^{(\ell)}_{*,j} - \alpha_j W^{(\ell)}_{*,j}$
- $\tilde{W}^{(\ell)}_{*,i} = W^{(\ell)}_{*,i}$
- $\forall j \in B^{(\ell-1)}$: $\tilde{w}^{(\ell-1)}(i,j) = w^{(\ell-1)}(i,j) + d^{(\ell-1)}_{j,i}$

Afterward, we subtract $\alpha_j \cdot w^{(\ell)}(i,i')$ from $d^{(\ell)}_{j,i'}$ for $i' \in \{1,\dots,n_{\ell+1}\}$ and $j \in B^{(\ell)}$.

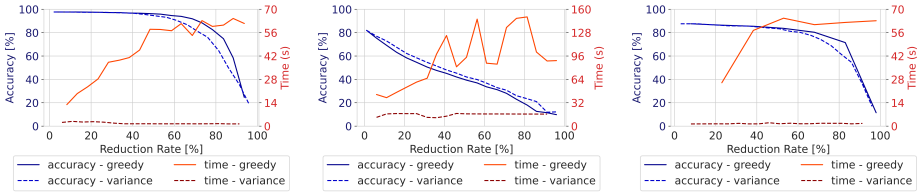
5 Experimental Results

Our experimental section is divided into several parts: The first one covers how the different methods for finding a basis and the coefficients compare, as described in Section 3.2 and Section 3.1. The second part shows experiments on our approach in comparison to existing works, namely *DeepAbstract* [2] and our implementation of bisimulation [19] (which was not implemented before). The third part contains the comparison between the abstraction based on syntactic and semantic information. The fourth part describes our experiments on abstraction refinement. Finally, the last part contains experiments on the error induced by our abstraction. Note that supplemental experiments can be found in the Appendix.

Lastly, the work of Katz et al. [6] tightly couples the abstraction with the subsequent particular verification, by integrating the specification as layers into the network. It is, thus, not clear how an abstraction from [6] could be extracted from the tool and reused for another purpose. Additionally, our abstraction would have to be connected with some verification algorithm (DeepPoly, as done by DeepAbstract, or some other) to compare. *Any comparison of the two works would then mostly compare the different verification tools, not really the abstractions.* Although a comparison of different verifiers linked to our LiNNA is an interesting next step into one of the possible applications, it is out of the scope of this paper, which examines the abstraction itself (see Introduction).

Implementation We implemented the approach in our tool *LiNNA* (*Li*near *N*eural *N*etwork *A*bstraction)[2]. We used networks that were trained on MNIST [16], CIFAR-10 [14], and FashionMNIST [29] for our experiments. In the following, we refer to the corresponding trained networks with "$L \times n$", where L denotes the number of hidden layers and n is the number of neurons in these hidden layers. All experiments were conducted on a computer with Ubuntu 22.04 LTS with 2.6 GHz Intel© Core™ i7 processors, and 32GB of RAM.

[2] https://github.com/cxlvinchau/LiNNA



(a) MNIST 3x100 (b) CIFAR-10 3x2500 (c) FashionMNIST 3x100

Fig. 2: *Finding the basis for replacement* - Evaluation on different datasets. The plots contain a comparison of LiNNA while using the greedy variant (solid) and the variance-based heuristic (dashed) for finding a basis with orthogonal projection. Comparison of accuracy (blue) in percent and computation time (red) in seconds.

Performance Measures We will compare the approaches mostly on (i) the reduction rate and (ii) the accuracy on a test set. Intuitively, the *reduction rate* describes how much the NN was reduced by abstraction. If an NN N has in total n neurons, but after reduction, there are m neurons left, then the reduction rate is then defined as $RR(N) = 1 - \frac{m}{n}$. The *accuracy* of a NN on a test set is defined as the ratio of how many inputs are predicted with the correct label. This is the key performance indicator in machine learning and shows how well a network generalizes to unseen data. In evaluating our abstraction, we follow the same principle since we want to know how well the NN generalizes after abstraction. Note that this test set was not used for training or computing the abstraction.

5.1 Abstraction

Finding the Basis We have given two different methods in Section 3.1 to find a good basis B. While the orthogonal projection yields an equally good abstraction compared to linear programming, it outperforms the latter in terms of runtime by magnitudes. Hence, we conducted the rest of the experiments with orthogonal projection. The full comparison between orthogonal projection and linear programming can be found in Fig. 14 in Appendix E.

When we compare the greedy and the heuristic-based approach, shown in Fig. 2, we see that the former outperforms the latter in terms of accuracy on MNIST and FashionMNIST. On CIFAR-10, the variance-based approach is slightly better. However, the variance-based approach is always faster than the greedy approach and scales better, as can be seen for all datasets. Unsurprisingly, the greedy approach takes more time for higher reduction rates, because it needs to evaluate many candidates. The variance-based approach just takes the best neurons according to their variance, which has to be calculated only once. Therefore, the calculation is constant in terms of removed neurons.

The plots show one more difference in the behavior: On MNIST and FashionMNIST, we see a quite stable accuracy until a reduction rate of 60%. We cannot see the same behavior on CIFAR-10. We believe this is due to the accuracy and size of the networks. Whereas it is fairly easy to train a feedforward network for MNIST and FashionMNIST on a regular computer, this is more challenging for CIFAR-10. We plan to include more extensive experiments including more

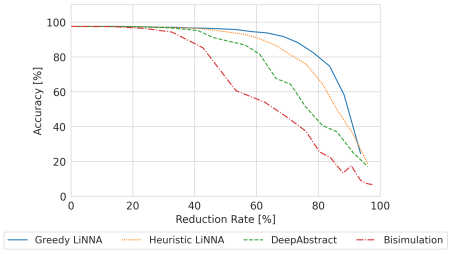


Fig. 3: *Comparison of LiNNA to related work* - LiNNA (greedy and heuristic-based variant), *DeepAbstract* [2], and our implementation of the bisimulation [19] is evaluated in terms of accuracy on the test set for a certain reduction rate. The experiment was conducted on an MNIST 3x100 network.

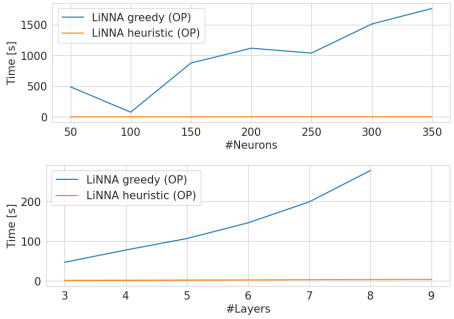


Fig. 4: *Scalability of LiNNA* - Average runtime for 20 different reduction rates on one network. The plot at the top depicts the runtime for MNIST networks with 4 layers, w.r.t. number of neurons. The plot at the bottom shows the runtime for MNIST networks with 100 neurons per layer, w.r.t. number of layers.

involved NN architectures in future work. Finally, our abstraction relies on the assumption that NNs contain a lot of redundant information. We want to emphasize, that in machine learning, it is common to train a huge network that contains many more neurons than necessary to solve the task. Our approach can detect these cases and abstract away this redundant information.

Finding the Coefficients We have in total four different approaches to finding the coefficients: greedy or heuristic-based linear programming, and greedy or heuristic-based orthogonal projection. All four have similar accuracies for the same reduction rate, whereas the heuristic ones are mostly just slightly worse than the greedy ones. For a more detailed evaluation, please refer to Appendix G.

The runtimes of the four approaches, however, differ a lot. Take for example an MNIST 3x100 network. We assume that the abstraction is performed by starting with the full network and reducing up to a certain reduction rate. Thus, we have runtimes for each of the approaches for each reduction rate. We take the average over all the reductions and get 47s for the greedy orthogonal projection, 5130s for the greedy linear programming, 1s for the heuristic orthogonal projection, and 2s for the heuristic linear programming. Linear programming takes a lot more time than orthogonal projection, and, as already seen before, the heuristic approaches are much faster than the greedy ones. Please refer to Appendix J for more experiments on the runtime. Therefore, we propose to use the heuristic approach and the orthogonal projection.

Scalability We evaluate how our approach scales to networks of different sizes. We evaluate (1) how our approach scales with an increasing number of layers, and (2) how it scales with a fixed number of layers but an increasing number of neurons. We show our experiments in Fig. 4. The runtime is the average runtime over 20 different reduction rates on the same network. One can

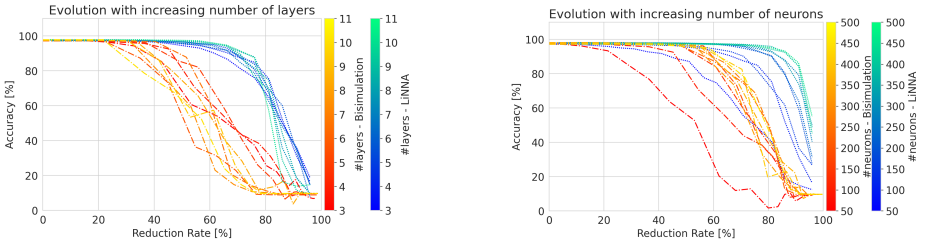


Fig. 5: *Evolution of the accuracy* on the test set for different reduction rates, for an increasing number of layers, or neurons. We show LiNNA (blue-green) for semantic abstraction, and for syntactic abstraction, bisimulation (red-yellow). The networks were trained on MNIST and have a fixed number of neurons (100) on the left, and a fixed number of layers (4) on the right.

imagine this as averaging the runtimes shown in Fig. 2. We can see that the variance-based approach has almost constant runtime, whereas the runtime of the greedy approach is increasing for both a higher number of layers and neurons.

Final Assessment We have four possibilities on how to abstract an NN: greedy orthogonal projection, greedy linear programming, heuristic-based orthogonal projection, and heuristic-based linear programming. Given that the orthogonal projection outperforms linear programming in terms of accuracy and computation time, we propose to use orthogonal projection. We believe that it is sufficient to use the heuristic-based approach, thereby gaining faster runtimes and only barely sacrificing any accuracy. Whenever we refer to *LiNNA* from now on without any additions, it will be the heuristic-based orthogonal projection.

5.2 Comparison to Existing Work

We want to show how our approach compares to existing works, i.e. *DeepAbstract* and the *bisimulation*. Since there is no implementation available for the latter, we implemented it ourselves. Please refer to Appendix F for the details. The results of the comparison are shown in Fig. 3. It is evident that DeepAbstract achieves higher accuracies than the bisimulation, but LiNNA outperforms DeepAbstract and the bisimulation in terms of accuracy for all reduction rates.

Concerning the runtime, we measure the runtime of each approach for a certain reduction rate, starting from the full network. We find that (in the median) LiNNA (greedy) needs 55s up to 199s, LiNNA (heuristic) 2s up to 3s, DeepAbstract 187s up to 2420s, and the bisimulation 1s up to 2s, on MNIST networks of different sizes (starting from 4x50 up to 11x100). The details can be found in Appendix J. The bisimulation performs best, however just slightly ahead of the heuristic-based LiNNA. The greedy LiNNA, as well as DeepAbstract both have a much higher computation time.

However, in terms of accuracy, greedy LiNNA seems to be the best-performing approach, given sufficient time. Due to efficiency, we suggest using heuristic-based LiNNA, as it is as fast as the bisimulation, but its accuracy is a lot better and even close to greedy LiNNA.

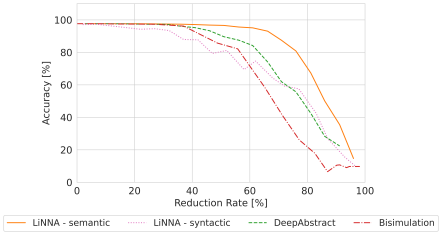


Fig. 6: *Syntactic VS. Semantic* - This plot shows the difference between using semantic resp. syntactic information for the abstraction on an MNIST 5x100 network. Semantic: LiNNA (semantic) and DeepAbstract. Syntactic: LiNNA (syntactic) and the bisimulation.

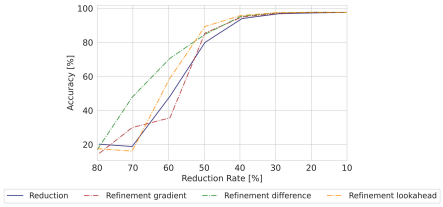


Fig. 7: *Refinement* - This plot shows the accuracy of an MNIST 5x100 network that was abstracted and refined to a certain reduction rate R. There is also a plot for an abstraction to the same reduction rate as after the refinement but without refining.

Since we are interested in the general behavior of the abstraction, we want to see how the methods work for varying sizes of networks, but not only in terms of scalability. In Fig. 5, we show the trend for bisimulation and LiNNA for an increasing number of layers resp. neurons per layer. On the left, we fix the number of neurons per layer to 100 and incrementally increase the number of layers. On the right, we fix the number of layers to four and increase the number of neurons.

We can see that the performance of the networks from the bisimulation varies a lot and gets slightly worse when there are more layers, whereas LiNNA has a very small variation and the performance of the abstractions increases slightly for more layers. Both approaches compute abstractions that perform better the more neurons are in a layer, but LiNNA converges to a much steeper curve at high reduction rates.

For NNs with 400 or more neurons, LiNNA can reduce 80% of the neurons without a significant loss in accuracy, whereas the bisimulation can do the same only for up to a reduction rate of 55%.

5.3 Semantic vs Syntactic

In the following, we want to show the differences between semantic and syntactic abstractions. Recall that syntactic abstraction makes use of the weights of the network, the syntactic information, with no consideration of the actual behavior of the NN on the inputs. Semantic abstraction, on the other hand, focuses on the values of the neurons on an input dataset, which also incorporates information about the weights. DeepAbstract and LiNNA, both use semantic information, whereas bisimulation uses syntactic information. We additionally evaluate the performance of LiNNA on syntactic information.

Which type of information is better for abstraction: semantic or syntactic? Note that both DeepAbstract and the bisimulation represent a group of neurons by one single representative, whereas LiNNA makes use of a linear combination. We summarize our results in Fig. 6. For smaller reduction rates, the bisimulation performs better than LiNNA on syntactic information; for higher reduction rates

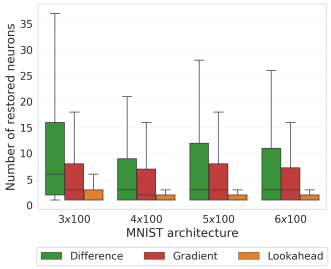


Fig. 8: *Comparison of refinement techniques* on different architectures for MNIST. The respective networks were abstracted with a reduction rate of 50%. The lines show the variance, the box represents 50% of the data, the line in the box shows the median.

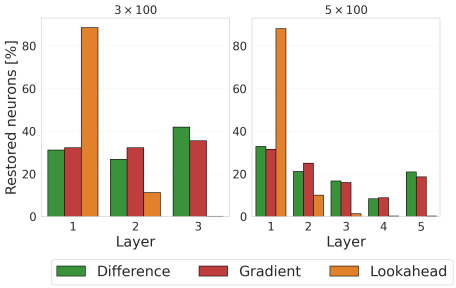


Fig. 9: *Refinement on different layers* - We considered abstractions that were obtained with a 50% reduction rate and fixed 1000 counterexamples. The plots depict the percentage of restored neurons in the layers of the different MNIST networks.

it is reversed. In general, the approaches based on semantics (DeepAbstract and LiNNA - semantic) outperform the other two approaches w.r.t. accuracy. While abstraction based on syntactic information can provide global guarantees for any input, abstraction based on semantic information relies on the fact that its inputs during abstraction are similar to the ones it will be evaluated on later. However, we see that still the semantic information is more appropriate for preserving accuracy because it combines the knowledge about possible inputs with the knowledge about the weights.

5.4 Refining the Network

We propose refinement of the abstraction in cases where it does not capture all the behavior anymore, instead of restarting the abstraction process. We consider networks that are abstracted up to certain reduction rates, i.e. 20%, 30%, ..., 90%, and use the refinement to regain 10% of the neurons. For example, we reduce the network by 90% and then use refinement to get back to a reduction rate of 80%. We evaluate this refined network on the test dataset and plot its accuracy. Additionally, we show the accuracy of the same NN which is directly reduced to an 80% reduction rate, without refinement. This plot is shown in Fig. 7 for a 5x100 network, trained on MNIST.

The gradient and look-ahead refinement have a similar performance. However, the difference-based approach even outperforms the direct reduction itself. This behavior can be explained by the fact that the refinement and the abstraction look at different metrics for removing/restoring neurons. The refinement can focus directly on optimizing for the inputs at hand, whereas the abstraction was generated on the training set. In conclusion, the refinement can even improve the abstraction and it is beneficial to abstract slightly more than required, and refine for the relevant inputs, rather than having a finer abstraction directly.

Comparison of the Different Approaches We collect images that are labeled differently by the abstraction and observe the number of neurons that are restored in order to fix the classification of each image. We ran the experiment on different networks that were abstracted with a 50% reduction rate and considered 1000 counterexamples for each network. The results are summarized in Fig. 8, where we have boxplots for each refinement method on four different network architectures. The look-ahead approach is the most effective technique since it requires the smallest number of restored neurons. In the median, it only requires 1 to 2 operations. The gradient-based approach performs noticeably worse but outperforms the difference-based approach on all networks. The computation time, however, gives a different perspective: Repairing one counterexample takes on average <1s for the difference-based approach, 1s for the gradient-based, but the look-ahead approach takes on average 4s. Interestingly, the look-ahead approach restores fewer neurons but performs worse in accuracy. The difference-based performs better in terms of accuracy while restoring more neurons.

Insight on the Relevance of Layers We also investigated in which layers the different refinement techniques tend to restore the neurons. The plots in Fig. 9 illustrate the percentage of restored neurons in each layer. Notably, the look-ahead approach restores most neurons in the first layer, and very few or none in the later layers, whereas the other approaches have a more uniform behavior. However, the more layers the network has, the more the gradient- and difference-based approaches tend to restore more neurons in the first layer. As reported already by [2], the first layers seem to have a larger influence on the network's output and hence should be focused on during refinement. It is even more interesting that the difference-based approach does not focus on the first layers as much as the look-ahead approach, but it is better in terms of accuracy.

5.5 Error Calculation

We want to show how the abstraction simulates the original network on unseen data not only w.r.t. the output but on every single neuron. In other words, is the discrepancy between the concrete and abstract network higher on the *test* data than on the *training* data that generate the abstraction, or does the link between the neuron and its linear abstraction generalize well?

In Fig. 10, we look at this ratio ("relative error of the abstraction"), i.e. the absolute difference of (activation values of) a simulated abstract neuron to the original neuron, once on the test dataset divided by the maximum value on the training dataset. We can see that there are cases where the error can be greater than one (meaning "larger than on the training set"), see the first row of the plot. However, the geometric mean, defined as $\left(\Pi_{i=1}^{N} a_i\right)^{\frac{1}{N}}$, calculated over all images is very small. Note that more experiments can be found in Appendix L. In conclusion, we can say that our abstraction is close to the original also on the test dataset, although the theoretical error calculation does not guarantee so tight a simulation. Future work should reveal how to further utilize the empirical proximity in transferring the reasoning from the abstraction to the original.

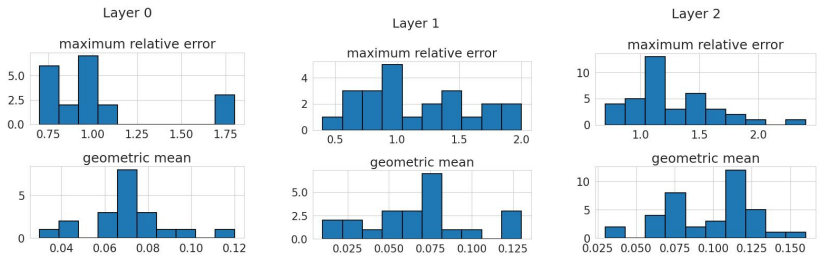


Fig. 10: *Histograms of the relative error* of the values of the neurons in an MNIST 3x100 network and its abstraction (reduced by 30%). The first row shows the maximum relative error of each neuron in the NN, that occurred for some input from the test set. The second row shows the geometric mean of the relative error of each neuron over 100 images of the test set.

6 Conclusions

The focus of this work was to examine abstraction not as a part of a verification procedure, but rather as a stand-alone transformation, which can later be used in different ways: as a preprocessing step for verification, as means of obtaining an equivalent smaller network, or to gain insights about the network and its training, such as identifying where redundancies arise in trained neural networks. (This is analogous to the situation of bisimulation, which has been largely investigated on its own not necessarily as a part of a verification procedure, and its use in verification is only one of the applications.)

We have introduced *LiNNA*, which abstracts a network by replacing neurons with *linear combinations* of other neurons and also equip it with a *refinement* method. We bound the error and thus the difference between the abstraction and the original network in Theorem 1. The theorem yields a lower and an upper bound on the network's output, thereby providing its over-approximation.

We showed that the linear extension provides better performance than existing work on abstraction for classification networks, both DeepAbstract, and the bisimulation-based approach. We focused our experimental evaluation on *accuracy*, since the aim of the abstraction is to faithfully mimic the *whole classification process* in the smaller, abstract network, not just one concrete property to be verified, which describes only a very specific aspect of the network. Interestingly, the practical error is dramatically smaller than the worst-case bounds. We hope this first, experimental step will stimulate interest in research that could utilize this actual advantage, which is currently not supported by any respective theory.

Furthermore, we show that the use of *semantic information should be preferred* over syntactic information because it allows for higher reductions while preserving similar behavior and being cheap since the I/O sets can be quite small. Bringing back semantics could take us closer to the efficiency of classical software abstraction, where the semantics of states is the very key, going way beyond bisimulation.

References

[1] J. Altschuler et al. "Greedy Column Subset Selection: New Bounds and Distributed Algorithms". In: ICML'16.

[2] P. Ashok, V. Hashemi, J. Křetínskỳ, and S. Mohr. "Deepabstract: Neural network abstraction for accelerating verification". In: *ATVA'20*.

[3] C. M. Bishop. *Pattern recognition and machine learning*. Springer, 2006.

[4] C. Brix et al. "First Three Years of the International Verification of Neural Networks Competition (VNN-COMP)". In: *CoRR* abs/2301.05815 (2023). DOI: 10.48550/arXiv.2301.05815. arXiv: 2301.05815. URL: https://doi.org/10.48550/arXiv.2301.05815.

[5] Y. Cheng, D. Wang, P. Zhou, and T. Zhang. "A survey of model compression and acceleration for deep neural networks". In: *arXiv preprint arXiv:1710.09282* (2017).

[6] Y. Y. Elboher, J. Gottschlich, and G. Katz. "An Abstraction-Based Framework for Neural Network Verification". In: *CAV'20*.

[7] M. Fazlyab et al. "Efficient and accurate estimation of lipschitz constants for deep neural networks". In: *Advances in Neural Information Processing Systems* (2019).

[8] Y. Gong, L. Liu, M. Yang, and L. Bourdev. "Compressing deep convolutional networks using vector quantization". In: *arXiv preprint arXiv:1412.6115* (2014).

[9] Gurobi Optimization, LLC. *Gurobi Optimizer Reference Manual*. 2022. URL: https://www.gurobi.com.

[10] G. Hinton, O. Vinyals, J. Dean, et al. "Distilling the knowledge in a neural network". In: *NeurIPS Deep Learning Workshop* (2014).

[11] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger. "Densely connected convolutional networks". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 4700–4708.

[12] G. Katz et al. "The Marabou Framework for Verification and Analysis of Deep Neural Networks". In: *Computer Aided Verification*. Ed. by I. Dillig and S. Tasiran. Cham: Springer International Publishing, 2019, pp. 443–452. ISBN: 978-3-030-25540-4.

[13] J. R. Kirkwood and B. H. Kirkwood. *Elementary Linear Algebra*. Chapman and Hall/CRC, 2017.

[14] A. Krizhevsky, G. Hinton, et al. "Learning multiple layers of features from tiny images". In: (2009).

[15] S. Lawrence, C. L. Giles, and A. C. Tsoi. "Lessons in neural network training: Overfitting may be harder than expected". In: *AAAI/IAAI*. 1997, pp. 540–545.

[16] Y. LeCun and C. Cortes. "MNIST handwritten digit database". In: (2010). URL: http://yann.lecun.com/exdb/mnist/.

[17] A. L. Maas, A. Y. Hannun, and A. Y. Ng. "Rectifier nonlinearities improve neural network acoustic models". In: *ICML'13*.

[18] A. Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *NeurIPS'19*.

[19] P. Prabhakar. “Bisimulations for Neural Network Reduction”. In: *VMCAI'22*. Ed. by B. Finkbeiner and T. Wies.
[20] P. Prabhakar and Z. R. Afzal. “Abstraction based Output Range Analysis for Neural Networks”. In: *NeurIPS*. 2019.
[21] Y. Shitov. “Column subset selection is NP-complete”. In: *Linear Algebra and its Applications* (2021), pp. 52–58.
[22] G. Singh, T. Gehr, M. Püschel, and M. T. Vechev. “An abstract domain for certifying neural networks”. In: *POPL'19*.
[23] G. Singh, T. Gehr, M. Püschel, and M. T. Vechev. “Boosting Robustness Certification of Neural Networks”. In: *ICLR (Poster)*. 2019.
[24] M. Sotoudeh and A. V. Thakur. “Abstract neural networks”. In: *SAS'20*.
[25] H.-D. Tran et al. “Robustness verification of semantic segmentation neural networks using relaxed reachability”. In: *CAV'21*.
[26] A. Virmaux and K. Scaman. “Lipschitz regularity of deep neural networks: analysis and efficient estimation”. In: *NeurIPS'18*.
[27] Q. Wang, Y. Ma, K. Zhao, and Y. Tian. “A Comprehensive Survey of Loss Functions in Machine Learning”. In: *Annals of Data Science* (2020).
[28] S. Wang et al. “Beta-CROWN: Efficient bound propagation with per-neuron split constraints for complete and incomplete neural network verification”. In: *Advances in Neural Information Processing Systems* 34 (2021).
[29] H. Xiao, K. Rasul, and R. Vollgraf. “Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms”. In: *arXiv preprint arXiv:1708.07747* (2017).
[30] K. Xu et al. “Fast and Complete: Enabling Complete Neural Network Verification with Rapid and Massively Parallel Incomplete Verifiers”. In: *International Conference on Learning Representations*. 2021.
[31] J. Xue, J. Li, and Y. Gong. “Restructuring of deep neural network acoustic models with singular value decomposition.” In: *Interspeech*. 2013, pp. 2365–2369.
[32] M. L. Zepeda-Mendoza and O. Resendis-Antonio. “Hierarchical Agglomerative Clustering”. In: *Encyclopedia of Systems Biology*. 2013.

Appendix

A Proof of Proposition 1

Proof. We show that $\mathbf{z}^{(\ell+1)}(\mathbf{x}) = \tilde{\mathbf{z}}^{(\ell+1)}(\mathbf{x})$ for all $\mathbf{x} \in X$. After this, it follows immediately that $\tilde{N}(\mathbf{x}) = N(\mathbf{x})$.

Let $\mathbf{x} \in X$ be fixed. We write $\mathbf{z}^{(\ell+1)}$ instead of $\mathbf{z}^{(\ell+1)}(\mathbf{x})$ to abbreviate. We have $\mathbf{z}^{(\ell+1)} = \phi(W^{(\ell)}\mathbf{z}^{(\ell)} + \mathbf{b}^{(\ell+1)})$ and $\tilde{\mathbf{z}}^{(\ell+1)} = \phi(\tilde{W}^{(\ell)}\mathbf{z}^{(\ell)} + \mathbf{b}^{(\ell+1)})$, since $\mathbf{z}^{(\ell)}$ has not changed.

For each neuron s of layer $\ell+1$, we have:

$$
\begin{aligned}
\tilde{z}_s^{(\ell+1)} &= \phi(\tilde{W}_{s,*}^{(\ell)}\mathbf{z}^{(\ell)} + b_s^{(\ell+1)}) \\
&= \phi(\sum_{t\in\{1,\dots,n_\ell\}} \tilde{w}^{(\ell)}(s,t)z_t^{(\ell)} + b_s^{(\ell+1)}) \\
&= \phi(\sum_{t\notin B^{(\ell)}, t\neq i} w^{(\ell)}(s,t)z_t^{(\ell)} + \sum_{t\in B^{(\ell)}} \tilde{w}^{(\ell)}(s,t)z_t^{(\ell)} + \tilde{w}^{(\ell)}(s,i)z_i^{(\ell)} + b_s^{(\ell+1)}) \\
&= \phi(\sum_{t\notin B^{(\ell)}, t\neq i} w^{(\ell)}(s,t)z_t^{(\ell)} + \sum_{t\in B^{(\ell)}} (w^{(\ell)}(s,t) + \alpha_t w^{(\ell)}(s,i))z_t^{(\ell)} + \tilde{w}^{(\ell)}(s,i)z_i^{(\ell)} + b_s^{(\ell+1)}) \\
&= \phi(\sum_{t\notin B^{(\ell)}, t\neq i} w^{(\ell)}(s,t)z_t^{(\ell)} + \sum_{t\in B^{(\ell)}} w^{(\ell)}(s,t)z_t^{(\ell)} + \sum_{t\in B^{(\ell)}} \alpha_t w^{(\ell)}(s,i)z_t^{(\ell)} + b_s^{(\ell+1)}) \\
&= \phi(\sum_{t\notin B^{(\ell)}, t\neq i} w^{(\ell)}(s,t)z_t^{(\ell)} + \sum_{t\in B^{(\ell)}} w^{(\ell)}(s,t)z_t^{(\ell)} + w^{(\ell)}(s,i)z_i^{(\ell)} + b_s^{(\ell+1)}) \\
&= \phi(\sum_{t\in\{1,\dots,n_\ell\}} w^{(\ell)}(s,t)z_t^{(\ell)} + b_s^{(\ell+1)}) \\
&= z_s^{(\ell+1)}
\end{aligned}
$$

□

B Proof of Theorem 1

Following the idea of [19], we first introduce a lemma that shows the difference of the abstraction to the original for one layer and then show how the theorem follows from that.

Lemma 1 (Replacement Error of One Layer). *Let N be a neural network with L layers. For one layer ℓ, we have a basis of neurons $B^{(\ell)}$, and a set of replaced neurons $I^{(\ell)}$. Then, let $\tilde{N}$ be the network after replacing neurons in $I^{(\ell)}$ as described in Section 3.3. Furthermore, let $\lambda^{(\ell)}$ be the Lipschitz constant of the activation function of that layer. If for all inputs $\mathbf{x} \in X \subset \mathcal{X}$, we have*

1. *for $i \in I^{(\ell)}$: $|z_i^{(\ell)}(\mathbf{x}) - \sum_{j\in B^{(\ell)}} \alpha_{i,j}^{(\ell)} z_j^{(\ell)}(\mathbf{x})| \leq \epsilon^{(\ell)}$*
2. *$|\sum_{i\in I^{(\ell)}} W_{*,i}^{(\ell)} \sum_{t\in B^{(\ell)}} \alpha_{i,j}^{(\ell)}| \leq \eta^{(\ell)}$*
3. *$\|\tilde{\mathbf{z}}^{(\ell)}(\mathbf{x}) - \mathbf{z}^{(\ell)}(\mathbf{x})\| \leq \delta^{(\ell)}$*

then, we get $\|\tilde{\mathbf{z}}^{(\ell+1)} - \mathbf{z}^{(\ell+1)}\| \leq \lambda^{(\ell)}(\|W^{(\ell)}\|_1\delta^{(\ell)} + \|W^{(\ell)}\|_1\epsilon^{(\ell)} + \eta^{(\ell)}\delta^{(\ell)})$

Please note that the error value ϵ talks about the differences in the values of the neurons before and after replacement, but without any change to any other layer. In constrast, the term δ references the difference between the values of the neurons before and after changing layers before layer ℓ.

Proof (of Lemma 1). Let $\mathbf{x} \in X$ be fixed. We write $\mathbf{z}^{(\ell+1)}$ instead of $\mathbf{z}^{(\ell+1)}(\mathbf{x})$ to abbreviate. We have $\mathbf{z}^{(\ell+1)} = \phi(\mathbf{h}^{(\ell+1)})$ and similarly $\tilde{\mathbf{z}}^{(\ell+1)} = \phi(\tilde{\mathbf{h}}^{(\ell+1)})$. It is

$$\begin{aligned}
\|\tilde{\mathbf{h}}^{(\ell+1)} - \mathbf{h}^{(\ell+1)}\| &= \|\tilde{W}^{(\ell)}\tilde{\mathbf{z}}^{(\ell)} + \mathbf{b}^{(\ell+1)} - (W^{(\ell)}\mathbf{z}^{(\ell)} + \mathbf{b}^{(\ell+1)})\| \\
&= \| \sum_{t \in B^{(\ell)}} \left(\tilde{W}^{(\ell)}_{*,t} \tilde{z}^{(\ell)}_t \right) + \mathbf{b}^{(\ell+1)} - \sum_{t \in B^{(\ell)} \cup I^{(\ell)}} \left(W^{(\ell)}_{*,t} z^{(\ell)}_t \right) - \mathbf{b}^{(\ell+1)} \| \\
&= \| \sum_{t \in B^{(\ell)}} \left(\tilde{W}^{(\ell)}_{*,t} \tilde{z}^{(\ell)}_t - W^{(\ell)}_{*,t} z^{(\ell)}_t \right) - \sum_{t \in I^{(\ell)}} \left(W^{(\ell)}_{*,t} z^{(\ell)}_t \right) \|
\end{aligned}$$

Using the replacement $\tilde{W}^{(\ell)}_{*,j} = W^{(\ell)}_{*,j} + \alpha^{(\ell)}_{i,j} W^{(\ell)}_{*,i}$ from (5)

$$\begin{aligned}
&= \| \sum_{t \in B^{(\ell)}} \left(\left(W^{(\ell)}_{*,t} + \sum_{i \in I^{(\ell)}} \alpha^{(\ell)}_{i,t} W^{(\ell)}_{*,i} \right) \tilde{z}^{(\ell)}_t - W^{(\ell)}_{*,t} z^{(\ell)}_t \right) - \sum_{t \in I^{(\ell)}} \left(W^{(\ell)}_{*,t} z^{(\ell)}_t \right) \| \\
&= \| \sum_{t \in B^{(\ell)}} \left(W^{(\ell)}_{*,t} (\tilde{z}^{(\ell)}_t - z^{(\ell)}_t) \right) + \sum_{i \in I^{(\ell)}} \left(\alpha^{(\ell)}_{i,t} W^{(\ell)}_{*,i} \tilde{z}^{(\ell)}_t \right) - \sum_{t \in I^{(\ell)}} \left(W^{(\ell)}_{*,t} z^{(\ell)}_t \right) \|
\end{aligned}$$

with 3., we get

$$\begin{aligned}
&\leq \| \sum_{t \in B^{(\ell)}} \left(W^{(\ell)}_{*,t} \delta^{(\ell)} \right) + \sum_{i \in I^{(\ell)}} \sum_{t \in B^{(\ell)}} \left(\alpha^{(\ell)}_{i,t} W^{(\ell)}_{*,i} \tilde{z}^{(\ell)}_t - W^{(\ell)}_{*,i} z^{(\ell)}_i \right) \| \\
&\leq \| \sum_{t \in B^{(\ell)}} \left(W^{(\ell)}_{*,t} \delta^{(\ell)} \right) + \sum_{i \in I^{(\ell)}} \sum_{t \in B^{(\ell)}} \left(\alpha^{(\ell)}_{i,t} W^{(\ell)}_{*,i} (z^{(\ell)}_t + \delta^{(\ell)}) - W^{(\ell)}_{*,i} z^{(\ell)}_i \right) \| \\
&\leq \| \sum_{t \in B^{(\ell)}} \left(W^{(\ell)}_{*,t} \delta^{(\ell)} \right) + \sum_{i \in I^{(\ell)}} \left(W^{(\ell)}_{*,i} \sum_{t \in B^{(\ell)}} \left(\alpha^{(\ell)}_{i,t} z^{(\ell)}_t \right) - z^{(\ell)}_i + \sum_{t \in B^{(\ell)}} \left(\alpha^{(\ell)}_{i,t} W^{(\ell)}_{*,i} \delta^{(\ell)} \right) \right) \|
\end{aligned}$$

with 1., we get

$$\leq \|\delta^{(\ell)} \sum_{t \in B^{(\ell)}} \left(W^{(\ell)}_{*,t} \right) + \epsilon^{(\ell)} \sum_{i \in I^{(\ell)}} \left(W^{(\ell)}_{*,i} \right) + \delta^{(\ell)} \sum_{i \in I^{(\ell)}} \sum_{t \in B^{(\ell)}} \left(\alpha^{(\ell)}_{i,t} W^{(\ell)}_{*,i} \right) \|$$

with 2., we get

$$\begin{aligned}
&\leq \|\delta^{(\ell)} \sum_{t \in B^{(\ell)}} \left(W^{(\ell)}_{*,t} \right) + \epsilon^{(\ell)} \sum_{i \in I^{(\ell)}} \left(W^{(\ell)}_{*,i} \right) + \eta^{(\ell)} \delta^{(\ell)} \| \\
&\leq \|W^{(\ell)}\|_1 \delta^{(\ell)} + \|W^{(\ell)}\|_1 \epsilon^{(\ell)} + \eta^{(\ell)} \delta^{(\ell)}
\end{aligned}$$

From the Lipschitz-continuity of ϕ, we get

$$\|\tilde{\mathbf{z}}^{(\ell+1)} - \mathbf{z}^{(\ell+1)}\| \leq \lambda^{(\ell)} (\|W^{(\ell)}\|_1 \delta^{(\ell)} + \|W^{(\ell)}\|_1 \epsilon^{(\ell)} + \eta^{(\ell)} \delta^{(\ell)})$$

□

Proof (of Theorem 1). We define $\lambda = \max_\ell \lambda^{(\ell)}$, $\|W\| = \max_\ell \|W^{(\ell)}\|_1$, $\eta = \max_\ell \eta^{(\ell)}$, and $\epsilon = \max_\ell \epsilon^{(\ell)}$. The first layer, the input layer, cannot be changed, thus we have $\delta^{(1)} = 0$.

We set $a = \lambda(\|W\| + \eta)$ and $b = \lambda \|W\| \epsilon$.

By Lemma 1, we have for the last layer L that
$\|\tilde{\mathbf{z}}^{(L)} - \mathbf{z}^{(L)}\| \leq \lambda^{(L-1)}(\|W^{(L-1)}\|_1 \delta^{(L-1)} + \|W^{(L-1)}\|_1 \epsilon^{(L-1)} + \eta^{(L-1)}\delta^{(L-1)})$

It holds that $\|\tilde{\mathbf{z}}^{(L)} - \mathbf{z}^{(L)}\| \leq a\delta^{(L-1)} + b$, since a and b consist of the maxima over all layers. Unrolling this leads to

$$a\delta^{(L-1)} + b \leq a(a\delta^{(L-2)} + b) + b \leq ... \leq a^L \delta^{(1)} + \sum_{i=0}^{L-2} ba^i$$

and from $\delta^{(1)} = 0$, we remain with $\sum_{i=0}^{L-2} ba^i = b(1 - a^{L-1})/(1 - a)$. □

C Linear Program for Finding the Coefficients

We want to minimize $\| \sum_{j \in B^{(\ell)}} \alpha_{i,j}^{(\ell)} \cdot \mathbf{Z}_{j,*}^{(\ell)} - \mathbf{Z}_{i,*} \|$ for $\alpha_{i,j}^{(\ell)}$. In the following, we describe the linear program for computing optimal coefficients $\alpha_{i,j}^{(\ell)}$.

$$\begin{aligned} \text{minimize} \quad & \mathbf{1}^\top \boldsymbol{\beta}^+ + \mathbf{1}^\top \boldsymbol{\beta}^- \\ \text{subject to} \quad & \textstyle\sum_{j \in B} \mathbf{z}_j^{(\ell)} \alpha_{i,j}^{(\ell)} - \mathbf{I}_d \boldsymbol{\beta}^+ + \mathbf{I}_d \boldsymbol{\beta}^- = \mathbf{z}_i^{(\ell)} \\ & \alpha_{i,j}^{(\ell)} \in \mathbb{R} \text{ for } j \in B, \;\; \boldsymbol{\beta}^+, \boldsymbol{\beta}^- \in \mathbb{R}^d, \;\; \boldsymbol{\beta}^+, \boldsymbol{\beta}^- \geq 0 \end{aligned} \tag{6}$$

We use $\mathbf{1}$ to denote a vector containing only ones and $\mathbf{I}_d$ to denote the identity matrix with d rows and columns. Since the equation $\| \sum_{j \in B^{(\ell)}} \alpha_{i,j}^{(\ell)} \cdot \mathbf{Z}_{j,*}^{(\ell)} - \mathbf{Z}_{i,*} \|$ typically does not have an exact solution, we introduce the *slack variables* $\boldsymbol{\beta}^+$ and $\boldsymbol{\beta}^-$. This is a common method to create a relaxed optimization problem for which solutions exist. Note that the sum of the two slack variables gives us the L1 distance between the obtained linear combination and the actual activation. Particularly, this means that the optimal coefficients $\alpha_{i,j}^{(\ell)}$ give us the best solution w.r.t. L_1 distance.

D Refinement Heuristics

Difference-guided Refinement Let $\mathbf{x}$ be a counterexample. The difference guided refinement considers the difference between a neuron $\mathbf{z}_i^{(\ell)}(\mathbf{x})$ in the original network and the linear combination $\sum_{j \in B^{(\ell)}} \alpha_j \mathbf{z}_j^{(\ell)}(\mathbf{x})$ with which it was replaced in the abstracted network. We evaluate this for all neurons and restore the neuron with the largest difference. Intuitively, a large difference suggests that the linear combination does not accurately capture the behavior of the neuron on the given counterexample.

Gradient-guided Refinement The gradient-guided refinement follows a similar approach, but additionally, we take the gradient into account. Suppose the original network outputs $\mathbf{y}$ and the reduced network $\vec{\tilde{y}}$. We can quantify the error with a typical loss function $g(\mathbf{y}, \tilde{\mathbf{y}})$ for NN outputs , e.g. mean-squared error or cross-entropy loss [27]. For the neurons in the basis $j \in B^{(\ell)}$ and all neurons of the layers before, $j' \in \{1, \ldots, n_{\ell-1}\}$, we then compute the gradient of the loss function w.r.t. their incoming weights. That is we compute the gradient $\dfrac{\mathrm{d}g}{\mathrm{d}w^{(\ell)}(j, j')}$, which can be done with the *back-propagation* algorithm as generally used for training a NN [3]. Afterward, we simulate a single optimization step with gradient descent. Consequently, the values of the neurons in the basis change

and, thus, also the value of the linear combination of a replaced neuron i. Let us denote the new value of the linear combination for neuron i by $\bar{z}_i$. We evaluate $(\bar{z}_i - z_i^\ell) \cdot (\sum_{j \in B^{(\ell)}} \alpha_j z_j^\ell - z_i^\ell)$ on the counterexample. The first factor $\bar{z}_i - z_i^\ell$ describes the difference between the updated linear combination and the actual output and it indicates whether a neuron's output value needs to be decreased or increased to minimize the loss function. The second factor $\sum_{j \in B^{(\ell)}} \alpha_j z_j^\ell - z_i^\ell$ shows how the value would change if we were to restore the neuron. Therefore, we choose the neuron with the largest value and restore it.

While the difference-guided refinement only considers the *local behavior* of a neuron, the gradient-guided refinement also takes the influence of a neuron on the network's output into account.

Look-ahead Refinement The look-ahead refinement is the most greedy approach for refinement, where we simulate the restoration of each replaced neuron and observe how it changes the difference between the output $\tilde{\mathbf{y}}$ of the abstraction and $\mathbf{y}$ of the original network. We then choose to restore the neuron that minimizes this difference. Again, we can quantify the difference with an appropriate loss function, as we have done in the gradient-guided refinement.

E Supplemental Experiments on Finding a Basis

In this section, we provide more experiments on how to find a basis, similar to Fig. 2. We can see the results in Figs. 11 and 12. The x-axis shows the reduction rate. On the left (blue), the y-axis of the plots shows the accuracy of the network on the test data set, which was not used for generating the abstraction, and on the right (red) the time for the reduction, where the time is measured in seconds. The title of each plot indicates the architecture. On MNIST, Fig. 11, we can see that the greedy approach outperforms the variance-based approach in terms of accuracy, but it performs much worse in terms of runtime. The latter can also be seen on CIFAR-10. However, the accuracies of the greedy and the variance-based approach are much more similar on this benchmark.

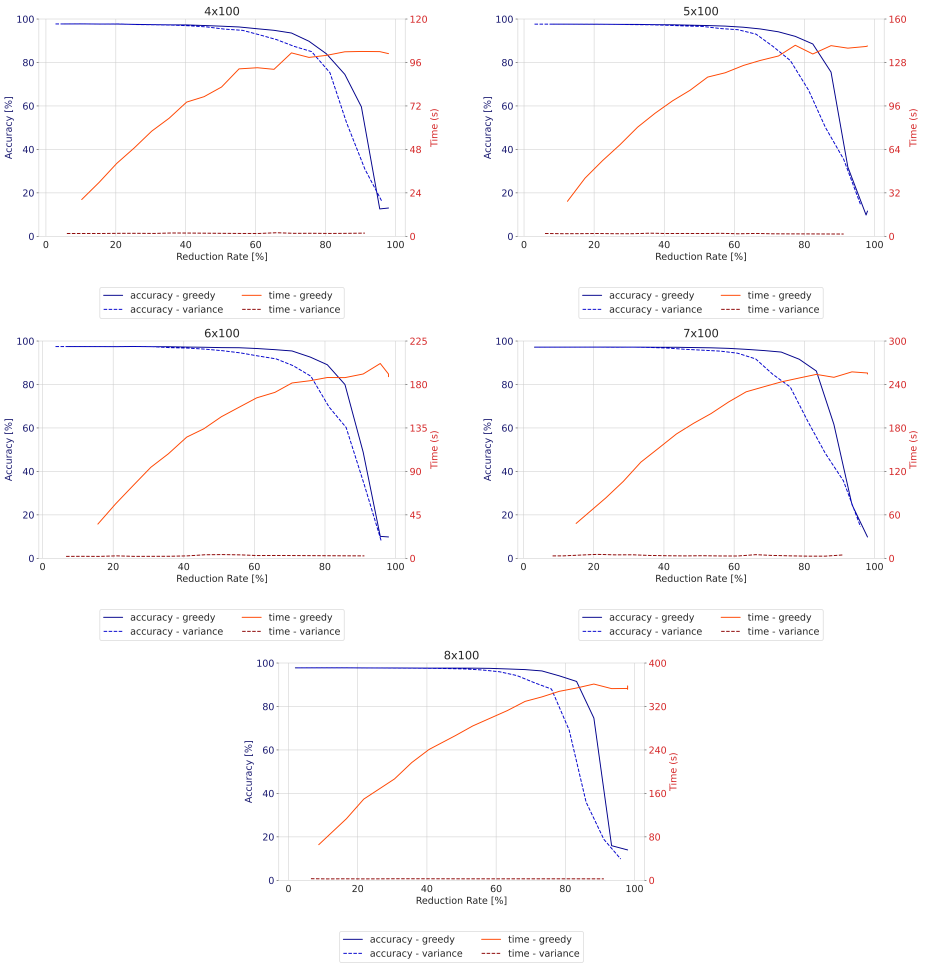


Fig. 11: *Comparison of how to find a basis on MNIST* - These plots show the accuracy and runtime for the greedy and variance-based LiNNA with orthogonal projection.

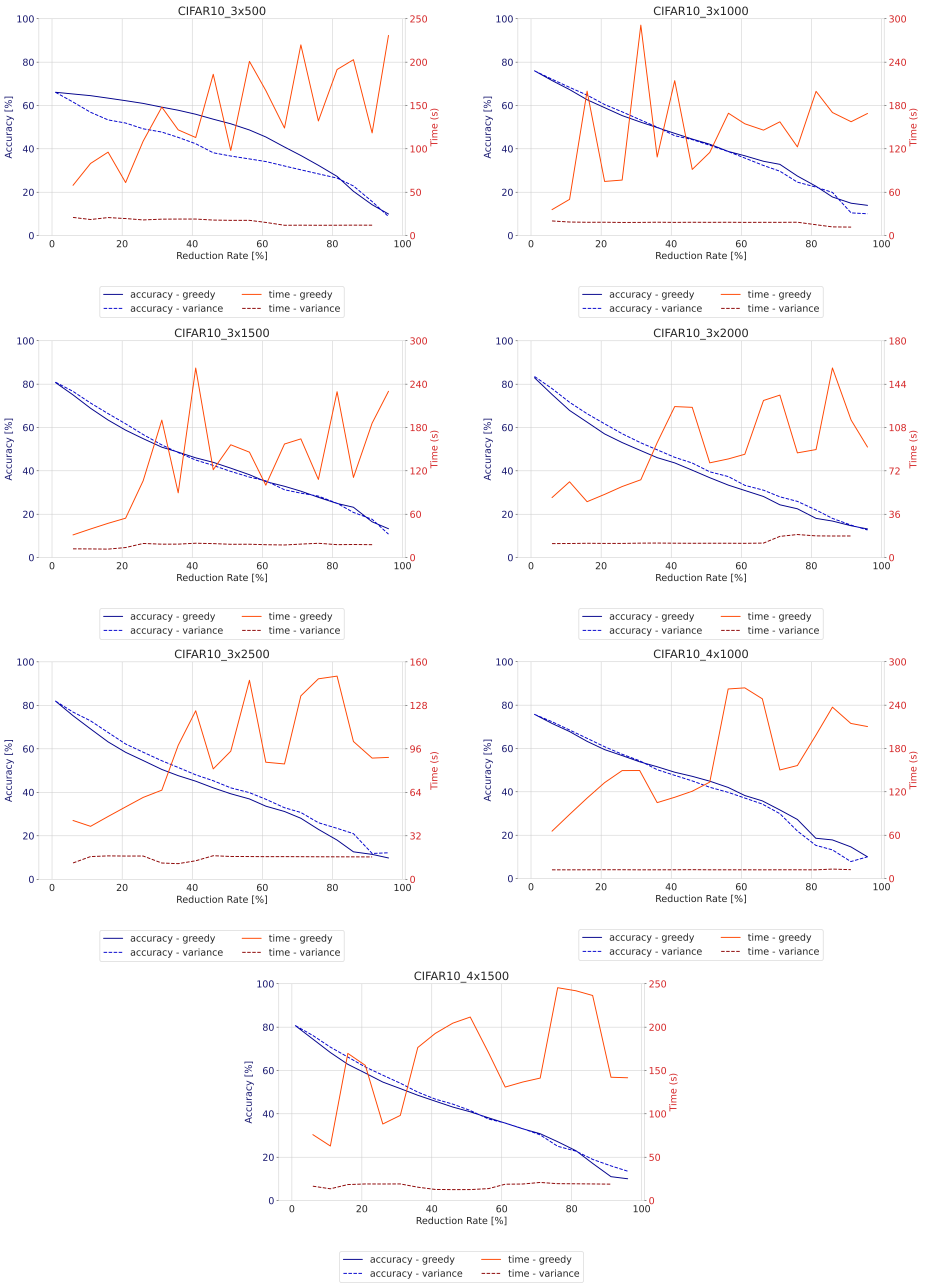


Fig. 12: *Comparison of how to find a basis on CIFAR-10* - These plots show the accuracy and runtime for the greedy and variance-based LiNNA with orthogonal projection.

F Supplemental Details on the Implementation of the Bisimulation

The bisimulation chooses a random neuron as representative of a group, where all neurons of the group agree on their incoming weights up to a deviation of δ. We start by calculating the distance of two neurons i, j as follows: $d(i,j) = \max_k\{|w^{(\ell-1)}(i,k) - w^{(\ell-1)}(j,k)|, |b_i^{(\ell)} - b_j^{(\ell)}|\}$. Note that the neurons i, j are δ-bisimilar if and only if $d(i,j) \leq \delta$. Afterward, we use agglomerative clustering [32] to find groups of neurons that have δ-similar incoming weights. Since agglomerative clustering belongs to the methods of hierarchical clustering, it is somewhat similar to the minimization approach as described in [19] for the exact bisimulation without δ-deviation, and we hope that it captures the same intended behavior.

G Supplemental Experiments on Comparing Our Approaches

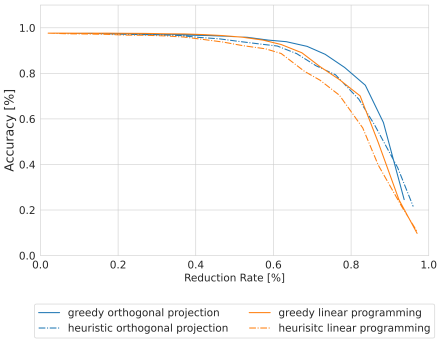


Fig. 13: *Comparison of Our Approaches* - On MNIST 3x100. This plot contains the comparison of all four possible approaches: orthogonal projection (blue) and linear programming (orange), each either greedy (solid) or based on a heuristic (dashed).

In Fig. 13, we have four plots in total. Each two for linear programming and orthogonal projection, and the greedy and the heuristic-based approach. The results for the linear programming are shown in green, and the results for the orthogonal projection are in blue. The greedy approaches are shown with a solid line and the heuristic-based approaches with a dashed line. As already seen, the greedy approach outperforms the heuristic-based approach. This also holds for linear programming. However, we can also see, that the greedy orthogonal projection always outperforms all other approaches, except for reduction rates close to 90%. Additionally, and more importantly, the heuristic-based orthogonal projection is almost as good as the greedy linear programming; for high reduction rates, it is even better. Fig. 14 shows on the left a plot for the orthogonal projection. It shows the accuracy (blue) and the time (red) of the greedy (solid) against the variance-based (dashed) method. On the right, we have the same plot

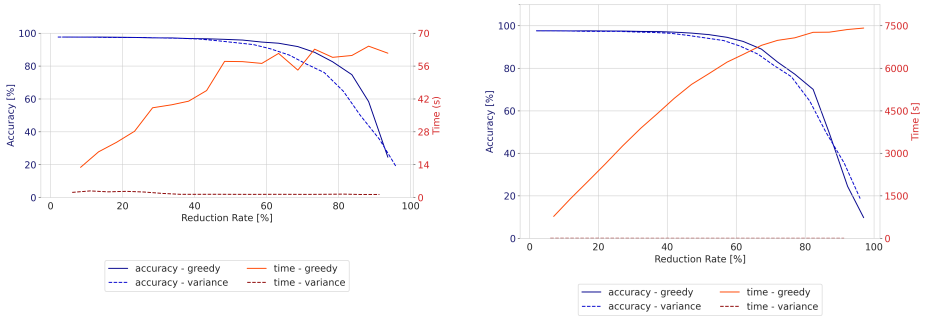


Fig. 14: *Comparison of Linear Programming to Orthogonal Projection* - Plots for the orthogonal projection (left) and linear programming (right) on MNIST 3x100. Each is either greedy (solid) or based on a heuristic (dashed). The accuracy is shown in blue (left y-axis), and the computation time is in red (right y-axis).

for the linear programming. Note here that the time for the linear programming is 100 times as much as for the orthogonal projection.

H Supplemental Experiments on Comparison to Related Works

We have some more experiments in this section on the comparison of LiNNA to related works, i.e. DeepAbstract and bisimulations. We performed the experiments from Fig. 3 on some more architectures of the MNIST. We have also conducted

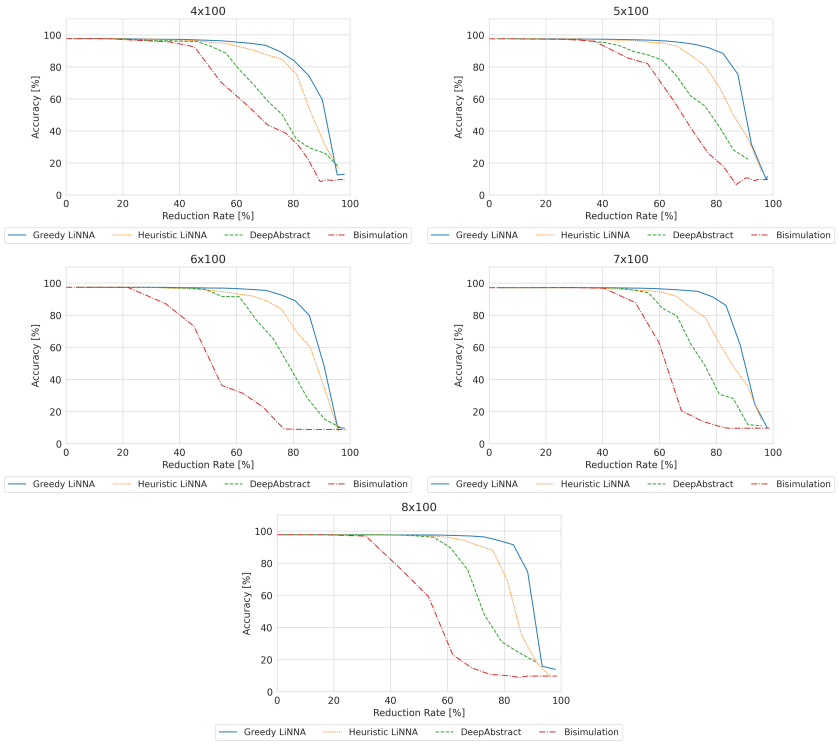


Fig. 15: *Comparison on MNIST* - There are plots for different architectures. Each plot contains four graphs: Greedy LiNNA, Heuristic LiNNA, DeepAbstract [2], and the bisimulation [19].

some experiments on another dataset: FashionMNIST. In contrast to MNIST, which contains digits from zero to 10, FashionMNIST contains images of different clothes. Note that the results are very similar to the ones that we generated on MNIST.

In Fig. 16, we show the comparison of LiNNA, greedy and variance-based, in comparison with DeepAbstract and the bisimulation on a FashionMNIST 3x100 network in terms of accuracy. We can see that it looks very similar to the MNIST plots: LiNNA greedy outperforms all other approaches, closely followed by LiNNA with heuristic. DeepAbstract performs better than the bisimulation, which shows a rapid decrease in the accuracy already at 40% reduction rate, whereas LiNNA can keep the accuracy stable up until 60%.

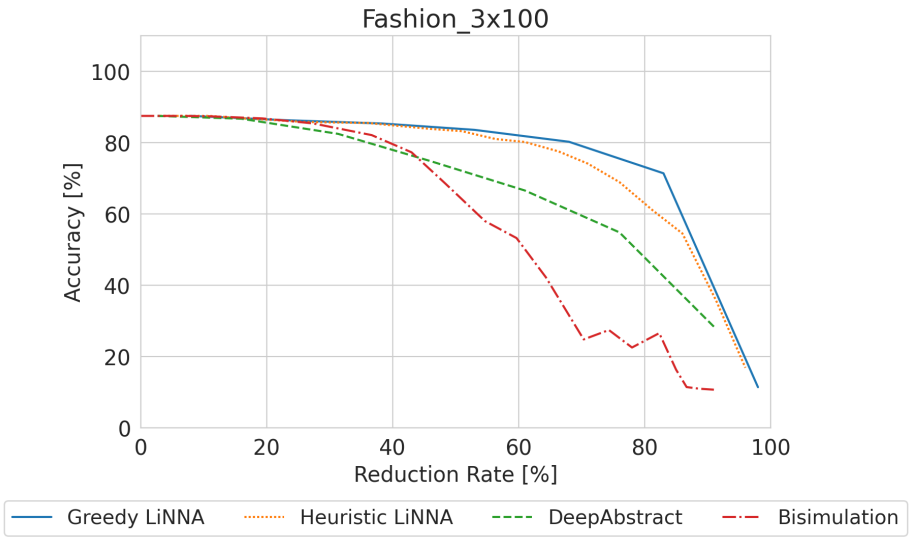


Fig. 16: Comparison of all related work on FashionMNIST

Since the greedy approach of LiNNA and DeepAbstract can take quite long, we performed only a comparison of the bisimulation and the heuristic based LiNNA on more networks, see Fig. 17. The plots differ slightly, but their overall message is the same: LiNNA performs better in terms of accuracy. Additionally, we can see that it is easier to reduce networks that were bigger to begin with. Take, for example, the 6x50 and the 6x200 network in comparison. LiNNA can reduce up to 50% on the first without a relevant decrease in the accuracy, but up to 70% on the bigger network. This can be explained by the fact that bigger networks can contain more redundant information that our approach can detect and remove.

In Fig. 18, we provide a comparison of the bisimulation and LiNNA based on the variance heuristic on some more networks that were trained on CIFAR-10. Some values for the bisimulation were not possible to provide, because it was difficult to find suitable δ-values for the bisimulation. Nevertheless, we can still see that LiNNA performs much better than the bisimulation. Its abstractions have always a higher accuracy as the ones resulting from the bisimulation.

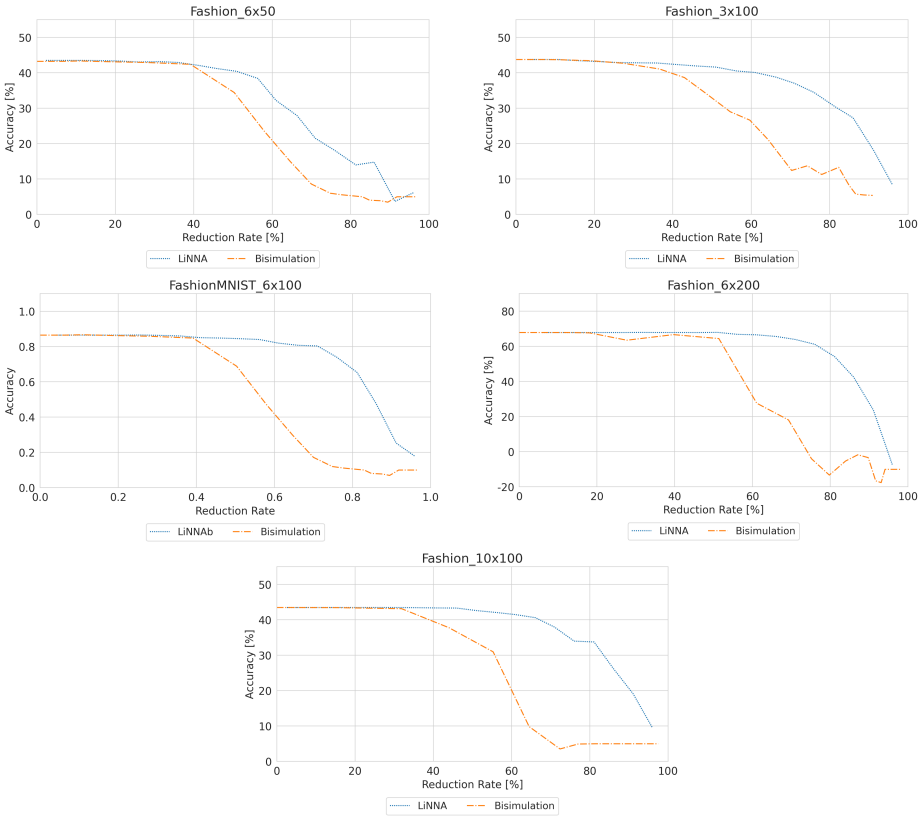


Fig. 17: Comparison of bisimulation and heuristic LiNNA on FashionMNIST

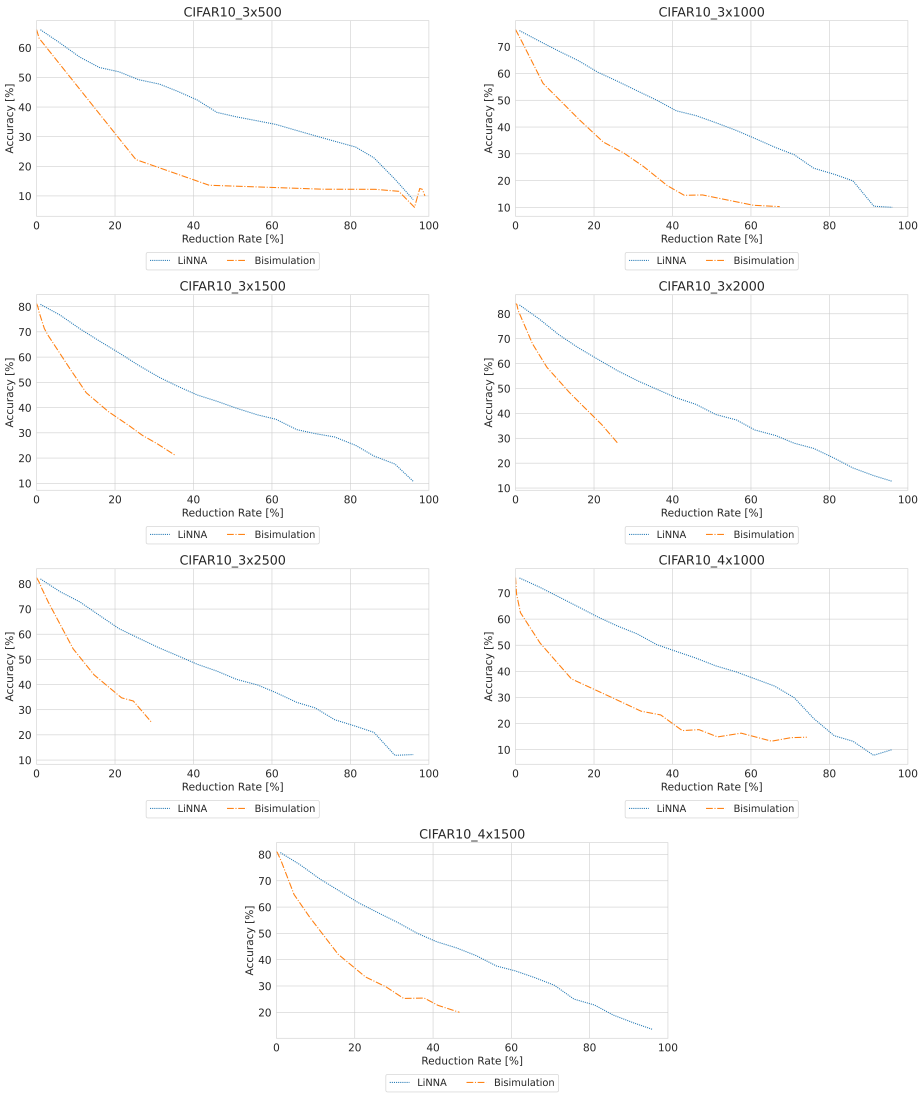


Fig. 18: Comparison of bisimulation and heuristic LiNNA on CIFAR-10

I Supplemental Experiments on Syntactic VS Semantic

This section contains more plots on the comparison of the syntactic- and semantic-based abstraction, similar to Fig. 6.

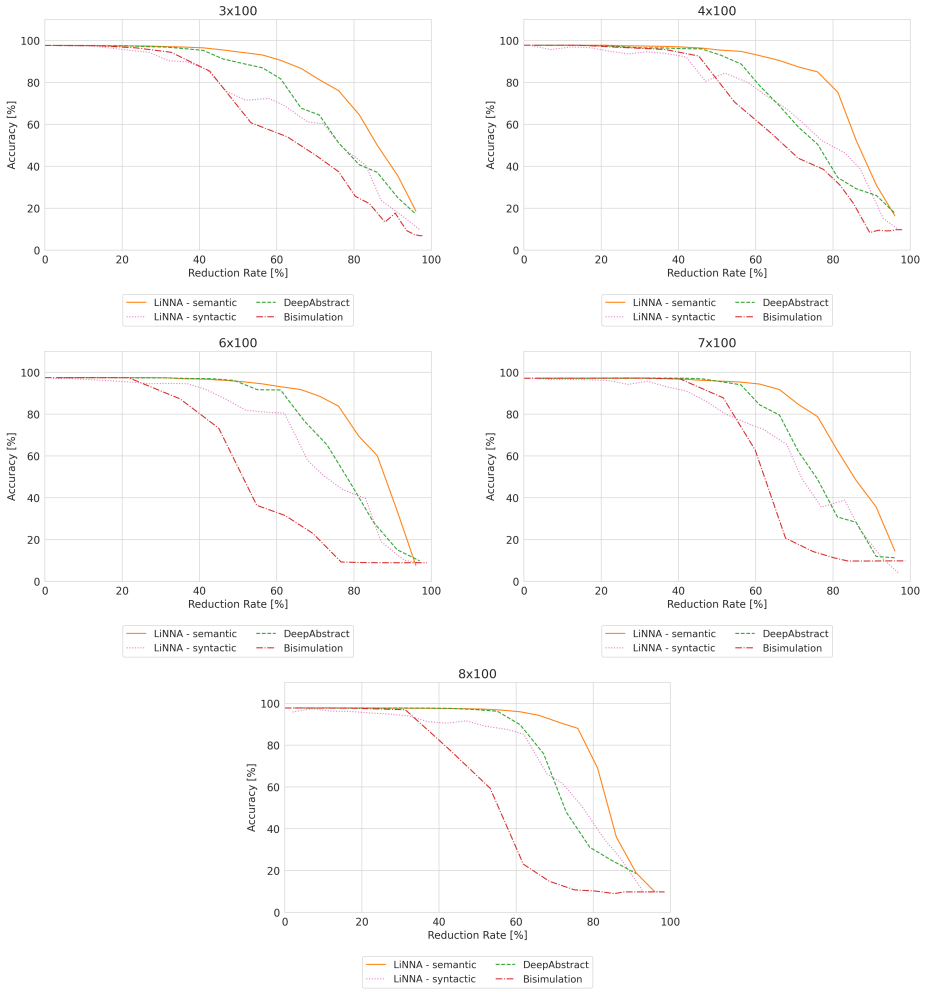


Fig. 19: *Syntactic VS. Semantic* - These plots show the difference of using semantic resp. syntactic information for the abstraction on different networks, trained on MNIST

J Time Comparison of the Different Approaches

This section contains tables with the runtimes of the approaches. Each neural network was abstracted with increasing reduction rates. In the tables, we report the mean, median, minimum and maximum runtimes for the different approaches over all reduction rates. This should offer the possibility to have a more detailed insight into the computation times.

Table 1: MNIST3x100 - Comparison of computation times [s]

type	median	mean	min	max
LiNNA greedy (OP)	55.70	47.03	12.87	64.47
LiNNA heuristic (OP)	1.48	1.47	1.32	1.53
LiNNA greedy (LP)	5807.21	5129.88	774.40	7420.29
LiNNA heuristic (LP)	17.50	16.49	2.42	27.90
Bisimulation	1.07	1.10	1.00	1.44
DeepAbstract	187.03	183.30	147.54	208.97

Table 2: MNIST4x100 - Comparison of computation times [s]

type	median	mean	min	max
LiNNA greedy (OP)	92.21	77.93	20.30	102.08
LiNNA heuristic (OP)	1.81	1.84	1.70	2.05
Bisimulation	1.25	1.27	1.12	1.43
DeepAbstract	335.73	329.88	223.52	367.45

Table 3: MNIST5x100 - Comparison of computation times [s]

type	median	mean	min	max
LiNNA greedy (OP)	120.41	106.81	25.76	140.67
LiNNA heuristic (OP)	2.21	2.20	1.97	2.35
Bisimulation	1.38	1.42	1.23	1.73
DeepAbstract	1142.42	1269.52	541.95	2205.43

Table 4: MNIST6x100 - Comparison of computation times [s]

type	median	mean	min	max
LiNNA greedy (OP)	166.15	146.38	35.42	201.71
LiNNA heuristic (OP)	2.55	2.51	2.24	2.66
Bisimulation	1.61	1.66	1.41	2.18
DeepAbstract	756.35	695.91	64.04	832.05

Table 5: MNIST7x100 - Comparison of computation times [s]

type	median	mean	min	max
LiNNA greedy (OP)	229.85	199.01	48.14	257.41
LiNNA heuristic (OP)	3.01	3.00	2.60	3.54
Bisimulation	1.84	1.86	1.60	2.17
DeepAbstract	2418.94	2181.16	151.61	2626.65

Table 6: MNIST4x50 - Comparison of computation times [s]

type	median	mean	min	max
Bisimulation	0.86	0.87	0.82	0.93
LiNNA heuristic (OP)	1.44	1.47	1.33	2.00

Table 7: MNIST4x150 - Comparison of computation times [s]

type	median	mean	min	max
Bisimulation	2.02	2.04	1.74	2.37
LiNNA heuristic (OP)	1.74	1.77	1.52	2.31

Table 8: MNIST4x200 - Comparison of computation times [s]

type	median	mean	min	max
Bisimulation	2.53	2.48	2.10	2.65
LiNNA heuristic (OP)	1.95	2.05	1.79	2.61

Table 9: MNIST4x250 - Comparison of computation times [s]

type	median	mean	min	max
Bisimulation	3.12	3.04	2.56	3.28
LiNNA heuristic (OP)	2.18	2.09	1.77	2.52

Table 10: MNIST4x300 - Comparison of computation times [s]

type	median	mean	min	max
Bisimulation	4.08	4.05	3.48	4.72
LiNNA heuristic (OP)	2.07	2.09	1.70	2.38

Table 11: MNIST4x350 - Comparison of computation times [s]

type	median	mean	min	max
Bisimulation	5.26	5.21	4.52	5.62
LiNNA heuristic (OP)	2.56	2.59	1.90	3.22

Table 12: MNIST4x400 - Comparison of computation times [s]

type	median	mean	min	max
Bisimulation	5.84	5.91	5.31	6.54
LiNNA heuristic (OP)	2.55	2.46	1.62	2.89

Table 13: MNIST4x450 - Comparison of computation times [s]

type	median	mean	min	max
Bisimulation	7.38	7.30	6.51	8.11
LiNNA heuristic (OP)	2.79	2.66	1.72	2.90

Table 14: MNIST4x500 - Comparison of computation times [s]

type	median	mean	min	max
Bisimulation	8.29	8.52	7.39	9.39
LiNNA heuristic (OP)	3.10	2.95	1.81	3.28

K Details on Refinement

This section contains some more supplementary material for the refinement. We have a table to give a more detailed insight into the computation time. Note, however, that there appeared to be a malfunction in the difference-based approach on the 3x100 network. The high maximum number most likely occurred due to some issues with the machine it was run on.

Table 15: Refinement - Comparison of computation times [s]

NN	method	mean	median	min	max
3x100	Difference	0.77	0.02	0.02	7901.53
	Gradient	0.45	0.05	0.04	11.27
	Lookahead	5.19	1.11	0.16	8724.71
4x100	Difference	0.20	0.02	0.02	8.13
	Gradient	0.56	0.07	0.02	13.45
	Lookahead	6.85	1.53	0.22	10719.30
5x100	Difference	0.21	0.03	0.02	9.87
	Gradient	1.93	0.07	0.02	11499.10
	Lookahead	1.32	1.94	0.30	9.31
6x100	Difference	0.24	0.03	0.03	7.14
	Gradient	2.91	0.08	0.02	16172.10
	Lookahead	1.56	2.37	0.34	5.45

Additionally, we have the plot from Fig. 9 with two more networks to show more on the evolution of the refinement approaches.

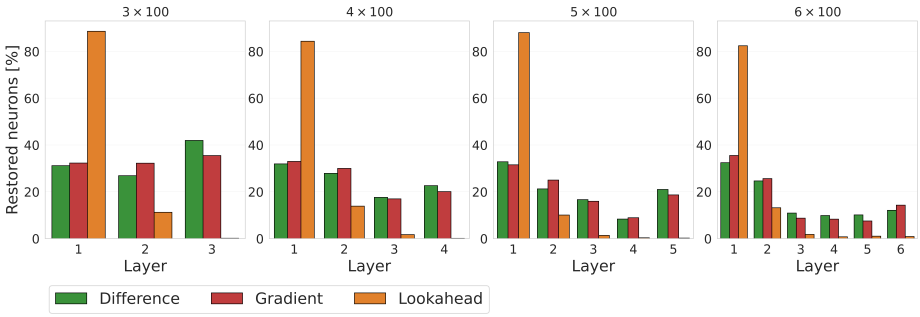


Fig. 20: We considered abstractions that were obtained with a 50% reduction rate and fixed 1000 counterexamples. The plots depict the percentage of restored neurons in the layers of the different MNIST networks.

L Experiments on the Error

In addition to the plots that we have already seen in Fig. 10, we have a plot the shows the histogram over all errors of all replaced neurons in a layer. The values are usually very close to 0.

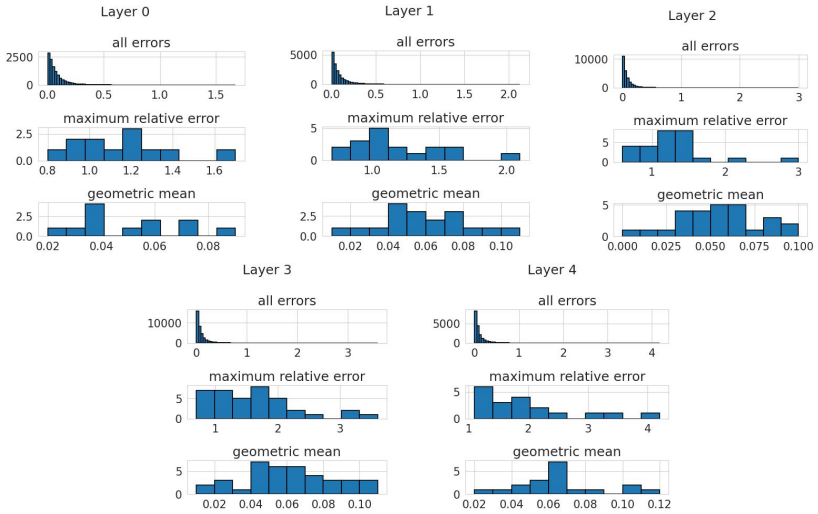


Fig. 21: Histograms of the relative error of a MNIST5x100 network that was reduced by 30%. The first row depicts a histogram over all relative errors for all replaced neurons on 1000 images of the test set. The second row shows the maximum relative error of each neuron that occurred for some input from the test set. The las row plots the geometric mean of the relative error of each neuron over 100 images of the test set.

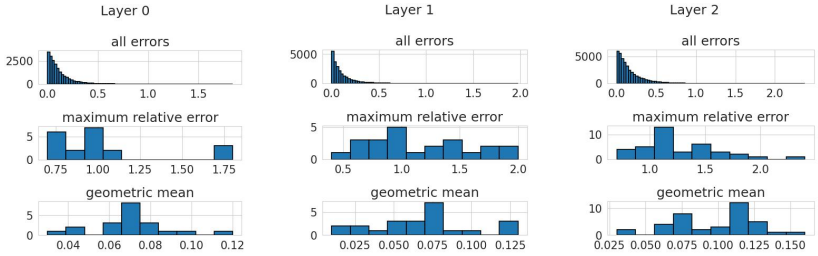
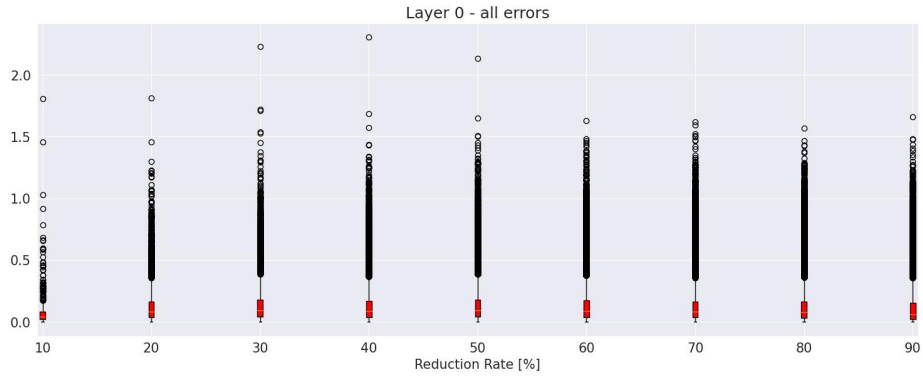


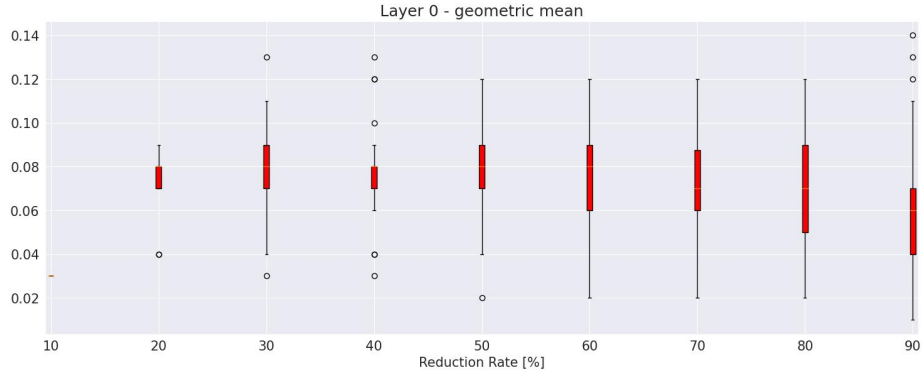
Fig. 22: Histograms of the relative error of a MNIST3x100 network that was reduced by 30%. The first row depicts a histogram over all relative errors for all replaced neurons on 1000 images of the test set. The second row shows the maximum relative error of each neuron that occurred for some input from the test set. The las row plots the geometric mean of the relative error of each neuron over 100 images of the test set.

Additionally, we show how the error evolves when reducing the network more. To this end, we have in Fig. 23, boxplots for a) all appearing relative errors, b) the maximum relative error, c) the geometric mean of the relative error on a MNIST network with 3x100 neurons for different reduction rates. We can see that the error looks most stable in the first layer and least stable in the last layer. However, even for a reduction of 90%, the geometric mean of the relative error is still below 0.3 for all cases but one. This could indicate that the number of cases where the abstraction fails increase only slightly. The maximum relative error

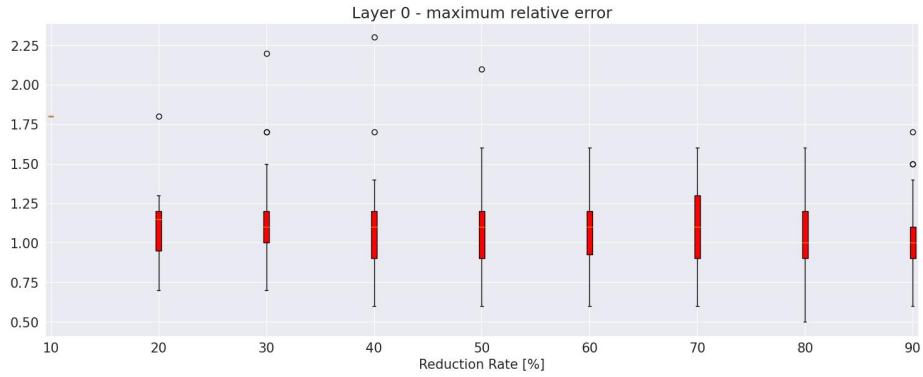
seems to increase steadily, which could mean that whenever the abstraction fails, it fails even more.



(a)

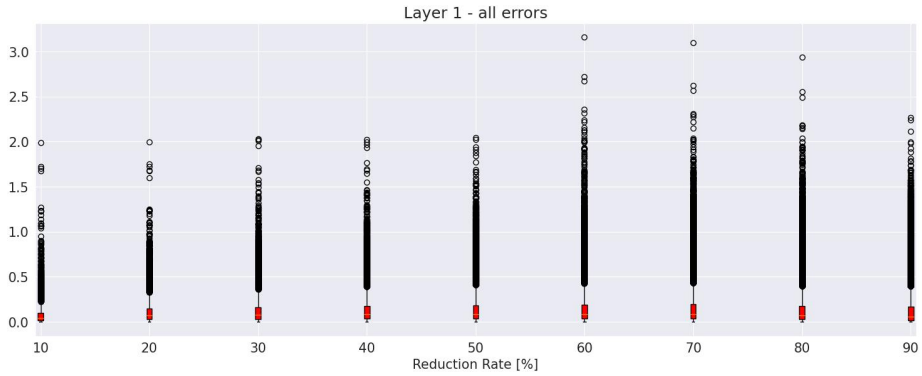


(b)

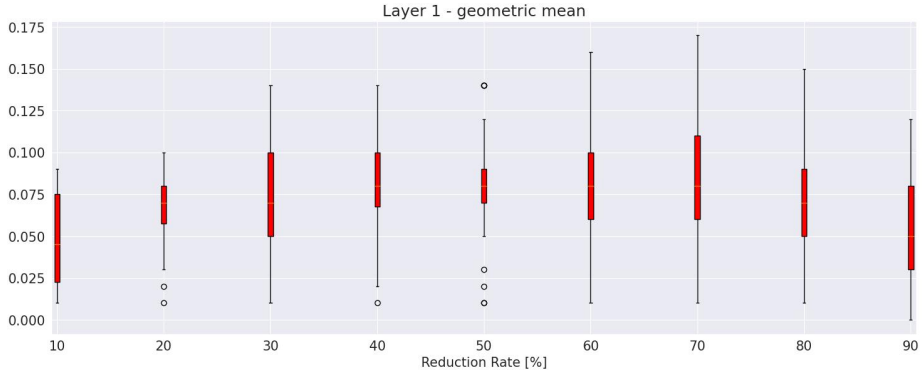


(c)

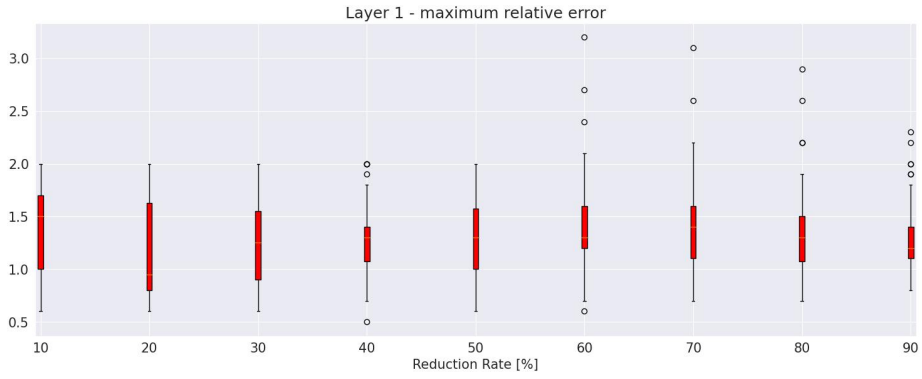
Fig. 23: Evolution of the relative error for different reduction rates. The network is MNIST3x100 and the zeroth layer. We see for each reduction rate in [10-90] a boxplot for a) all errors, b) the geometric mean, c) the maximum error.



(a)

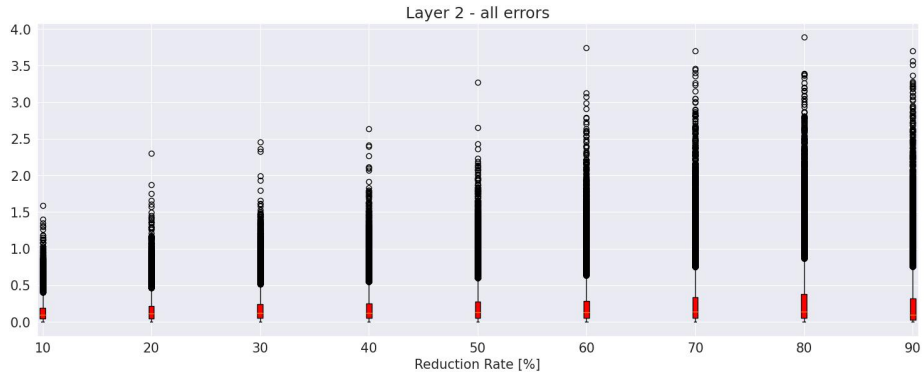


(b)

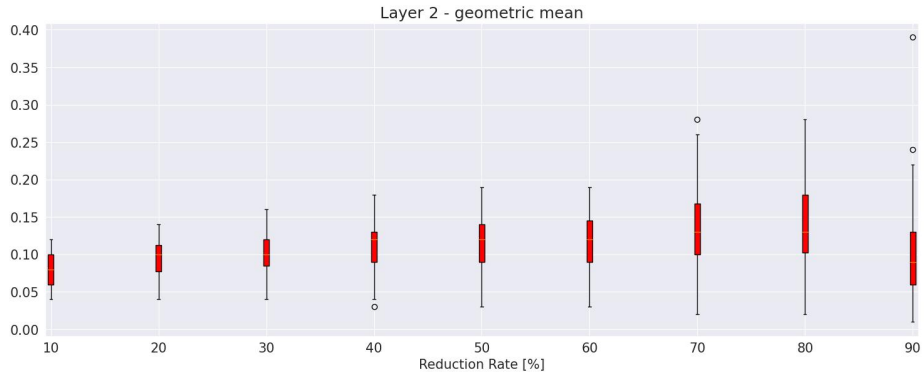


(c)

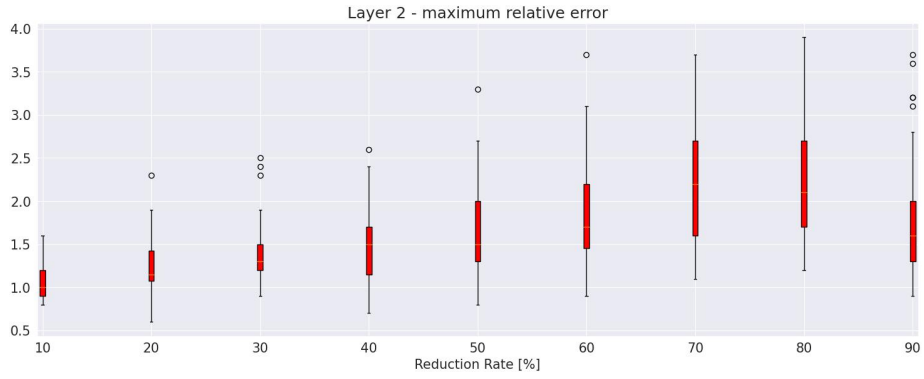
Fig. 24: Evolution of the relative error for different reduction rates. The network is MNIST3x100 and the first layer. We see for each reduction rate in [10-90] a boxplot for a) all errors, b) the geometric mean, c) the maximum error.



(a)



(b)



(c)

Fig. 25: Evolution of the relative error for different reduction rates. The network is MNIST3x100 and the second layer. We see for each reduction rate in [10-90] a boxplot for a) all errors, b) the geometric mean, c) the maximum error.