

Mini-Batch Robustness Verification of Deep Neural Networks

SAAR TZOUR-SHADAY, Technion, Israel

DANA DRACHSLER-COHEN, Technion, Israel

Neural network image classifiers are ubiquitous in many safety-critical applications. However, they are susceptible to adversarial attacks. To understand their robustness to attacks, many local robustness verifiers have been proposed to analyze ϵ -balls of inputs. Yet, existing verifiers introduce a long analysis time or lose too much precision, making them less effective for a large set of inputs. In this work, we propose a new approach to local robustness: *group local robustness* verification. The key idea is to leverage the similarity of the network computations of certain ϵ -balls to reduce the overall analysis time. We propose BaVerLy, a sound and complete verifier that boosts the local robustness verification of a set of ϵ -balls by dynamically constructing and verifying mini-batches. BaVerLy adaptively identifies successful mini-batch sizes, accordingly constructs mini-batches of ϵ -balls that have similar network computations, and verifies them jointly. If a mini-batch is verified, all its ϵ -balls are proven robust. Otherwise, one ϵ -ball is suspected as not being robust, guiding the refinement. BaVerLy leverages the analysis results to expedite the analysis of that ϵ -ball as well as the analysis of the mini-batch with the other ϵ -balls. We evaluate BaVerLy on fully connected and convolutional networks for MNIST and CIFAR-10. Results show that BaVerLy scales the common one by one verification by 2.3x on average and up to 4.1x, in which case it reduces the total analysis time from 24 hours to 6 hours.

CCS Concepts: • **Theory of computation** → **Program analysis**; **Program verification**; • **Software and its engineering** → **Formal methods**; • **Computing methodologies** → **Neural networks**.

Additional Key Words and Phrases: Neural Network Verification, L_∞ local robustness

ACM Reference Format:

Saar Tzour-Shaday and Dana Drachslers-Cohen. 2025. Mini-Batch Robustness Verification of Deep Neural Networks. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 372 (October 2025), 30 pages. <https://doi.org/10.1145/3763150>

1 Introduction

Neural networks are successful in many applications, including object detection, speech recognition, text generation and machine translation [Bahdanau et al. 2015; Graves and Jaitly 2014; Redmon et al. 2016; Sutskever et al. 2014]. In particular, they are widely ubiquitous as image classifiers [Krizhevsky et al. 2012], playing a crucial role in safety-critical applications, such as autonomous cars [Ayachi et al. 2020; Bachute and Subhedar 2021; Bojarski et al. 2016], medical diagnosis [Anthemopoulos et al. 2016; Esteva et al. 2017], and surveillance systems [Javed and Shah 2002; Zahrawi and Shaalan 2023]. Guaranteeing the safety of these networks is imperative in these settings, especially in light of the recent European Regulations on Artificial Intelligence [European Commission 2020].

However, neural networks are known to be vulnerable to different kinds of attacks. One of the attacks that has drawn a lot of attention in recent years is adversarial example attacks [Croce and Hein 2019; Goodfellow et al. 2015; Guo et al. 2019; Ilyas et al. 2018; Karim et al. 2021; Szegedy et al. 2013; Yuan et al. 2019]. An adversarial attack that targets an image classifier computes a small noise – typically imperceptible to the human eye – that leads the network to misclassify. To

Authors' Contact Information: Saar Tzour-Shaday, Technion, Haifa, Israel, saartz@campus.technion.ac.il; Dana Drachslers-Cohen, Technion, Haifa, Israel, ddana@ee.technion.ac.il.



Please use nonacm option or ACM Engage class to enable CC licenses

This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2025 Copyright held by the owner/author(s).

ACM 2475-1421/2025/10-ART372

<https://doi.org/10.1145/3763150>

show the robustness of a neural network to these attacks, many robustness verifiers have been introduced [Ferrari et al. 2022; Gehr et al. 2018; Katz et al. 2017, 2019; Lopez et al. 2023; Mohapatra et al. 2020; Singh et al. 2018, 2019a; Tjeng et al. 2019; Wang et al. 2021]. Most of them focus on proving the *local robustness* of a given L_∞ ϵ -ball [Singh et al. 2018, 2019a; Tjeng et al. 2019; Wang et al. 2021], though some of them focus on other kinds of perturbations, such as other L_p ϵ -balls [Huang et al. 2021; Shapira et al. 2023, 2024], geometric perturbations [Balunovic et al. 2019; Wang et al. 2023], or global robustness [Kabaha and Drachler-Cohen 2024; Leino et al. 2021].

Despite the immense research on verifiers for determining the local robustness in a single ϵ -ball, they still face challenges in providing formal guarantees to deep networks: complete verifiers struggle to scale because of their exponential time complexity, while incomplete verifiers struggle to successfully verify robustness of deep networks because of their precision loss. Additionally, typically network designers are not interested in the local robustness of a single ϵ -ball. Ideally, they aim at understanding the local robustness in all “relevant” ϵ -balls. Since the set of relevant ϵ -balls does not have a formal characterization, it is often estimated as the set of ϵ -balls around inputs in a given test set. Although these sets often contain similar inputs, most local robustness verifiers do not leverage this setting and verify ϵ -balls one by one. An exception is works on shared certificates [Fischer et al. 2022; Ugare et al. 2022], which learn verification templates with the goal of expediting the analysis of unseen ϵ -balls. However, they are not designed to directly leverage the given test set to reduce the overall analysis time. Additionally, the template generation takes several hours and existing shared certification techniques focus on incomplete verification, and consequently they may not expose the true robustness level of a network.

In this work, we consider the problem of *group local robustness verification*. Given a network, a set of inputs, and a real number ϵ , the goal is to determine for every input’s ϵ -ball whether it is robust or not while minimizing the overall analysis time. We focus on complete verification, because it enables to understand the robustness level of the network. In particular, it provides a faithful approach to compare the robustness levels of two networks. This problem is challenging since it requires to identify which ϵ -balls can be analyzed together without leading to spurious adversarial examples and without increasing the verification’s complexity. The latter may happen since the verification’s complexity is exponential in the number of non-stable neurons (i.e., neurons for which the activation function exhibits nonlinearity). Generally, unifying ϵ -balls may lead to increasing the number of non-stable neurons. In particular, unwise unification may lead to a significantly higher number of non-stable neurons, thus increasing the verification’s complexity and making the overall analysis time longer than analyzing the ϵ -balls one by one.

To balance between verifying multiple ϵ -balls and avoiding increased complexity as well as precision loss, we propose to verify *mini-batches*. A mini-batch is a small subset of inputs for which the network performs similar computations. This concept is inspired by common machine learning training algorithms, which process data in mini-batches to significantly enhance computational efficiency (though their mini-batches need not consist of inputs with similar network computations). Verifying a mini-batch can be encoded by a mixed-integer linear program (MILP), extending the encoding of a previous local robustness verifier for a single ϵ -ball [Tjeng et al. 2019]. However, the naive extension suffers from higher verification’s complexity as well as precision loss. To cope, we propose several ideas. First, we begin the joint verification of a mini-batch in an intermediate layer of the network (like the generated templates of Fischer et al. [2022]; Ugare et al. [2022]). Unifying in an intermediate layer enables to focus on the computations where the ϵ -balls are *perceived similar*, thereby the verification’s complexity does not grow significantly and the overapproximation error is low. Second, we encode the mini-batch verification such that the MILP solver either determines that the mini-batch is fully verified, or detects an ϵ -ball which may be not robust. This encoding enables a simple refinement: this ϵ -ball is analyzed separately and the other ϵ -balls continue their

joint analysis. Thus, the time spent on the analysis of a mini-batch is not wasted. Further, after separating the possibly non-robust ϵ -ball from the mini-batch, its analysis and the analysis of the remaining batch leverage the analysis results of the previous mini-batch to terminate faster. Third, we estimate the similarity of ϵ -balls by the activation patterns of their center input. This approach is both fast and, in practice, estimates well closeness of ϵ -balls. Fourth, we learn the optimal mini-batch size throughout the analysis. In particular, it may start with larger mini-batches, consisting of the most similar ϵ -balls and reduce the mini-batch sizes, when the remaining ϵ -balls are further apart. This step relies on multi-armed bandit with the Thompson Sampling.

We implemented our approach in a system called BaVerLy (a **batch verifier** for **local** robustness). We evaluate BaVerLy on fully connected networks and convolutional networks for MNIST and CIFAR-10. BaVerLy boosts the verification time by 2.3x on average and up to 4.1x compared to one by one verification. In particular, it reduces the analysis time from 13 hours to 5 hours, on average. We further show that learning the optimal mini-batch size boosts BaVerLy by 2.5x.

2 Problem Definition

In this section, we define our problem: group robustness verification. We begin with background on image classifiers and local robustness. We then define our problem and discuss existing approaches.

Image classifiers. Image classifiers take an input image x and determine which class from a set of classes \mathcal{C} describes the object shown in the image. For example, a CIFAR-10 classifier maps images to one of ten classes, e.g., a ship or a deer. An image classifier implemented by a deep neural network (DNN) is a function $N : [0, 1]^{d_{in}} \rightarrow \mathbb{R}^{d_{out}}$ composed from L hidden layers $N = N_L \circ \dots \circ N_1$. The input to the first hidden layer, referred to as the input layer, is denoted by $z_0 = x \in [0, 1]^{d_{in}}$, while the output of the last hidden layer, known as the output layer, is denoted by $z_L \in \mathbb{R}^{d_{out}}$. Each hidden layer N_i takes as input the output vector of the previous layer z_{i-1} and returns a vector z_i . To compute the output vector, it first executes an affine transformation $z'_i = W_i z_{i-1} + b_i$, where W_i and b_i are the layer's weight matrix and bias vector, respectively. This transformation is then followed by a nonlinear activation function. We focus on piecewise-linear networks, whose predominant activation function is the Rectified Linear Unit (ReLU). The ReLU function, computing $z_i = \text{ReLU}(z'_i)$, is invoked component-wise and returns the maximum of each component and zero: $\forall j. (z_i)_j = \text{ReLU}((z'_i)_j) = \max((z'_i)_j, 0)$. The output of the last layer N_L contains d_{out} neurons, each returns the score of a unique class $c \in \mathcal{C}$ (where $|\mathcal{C}| = d_{out}$). The process of passing an input $x \in [0, 1]^{d_{in}}$ through the DNN to receiving the output $N(x) \in \mathbb{R}^{d_{out}}$ is called a feed-forward pass. At the end of this process, the classification for x is the class with the highest score: $c' = \arg \max(N(x))$.

Local robustness. To prove safety to adversarial attacks, many works focus on analyzing the *local robustness* of a network classifier [Gehr et al. 2018; Katz et al. 2017; Lopez et al. 2023; Müller et al. 2021; Singh et al. 2019a; Tjeng et al. 2019; Tran et al. 2020; Wu et al. 2024; Zhou et al. 2024]. The vast majority of works focuses on proving robustness in the L_∞ ϵ -ball of a given input. Formally, given an input image $x \in \mathbb{R}^{d_{in}}$ and an $\epsilon \in \mathbb{R}^+$, the L_∞ ϵ -ball of x is the set of all inputs that differ from x by at most ϵ , that is $B_\epsilon^\infty(x) = \{x' \mid \|x - x'\|_\infty = \max(|x_1 - x'_1|, \dots, |x_{d_{in}} - x'_{d_{in}}|) \leq \epsilon\}$. A network classifier N is locally robust in $B_\epsilon^\infty(x)$ if it classifies all its inputs the same: $\forall x' \in B_\epsilon^\infty(x), \arg \max(N(x)) = \arg \max(N(x'))$. In the following, we say that N is robust in the ϵ -ball (or the neighborhood) of x if N is locally robust in $B_\epsilon^\infty(x)$. Local robustness has been shown to be NP-hard [Katz et al. 2017], which stems from the nonlinear activation function's computations (e.g., the ReLUs). Thus, existing local robustness verifiers balance between their precision and scalability. This gave rise to two approaches: complete and incomplete verifiers. Complete verifiers guarantee to determine whether an ϵ -ball is robust but suffer from a long runtime, which increases as the network is deeper (i.e., has

more layers). In contrast, incomplete verifiers favor scalability and overapproximate the activation computations to expedite the analysis at the expense of precision loss, i.e., the verifier may fail to prove robustness for some robust ϵ -balls. Commonly, the deeper the network or the larger the ϵ , the higher the precision loss, and thus the higher failure rate of incomplete verifiers.

Group local robustness. While many complete verifiers propose ways to scale their analysis, they still struggle to scale. In this work, we aim to leverage the practical scenario of local robustness: verifying local robustness of a set of ϵ -balls. While, ideally, a network designer wishes to understand the local robustness in every input's ϵ -ball (called global robustness), this is much more challenging and existing global robustness verifiers do not scale to the size of networks that local robustness verifiers scale. Instead, it is common to “estimate” the global robustness of the network by evaluating its local robustness in the ϵ -balls of a set of inputs. While there is no guarantee that the network is locally robust in unseen ϵ -balls, this approach helps designers compare the robustness of networks to adversarial attacks. We next formally define this problem and discuss existing approaches.

Definition 2.1 (Group Local Robustness Verification). Given a set of inputs $S \subseteq [0, 1]^{d_{in}}$, a classifier $N : [0, 1]^{d_{in}} \rightarrow \mathbb{R}^{d_{out}}$, and $\epsilon \in \mathbb{R}^+$, *group local robustness verification* determines for every input $x \in S$ whether N is locally robust in its ϵ -ball $B_\epsilon^{\infty}(x)$ while minimizing the overall analysis time.

Existing approaches. The most common approach to addressing our problem involves designing a verifier that analyzes the local robustness of an ϵ -ball around an input and invoke it on every input in S one by one. However, this approach does not leverage the similarity of the network computations, which leads to a long analysis time. To mitigate this, several studies have proposed reusing analysis computations. For instance, Ugare et al. [2022] generate and transform templates that capture symbolic shapes at intermediate network layers, allowing proof computations to be reused across multiple approximate versions of a network. However, this technique is tailored for proof transfer across similar networks rather than across different inputs. Fischer et al. [2022] propose the concept of shared certificates, which leverages the proofs of certain ϵ -balls to speed up the verification of other ϵ -balls, through a two-step process: offline template generation and inference. During the template generation, a large set of ϵ -balls of training inputs (e.g., several thousands [Fischer et al. 2023]) are verified one by one. The intermediate analysis results (e.g., zonotopes or polyhedrons) are attempted to be generalized to *templates* through clustering, convex-hull extension, and other expansion techniques. These templates, encoded in the box or the star domain [Bak and Duggirala 2017; Tran et al. 2019], are subsequently verified using an exact verifier. At inference, an ϵ -ball begins the analysis and after every layer in which templates were generated, it is checked whether its analysis result is contained in one of the templates. If yes, the analysis terminates; otherwise, the analysis continues as usual. While shared certificates have been shown successful, they are coupled to the chosen abstract domain, which limits them to incomplete verification. Consequently, the network designer may not understand the actual local robustness in the given set of ϵ -balls, which can lead to incorrect conclusion when comparing the robustness of networks to one another. Additionally, the training time has high overhead (multiple hours). Further, the training procedure is invoked once and does not consider the ϵ -balls that are later analyzed. In all existing approaches, the local robustness analysis is performed ϵ -ball by ϵ -ball, as illustrated in Figure 1. While shared certification analysis aims at reducing the joint analysis time, it assumes that future unseen ϵ -balls have similar intermediate analysis results as the ϵ -balls of the inputs in the training set.

3 Overview: Mini-Batch Complete Verification

In this section, we describe our approach to group local robustness verification: dynamically identifying small subsets of inputs – called mini-batches – whose ϵ -balls are likely to be successfully

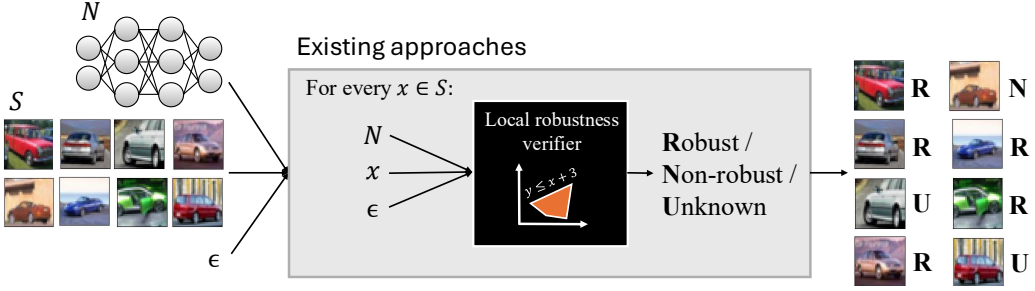
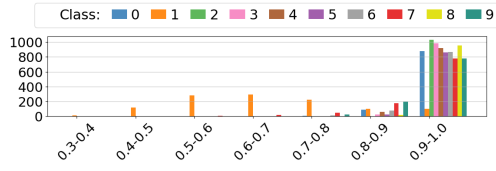


Fig. 1. Existing approaches analyze local robustness for each input's ϵ -ball separately. Complete verifiers determine for each ϵ -ball whether it is robust or not robust. Incomplete verifiers may also return unknown.



(a) Histogram of L_∞ distance of MNIST closest pairs.



(b) CIFAR-10's first 8 car images (top) and the first 8 images after sorting by SSIM (bottom).

verified together. At high-level, our verification relies on an abstraction-refinement procedure, similarly to [Ostrovsky et al. \[2022\]](#); [Singh et al. \[2019b\]](#); [Wang et al. \[2018a,b\]](#). However, to minimize the analysis time, there are several inherent questions to address: (1) what computations are abstracted given a mini-batch, (2) how to identify how many and which inputs to include in mini-batches and (3) how to perform refinement. We next discuss our ideas to address these questions.

3.1 Abstraction

Given a set of inputs S , the most naive approach abstracts all their ϵ -balls. However, it is very likely to fail proving robustness, especially if the inputs are classified differently or if the inputs are not very close to one another. This is caused by two reasons. First, existing local robustness verifiers determine robustness by checking that all inputs in the given neighborhood are classified the same, thus abstracting inputs of different classes will lead the verifier to determine that the abstracted neighborhood is not robust. Second, the more different the inputs, the higher the overapproximation error and the more likely it contains spurious counterexamples, which will lead to failure.

A more natural approach abstracts the ϵ -balls of inputs that are classified the same and are close, based on some similarity metric. A natural candidate for similarity metric is the L_∞ distance. However, even for the MNIST dataset, whose images are relatively similar, the images are not close enough to keep the overapproximation low. Figure 2a shows a histogram of the L_∞ distance of every MNIST test image and its closest image. It shows that the closest pair has distance of 0.33 (typically, the radius of the ϵ -ball is much smaller), and that most pairs have significantly larger distance. For images, a better similarity metric is the structural similarity (SSIM) index [[Wang et al. 2004](#)] (illustrated in Figure 2b) or LPIPS [[Zhang et al. 2018](#)]. In Section 5, we show that abstracting in the input layer based on these metrics leads to a large analysis time and fails proving robustness for most ϵ -balls. The reason is that despite the similarity, the abstraction still adds too many spurious inputs, which increases the analysis time at best and leads to spurious counterexamples at worst.

Instead, we rely on the following observation: given inputs classified to the same class, abstracting in a deeper layer loses less precision. Intuitively, the reason is that the output vectors of such inputs

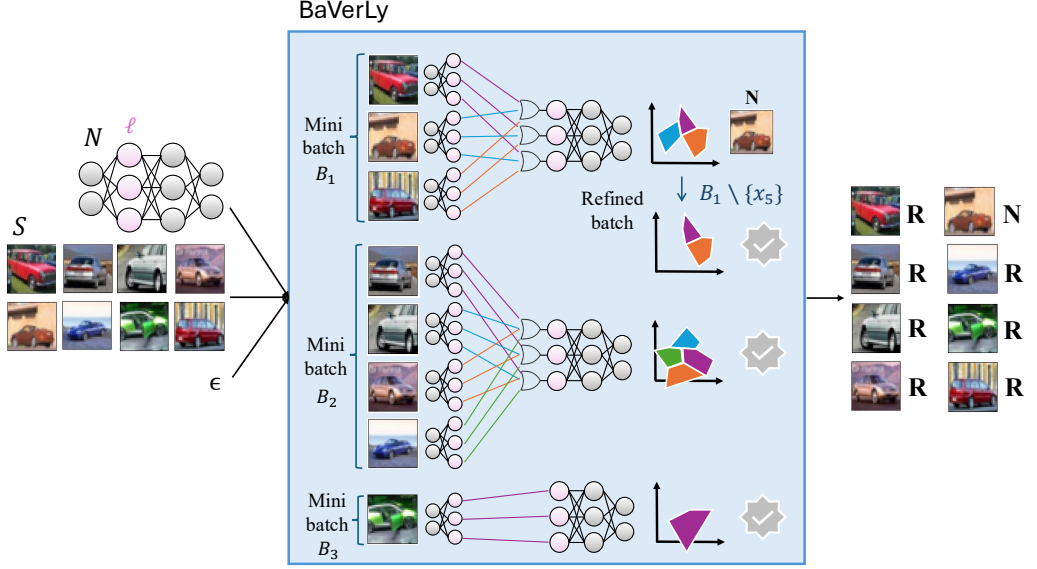


Fig. 3. Our approach for complete group local robustness verification forms mini-batches and analyzes them jointly starting from an intermediate layer. If a mini-batch fails, our verifier identifies a potentially non-robust ϵ -ball and analyzes it separately. It then continues the analysis for the remaining ϵ -balls in the mini-batch.

tend to become closer for deeper layers, where ultimately the last output vectors are equivalent in terms of the chosen classification. This observation is supported theoretically: the *information bottleneck principle in deep learning* [Tishby and Zaslavsky 2015] states that neural networks compress the input to enhance generalization. That is, the input layer has a raw representation of the input x , where not all pixels contribute to its classification. As the input is propagated through the network, each layer processes the representation of the previous layer, extracting the meaningful information while discarding irrelevant details. Thus, by abstracting ϵ -balls in an intermediate layer, we can focus on the network computations where they are *perceived similar*, which is more effective to expedite the analysis. Inspired by this observation, we join the analysis of the batch's ϵ -balls in an intermediate layer ℓ . This observation has also been leveraged in shared certificates [Fischer et al. 2022], which are formed by templates in an intermediate layer. Unlike shared certificates, we do not compute templates with the goal of expediting the analysis of future unseen ϵ -balls, but batch the analysis of subsets of ϵ -balls. Figure 3 illustrates our approach, called BaVerLy. Given a network, a set of inputs and an ϵ , it iteratively forms batches (we explain how shortly). For each batch, it verifies each ϵ -ball separately up to layer ℓ (the choice of ℓ is described in Section 4.5). It then continues their analysis together. If the analysis succeeds, all ϵ -balls of the batch are proven robust. If not, BaVerLy identifies an ϵ -ball that may be not robust (we explain how later). It then analyzes this ϵ -ball separately and continues the joint analysis for the remaining batch. We explain later in this section why our refinement steps lead to very low overhead.

3.2 Mini-Batches

Our second observation is that to balance well the precision-scalability trade-off, it is best to abstract to *mini-batches*. That is, unify the verification of relatively small subsets of ϵ -balls. While our algorithm works for any batch size, large batches increase the time overhead substantially

because of their overapproximation error. Beyond balancing well the precision-scalability trade-off, there is an additional advantage in verifying mini-batches when the set of inputs S is large: it enables BaVerLy to learn the best mini-batch size based on previous mini-batches. This is possible because the group verification problem is invariant to the order in which ϵ -balls are verified. Naturally, the best batch size depends on the inputs in the batch, thus our batches consist of inputs whose ϵ -balls have similar network computations. We next describe how BaVerLy learns the best batch size from previous mini-batches and how it clusters inputs into a mini-batch of a selected size.

Learning the mini-batch size. Choosing a good batch size is challenging. On the one hand, too large mini-batches can substantially increase the analysis time due to the overapproximation error. On the other hand, too small batches may also increase the analysis time, similarly to verifying the ϵ -balls one by one. The best batch size also depends on the inputs in S : the more similar inputs in S , the more effective larger mini-batches are. Even if BaVerLy begins by grouping the most similar ϵ -balls to relatively large mini-batches, as the analysis progresses, the remaining ϵ -balls are likely to have more distant network computations, for which smaller mini-batches are more effective. We rely on an adaptive approach to learn the optimal batch size on the fly. Our approach adopts a strategy from reinforcement learning (RL), where an agent learns a policy that maximizes the received reward. In our setting, the optimal policy's goal is to predict batch sizes that enable BaVerLy to minimize the analysis time per input in S . We formalize this goal as increasing the *batch velocity*, that is the number of ϵ -balls which were proven robust within the batch verification, divided by the analysis time of the batch (excluding its refinements). We note that the concept of partitioning a local robustness task by predicting the subparts that maximize the proof velocity has been proposed by others [Kabaha and Drachsler-Cohen 2022], however they focus on verifying the local robustness of a single semantic feature neighborhood. An inherent dilemma of an RL agent is the *exploration-exploitation trade-off*. In our context, this means that whenever our RL agent chooses a mini-batch size it can choose between exploring new, potentially effective mini-batch sizes (which may be discovered as less effective) or exploiting batch sizes that have been shown to be reasonably effective (which may lead to not discovering more effective batch sizes). In our setting, this problem becomes even more challenging since the ϵ -balls can vary in the location of their perturbations and in the similarity level of their network computations. To cope, we frame the problem of predicting the best mini-batch size as a *multi-armed bandit* (MAB) scenario and rely on the Thompson Sampling [Thompson 1933] that seamlessly balances reward maximization (exploitation) and variance minimization (exploration). Technically, we introduce a different arm for each batch size. For each batch size, we learn a distribution that converges to the velocity of batches with this size. The distributions are updated throughout the execution of BaVerLy. Figure 4 illustrates our approach for learning the best batch size.

Constructing effective mini-batches. The next question is how to construct a mini-batch, given the selected size k . As mentioned, BaVerLy aims at grouping the k inputs whose ϵ -balls exhibit the most similar network computations. This raises two questions: (1) how to identify the ϵ -balls with the closest network computations and (2) how to identify k such ϵ -balls? The first question is particularly challenging since BaVerLy does not know what the network computations of a given ϵ -ball are without performing its analysis. Obviously, if BaVerLy performed the analysis of every ϵ -ball separately, there would be no point in the mini-batch analysis. Instead, we estimate the closeness of two ϵ -balls by the similarity of the network computations for their center inputs. This is obtained by first running every input in S through the network, which introduces negligible overhead. We then abstract the network computation of every input by its *activation pattern*. An input x 's activation pattern is a boolean vector consisting of a bit for every ReLU neuron in the network. A bit is one if the respective neuron is *active* (i.e., positive) when propagating x through

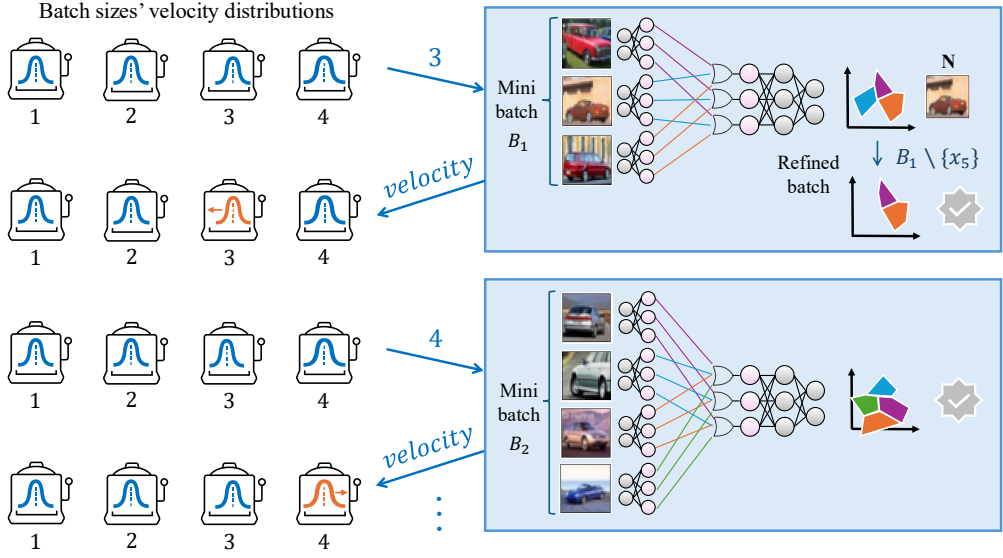


Fig. 4. Our approach for learning the best mini-batch size. We phrase the problem as a multi-armed bandit and learn the velocity distribution of every batch size. At every iteration, BaVerLy samples the mini-batch size by the Thompson Sampling. It then constructs a mini-batch and verifies it. Afterward, the batch velocity is computed and the respective batch size's distribution is updated.

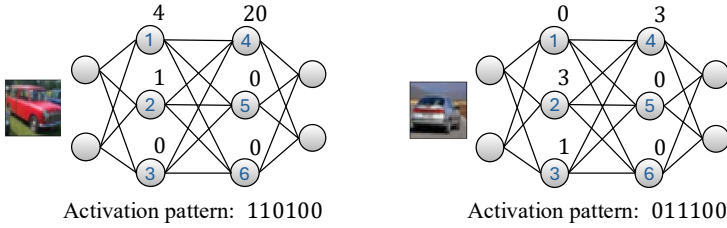


Fig. 5. Given an input, an activation pattern abstracts the computation of the intermediate neurons by a boolean vector whose i^{th} entry is one, if the i^{th} neuron outputs a positive value, and zero otherwise.

the network, and zero otherwise. Figure 5 exemplifies the activation patterns of two inputs. The advantage of relying on the similarity of activation patterns rather than metrics at the input layer is that it estimates the increase in the verification's complexity caused by the unification of ϵ -balls. At high-level, the verification's complexity is exponential in the number of unstable ReLUs. Given a neighborhood of inputs, the unstable ReLU neurons are the neurons whose weighted sum inputs can be both positive and negative. Namely, these ReLUs can be both active and inactive, making the ReLU computation nonlinear in this neighborhood. The fewer the unstable ReLU neurons, the lower the verification's complexity. While the activation pattern of an input does not indicate which neurons are unstable in its ϵ -ball (since it does not consider every possible input in the ϵ -ball), inputs which differ in the activation state of a certain neuron, imply that this neuron must be unstable if we unify their ϵ -balls. For example, consider neuron 1 in Figure 5. It is in active state for the first image and in inactive state for the second image. If we unify these images' ϵ -balls into one neighborhood, neuron 1 must be unstable. On the other hand, for neurons 2 and 6 in Figure 5, both

images have the same state (for both, neuron 2 is active and neuron 6 is inactive). Thus, although it may be that these neurons are unstable if we unify these images' ϵ -balls, it may also be that these neurons are stable. We note that if a certain neuron has different active/inactive states for two images, it does not necessarily mean that if we unify the two images' ϵ -balls the verification's complexity increases, since it could be that one of their ϵ -balls makes this neuron unstable. This is our motivation for preferring to unify ϵ -balls whose center inputs have close activation patterns. We measure the distance of two activation patterns by their Hamming distance (i.e., the number of different bits). For example, the Hamming distance of the two images in Figure 5 is 2, since their first and third bits are different. Relying on activations to identify network similarities has been proposed in prior work. For example, Szegedy et al. [2013] show that activation values of neurons in the hidden layers encode semantic information about the features seen in the image. In particular, inputs that share many common features tend to have close activation patterns. Krizhevsky et al. [2012] show that the network perceives images as semantically similar when their deeper layer activations are proximal, even when the images' pixels differ substantially.

We now explain how BaVerLy forms a mini-batch of up to k inputs. A natural idea is to rely on clustering algorithms, such as K-Means [MacQueen 1967]. However, most clustering algorithms are effective in clustering a set of elements into a certain number of clusters, whereas we are interested in clusters of *given sizes* and the given sizes *change* during the execution of BaVerLy. We thus rely on *Hierarchical Clustering (H-Cluster)* [Ward 1963]. H-Cluster has been proposed in phylogenetics for revealing the evolutionary ancestry between a set of genes, species, or taxa. Given a set of vectors, the H-Cluster greedily constructs a diagrammatic representation of the clusters hierarchy, called a *dendrogram*. Figure 6a shows an example of a dendrogram over six activation patterns, where the number of an internal node is the maximum Hamming distance of the activation patterns in its subtree. For example, the Hamming distance between x_3 and x_4 is 420 and the distance between x_5 and x_6 is 477. The largest Hamming distance between every pair of inputs in $\{x_3, x_4, x_5, x_6\}$ is 595. BaVerLy transforms the dendrogram into a binary tree. Every leaf corresponds to an input $x \in S$ and inner nodes represent clusters consisting of all leaves in their subtrees. Figure 6b shows an example of this binary tree, where the numbers in the inner nodes are the size of their cluster. This binary tree enables BaVerLy to efficiently track the remaining inputs to verify and construct batches of given sizes. To construct a batch of size up to k , it traverses the tree in pre-order, stopping at the first node whose number is less than or equal to k . Then, it forms the batch by collecting the leaves and removes this subtree. The search time complexity is $O(\log |S|)$ on average and $O(|S|)$ in the worst case (since the binary tree can be unbalanced).

Refinement. We next discuss what BaVerLy does if the verification of a mini-batch finds a counterexample. We remind that our mini-batch verification analyzes every ϵ -ball of the mini-batch separately until some layer ℓ and then analyzes the ϵ -balls jointly till the output layer. A straightforward encoding of a mini-batch abstracts the ϵ -balls' outputs at layer ℓ into the bounding box containing all these outputs. However, this increases the input space of layer $\ell + 1$ significantly, possibly including spurious adversarial examples, which will fail the verification. Even if the verifier finds a true adversarial example, it can require time to determine this is the case. Instead, we propose to encode the *union* of the ϵ -balls' outputs at layer ℓ . Technically, we define MILP constraints capturing a disjunction that restricts the inputs to layer $\ell + 1$ to exactly the ϵ -balls' outputs at layer ℓ . Our constraints associate a binary variable for each ϵ -ball of the batch. If an adversarial example is found, one of these binary variables is one, indicating that the adversarial example belongs to the respective ϵ -ball. Because the mini-batch analysis begins from an intermediate layer (and not the first layer), this adversarial example may be spurious. Thus, BaVerLy excludes the ϵ -ball that may not be robust and analyzes it separately from the first layer to the last one. Accordingly, it

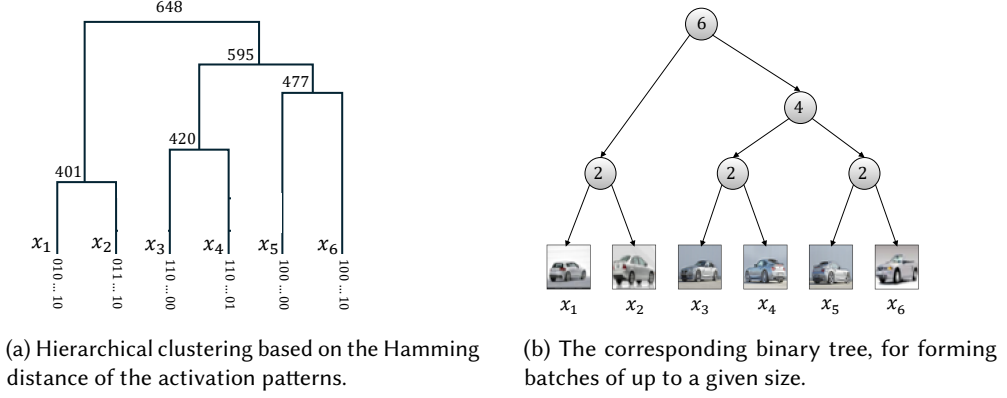


Fig. 6. Illustration of the H-Cluster algorithm.

determines whether it is robust or not. Then, BaVerLy continues the analysis for the remaining mini-batch. An advantage of our refinement step is that neither the analysis of the ϵ -ball that may not be robust nor the analysis of the remaining mini-batch begin from scratch. Both use the previous analysis computations to prune their search space. These two ideas enable our refinement step to introduce very low overhead.

4 BaVerLy: A Batch Verifier for Local Robustness

In this section, we present our group local robustness verifier. We describe its algorithm and then its components. Appendix A provides a running example.

4.1 BaVerLy's Algorithm

BaVerLy takes a network classifier N , a set of inputs S , a class c , and an ϵ . It returns a dictionary `is_robust` mapping every input in $x \in S$ to *Robust*, if N classifies all inputs in the ϵ -ball $B_\epsilon^\infty(x)$ as c , or to *Non-Robust* otherwise. It begins by passing each $x \in S$ through N (Lines 3–4). If x is not classified as c by N , BaVerLy tags it as *Non-Robust* and removes it from S . Then, BaVerLy chooses the split layer ℓ by calling `learnSplitLayer` (Line 5), defined in Section 4.5. At high-level, `learnSplitLayer` verifies $L - 1$ ϵ -balls of random inputs from S , where each verification splits at a different layer. Accordingly, it chooses for ℓ the layer with the minimal verification time. It removes the analyzed inputs from S and records their status in `is_robust`. Then, BaVerLy computes the activation patterns (defined in Section 4.3) for every input in S and adds it to a dictionary `AP` (Line 6). Then, it constructs the clusters' dendrogram and its binary tree \mathcal{T} (Line 7), described in Section 4.3, storing the unhandled inputs. It then initializes the multi-armed bandit agent \mathcal{MAB} (Line 8), which learns the optimal mini-batch size (described in Section 4.4). \mathcal{MAB} is initialized with the maximal mini-batch size `MAX_BATCH_SIZE` (a hyper-parameter) and other arguments, described later.

Then, the main loop runs while there are inputs in the tree \mathcal{T} (Line 9). At each iteration, \mathcal{MAB} recommends a batch size k (Line 10). BaVerLy then searches \mathcal{T} for a mini-batch B of up to size k , constructs it (Line 11) and updates k to be the actual batch size (Line 12). For every input x in the mini-batch, BaVerLy analyzes its ϵ -ball separately up to layer ℓ (Lines 13–16). This analysis computes for every $B_\epsilon^\infty(x)$ and every layer real-valued bounds, using MILPs (described in Section 4.2). The bounds are stored in a dictionary `bounds` mapping input to its bounds, i.e., `bounds[x]` is a list of lists, where the i^{th} list contains the bounds of the neurons in layer N_i .

Algorithm 1: BaVerLy (N, S, c, ϵ)

Input: A neural network N , a set of inputs S , a class c , and an epsilon $\epsilon \geq 0$.
Output: A dictionary `is_robust`, reporting *Robust* or *Non-robust* for every $x \in S$.

```

1 is_robust = []           // Dictionary mapping inputs to Robust or Non-Robust
2 AP = []                 // Dictionary mapping inputs to activation patterns
3 for  $x \in S$  do
4   if  $\arg \max N(x) \neq c$  then is_robust[ $x$ ] = Non-Robust;  $S = S \setminus \{x\}$ 
5  $\ell, S, \text{is\_robust} = \text{learnSplitLayer}(N, S, c, \epsilon, \text{is\_robust})$ 
6 for  $x \in S$  do AP[ $x$ ] =  $\text{activation\_pattern}(N, x)$ 
7  $\mathcal{T} = \text{getBinaryTree}(\text{H-Cluster}(\text{AP}))$ 
8  $\mathcal{MAB} = \text{initialize}(\text{MAX\_BATCH\_SIZE}, \rho, \text{BUCKET\_SIZE})$ 
9 while  $\mathcal{T} \neq \perp$  do
10   $k = \mathcal{MAB}.\text{getMiniBatchSize}()$ 
11   $B = \text{constructBatch}(\mathcal{T}, k)$ 
12   $k = |B|$  // Actual batch size
13  bounds = [] // Dictionary mapping inputs to their bounds up to  $N_\ell$ 
14  start_time = current_time()
15  for  $x \in B$  do // Compute bounds up to  $N_\ell$  for each input
16    bounds[ $x$ ] =  $\text{MILPBounds}(N_\ell \circ \dots \circ N_1, x, \epsilon)$ 
17  MILP_SUF =  $\text{MILPBatch}(N_\ell \circ \dots \circ N_{\ell+1}, \{\text{bounds}[x][\ell] \mid x \in B\}, c)$ 
18  cex =  $\text{MILPSolve}(\text{MILP\_SUF})$  // Verify the batch
19  total_time = current_time() - start_time
20  while cex  $\neq \perp$  do
21    Let  $x$  be the input whose variable  $I_x$  is 1 in cex //  $B_\epsilon^\infty(x)$  may be non-robust
22    cex =  $\text{MIPVerify}(N, x, \epsilon, \text{bounds}[x])$  // Verify  $B_\epsilon^\infty(x)$ 
23    is_robust[ $x$ ] = cex ==  $\perp$ ? Robust : Non-Robust
24     $B = B \setminus \{x\}$  // Update the batch
25    if  $B == \emptyset$  then break
26    start_time = current_time()
27    MILP_SUF =  $\text{MILP\_SUF} :: \{I_x = 0\}$  // Ignore the  $\epsilon$ -ball of  $x$ 
28    cex =  $\text{MILPSolve}(\text{MILP\_SUF})$  // continue with the rest
29    total_time = total_time + (current_time() - start_time)
30  for  $x \in B$  do is_robust[ $x$ ] = Robust // All  $\epsilon$ -balls are robust
31   $\mathcal{MAB}[k].\text{updateDistribution}(\frac{|B|}{\text{total\_time}})$  // Update based on the velocity
32 return is_robust

```

Next, BaVerLy verifies the mini-batch. It begins by encoding the mini-batch verification problem over all inputs in B from layer $N_{\ell+1}$ to the output layer (Line 17), where the input space to layer $N_{\ell+1}$ is the union of the bounds of layer N_ℓ over all inputs. We describe the encoding in Section 4.2. At high-level, it consists of constraints such that if they are satisfiable, there is a vector v , which is not classified as c , contained within the bounds of N_ℓ of some input $x \in B$. This vector is possibly an indication to an adversarial example within $B_\epsilon^\infty(x)$. This is the case if there is $x' \in B_\epsilon^\infty(x)$ such that v is the output of layer N_ℓ for x' , i.e., $v = N_\ell \circ \dots \circ N_1(x')$. Because the bounds provide an

overapproximation, it can happen that there is no such x' in $B_\epsilon^\infty(x)$. Our encoding of the union relies on a binary variable I_x , for each $x \in B$, and on constraints that guarantee that if the MILP is satisfiable, exactly one I_x is equal to 1. If $I_x = 1$ for $x \in B$, then the satisfying assignment includes a vector v within the bounds of N_ℓ for $B_\epsilon^\infty(x)$ that is not classified as c . This union encoding enables BaVerLy to not lose precision at the layer N_ℓ and identify *which* input from B *may* be non-robust. Our MILP encoding is submitted to a solver, which returns an assignment cex or \perp (Line 18).

If the MILP solver finds an assignment cex , BaVerLy begins a loop to refine the batch while there exists a counterexample (Line 20). At each iteration, it first identifies the input x whose bounds contain the counterexample v (Line 21), i.e., the input x whose binary I_x is equal to one. Then, it verifies the local robustness of N in x 's ϵ -ball separately (Line 22). This analysis is identical to MIPVerify [Tjeng et al. 2019], on which our MILP encoding builds, except that BaVerLy leverages the prior analysis and provides it with the bounds for all layers up to ℓ to expedite the analysis (explained in Section 4.2). This analysis is complete, and thus BaVerLy concludes whether N is robust in x 's ϵ -ball, depending on whether the MILP solver finds a counterexample (Line 23). Then, BaVerLy removes x from the batch B (Line 24). If B is empty, it breaks from the inner loop (Line 25). Otherwise, BaVerLy updates the MILP to ignore x 's bounds by forcing $I_x = 0$ (Line 27) and calls the solver to check if there is another counterexample (Line 28). Note that this analysis continues from the point that the solver terminates and does not start from scratch the computation.

The inner loop terminates when there is no counterexample, implying that all inputs in B are robust. Thus, BaVerLy updates their robustness status in `is_robust` (Line 30). It then updates the \mathcal{MAB} agent with the velocity of this batch (Line 31). The velocity is the number of ϵ -balls proven robust as part of the batch (i.e., the size of B at the end of the inner loop) divided by the overall analysis time of the batch. This analysis time is the total time of (1) computing the bounds of all inputs in the initial B up to layer ℓ , (2) computing the bounds of the batch starting from layer $\ell + 1$, and (3) looking for counterexamples in the batch. This time excludes the time to prove robustness of ϵ -balls suspected as not robust (Line 22), since they are not proven as part of the batch.

Beyond group verification. While we focus on group local robustness verification, we believe our approach can expedite the verification of other properties. It is applicable to any safety property whose input space can be partitioned into subspaces. For example, semantic feature neighborhoods often contain a large set of inputs that cannot be analyzed jointly and are split statically or dynamically (e.g., [Kabaha and Drachler-Cohen 2022; Singh et al. 2019a]). For such neighborhoods, BaVerLy can be extended to get as input the subspaces (explicitly or symbolically). Then, it constructs mini-batches and verifies them as described. If a mini-batch is not robust, our union encoding (relying on the binary variables I_x) enables BaVerLy to identify the subspace that may not be robust, analyze it separately, and continue verify the rest of the mini-batch. A key advantage of BaVerLy is that continuing verifying a mini-batch, after removing a subspace (by setting its binary variable $I_x = 0$), completes very fast, since it relies on the analysis of the original mini-batch.

4.2 Batch Local Robustness Verification by Mixed Integer Linear Programming

In this section, we present how we rely on mixed-integer linear programming (MILP) for batch local robustness verification. BaVerLy relies on three MILPs: for computing bounds on the neurons in the early layers, for verifying a batch in the subsequent layers and for verifying the robustness of ϵ -balls suspected as non-robust. Our MILPs rely on a prior encoding of local robustness verification of a single ϵ -ball [Tjeng et al. 2019]. We next describe it and our encodings.

MIPVerify. MIPVerify [Tjeng et al. 2019] is a verifier for determining the local robustness of a neural network using MILP. It takes a network classifier N , an input $x \in [0, 1]^{d_{in}}$ and its class c , and an $\epsilon > 0$. It determines whether N is robust in the L_∞ ϵ -ball of x or not, in which case it

Algorithm 2: MILPBounds($N_\ell \circ \dots \circ N_1, x, \epsilon$)

```

1  $l = []; u = []$ 
2  $\text{constraints} = \{\max(0, x_i - \epsilon) \leq z_{0,i} \leq \min(1, x_i + \epsilon) \mid i \in [d_{in}]\}$ 
3 for  $i = 1; i \leq \ell; i++$  do
4   for  $m = 1; m < m_i; m++$  do
5      $l_{i,m} = \text{MILPSolve}(\min b_{i,m} + \sum_{m'=1}^{m_i-1} w_{i,m,m'} \cdot z_{i-1,m'} \text{ subject to constraints})$ 
6      $u_{i,m} = \text{MILPSolve}(\max b_{i,m} + \sum_{m'=1}^{m_i-1} w_{i,m,m'} \cdot z_{i-1,m'} \text{ subject to constraints})$ 
7    $\text{constraints} = \text{constraints} \cup \text{MIPVerify\_constraints}(N_i, l_i, u_i)$ 
8 return  $(l, u)$ 

```

returns an adversarial example. MIPVerify is sound and complete. Its complexity is exponential in the number of unstable ReLU neurons. It can analyze classifiers with fully connected, convolutional, and max-pooling layers. We describe its constraints for fully connected layers, but our implementation supports the other layers. MIPVerify begins by encoding the input layer's neurons with variables $z_{0,1}, \dots, z_{0,d_{in}}$ and enforcing the neighborhood with linear constraints: $\forall m \in [d_{in}]. z_{0,m} \geq \max(0, x_m - \epsilon) \wedge z_{0,m} \leq \min(1, x_m + \epsilon)$. Then, it constructs the constraints capturing the network computations layer by layer. For every layer i with m_i neurons, it adds m_i linear constraints for capturing the affine computation: $z'_i = W_i z_{i-1} + b_i$. Technically, the vector z'_i is captured by m_i variables $z'_{i,1}, \dots, z'_{i,m_i}$. Then, it computes real-valued lower and upper bounds $l_{i,m}, u_{i,m}$ for $z'_{i,m}$ via optimization. This is computed by taking the constraints of all layers up to layer $i-1$ and for each neuron $m \in [m_i]$, solving two optimization problems (unless skipped by heuristics), one with objective $u_{i,m} = \max z'_{i,m}$ and the other one with objective $l_{i,m} = \min z'_{i,m}$. Then, if $l_{i,m} \geq 0$, the neuron is *active*, i.e., its function is the identity function: $z_{i,m} = z'_{i,m}$. Similarly, if $u_{i,m} \leq 0$, the neuron is *inactive*, i.e., its function is the constant 0: $z_{i,m} = 0$. Otherwise, $l_{i,m} < 0 \wedge u_{i,m} > 0$, the neuron is *unstable*, i.e., its function is piecewise linear and thus it is not expressible as a single linear constraint. To encode the ReLU computation precisely, MIPVerify introduces a binary variable $a_{i,m}$ that captures the two possible states and adds four constraints over $a_{i,m}, z'_{i,m}$ and the bounds $l_{i,m}, u_{i,m}$. After generating the constraints of all layers (the output layer has no ReLUs but its bounds are computed), MIPVerify adds an objective function and a constraint whose goal is to find the minimum adversarial perturbation that is misclassified: $\min_{x'} \|x - x'\|_\infty$ s.t. $z_{L,c} \leq \max_{c' \neq c} z_{L,c'}$, where $x' = (z_{0,1}, \dots, z_{0,d_{in}})$ and $x = (x_1, \dots, x_{d_{in}})$. It then submits all constraints to a MILP solver. If the MILP solver determines that the set of constraints is infeasible, then N is robust in this ϵ -ball. If it finds a satisfying assignment, the values $z_{0,1}, \dots, z_{0,d_{in}}$ form an adversarial example.

BaVerLy's MILPs. BaVerLy relies on this MILP encoding for three tasks: (1) for computing the bounds up to layer ℓ (Lines 13–16), (2) for batch verification (Line 17, Line 27), and (3) for verifying the local robustness of an ϵ -ball suspected as not robust (Line 22). We next describe these MILPs.

Bound computation. Algorithm 2 shows the bound computation for every neuron in the first ℓ layers of N , given the ϵ -ball of an input x . It first adds constraints bounding each input neuron within its interval, based on x and ϵ (Line 2). Then, for every layer i , it iterates the neurons and, for each, computes a lower and an upper bound on the affine function (Lines 5–6) by calling a MILP solver with all current constraints. Afterwards, it adds the layer's constraints (similar to Equations (2c) and (2d), but with respect to l_i, u_i instead of L_i, U_i) and continues to the next layer.

Batch verification. We next describe how BaVerLy forms a MILP for batch verification over the layers $N_{\ell+1}, \dots, N_L$. Given a batch B and the bounds of N_ℓ for all inputs in B , BaVerLy first defines

the input space of $N_{\ell+1}$ as the union of the outputs of layer N_ℓ . Then, it computes the real-valued bounds for every layer from $N_{\ell+1}$ to the output layer. Accordingly, it computes the same constraints as MIPVerify for these layers and adds the same constraint to look for an adversarial example. As in MIPVerify, if this MILP is infeasible, then there is no adversarial example, implying that *all* ϵ -balls of the inputs in the batch are robust. Otherwise, some ϵ -ball may be not robust.

We next present our encoding for the inputs to $N_{\ell+1}$, which provides a simple way to identify the input in the batch whose ϵ -ball may be non-robust. The input to $N_{\ell+1}$ is the output of layer N_ℓ . A straightforward encoding is to bound each of its outputs in its minimal containing interval, i.e.,

$$\forall m. z_{\ell,m} \in [\min(\{bounds[x].l_{\ell,m} \mid x \in B\}), \max(\{bounds[x].u_{\ell,m} \mid x \in B\})].$$

However, this results in a very high overapproximation error and is also difficult to identify a good refinement if an adversarial example is detected. Instead, we wish to encode a disjunction over the outputs of N_ℓ , thereby forcing the input to $N_{\ell+1}$ to be contained in one of them:

$$\bigvee_{x \in B} \bigwedge_{m \in [m_\ell]} (z_{\ell,m} \geq bounds[x].l_{\ell,m} \wedge z_{\ell,m} \leq bounds[x].u_{\ell,m})$$

However, disjunctions are not directly expressible in MILPs. Thus, we propose a MILP encoding adapting the big-M method for expressing the maximum function [Winston 1991]. Our encoding captures a function that takes a finite set of intervals and outputs a value in one of them. Formally, given k intervals $[l_1, u_1], \dots, [l_k, u_k]$ such that $l_i \geq 0$ for every $i \in [k]$, our encoding introduces k binary variables $I_1, \dots, I_k \in \{0, 1\}$ and a real-valued variable y for the output that is contained in one of the intervals. Our constraints force that: (1) exactly one interval is picked (by requiring that the sum of the binary variables is one) and (2) if $I_i = 1$, then $y \in [l_i, u_i]$. This is encoded by two types of constraints, each has a copy for each of the k intervals. The first type of constraints forces $y \geq l_i$ in case $I_i = 1$. The second type of constraints forces $y \leq u_i$ in case $I_i = 1$. We further denote the “big-M” as the maximum upper bound $u_M = \max(u_1, \dots, u_k)$. Our encoding is:

$$\sum_{i=1}^k I_i = 1 \tag{1a}$$

$$\forall i \in [k] : y \geq l_i \cdot I_i \tag{1b}$$

$$\forall i \in [k] : y \leq u_i \cdot I_i + u_M \cdot (1 - I_i) \tag{1c}$$

THEOREM 4.1. *Equation (1) is feasible if and only if there exists $y \in [l_i, u_i]$ for some $i \in [k]$.*

Appendix B shows the proof. BaVerLy uses this encoding to bound the output of N_ℓ . It introduces $k = |B|$ binary variables I_1, \dots, I_k , and then, for each $m \in [m_\ell]$, it adds the above constraints for $y = z_{\ell,m}$ and the intervals $[bounds[x].l_{\ell,m}, bounds[x].u_{\ell,m}]$ for every $x \in B$. It also relies on real-valued bounds $L_{i,m}$ and $U_{i,m}$ for all $i \in \{\ell + 1, \dots, L\}$ and $m \in [m_i]$, which are computed as described before, by solving the optimizations $L_{i,m} = \min z'_{i,m}$ and $U_{i,m} = \max z'_{i,m}$ over all constraints of layers $\ell, \dots, i - 1$. Overall, given a batch B with k inputs and their bounds $[l_{\ell,m}^j, u_{\ell,m}^j]$ for every $j \in [k], m \in [m_\ell]$, the batch verification is encoded by MILP_SUF:

$$\forall j \in [k] : I_j \in \{0, 1\}, \quad \sum_{j=1}^k I_j = 1 \tag{2a}$$

$$\forall j \in [k], \forall m \in [m_\ell] : \quad z_{\ell,m} \geq l_{\ell,m}^j \cdot I_j, \quad z_{\ell,m} \leq u_{\ell,m}^j \cdot I_j + u_{M,m} \cdot (1 - I_j) \tag{2b}$$

$$\forall i > \ell, \forall m \in [m_i] : \quad z'_{i,m} = b_{i,m} + \sum_{m'=1}^{m_i-1} w_{i,m,m'} \cdot z_{i-1,m'} \tag{2c}$$

Algorithm 3: MILPBatch($N_L \circ \dots \circ N_\ell, \{[l_{\ell,m}^j, u_{\ell,m}^j] \mid m \in [m_\ell], j \in [k]\}, c$)

```

1  $L = []; U = [];$ 
2  $\text{constraints} = \text{MILP\_OR}(\{[l_{\ell,m}^j, u_{\ell,m}^j] \mid m \in [m_\ell], j \in [k]\})$ ; // Equations (2a) and (2b)
3 for  $i = \ell + 1; i \leq L; i++$  do
4   for  $m = 1; m < m_i; m++$  do
5      $L_{i,m} = \text{MILPSolve}(\min b_{i,m} + \sum_{m'=1}^{m_i-1} w_{i,m,m'} \cdot z_{i-1,m'} \text{ subject to constraints});$ 
6      $U_{i,m} = \text{MILPSolve}(\max b_{i,m} + \sum_{m'=1}^{m_i-1} w_{i,m,m'} \cdot z_{i-1,m'} \text{ subject to constraints});$ 
7    $\text{constraints} = \text{constraints} \cup \text{MIPVerify\_constraints}(N_i, L_i, U_i);$  // (2c) and (2d)
8 return  $\text{constraints} \cup \{z_{L,c} \leq \max_{c' \neq c} z_{L,c'}\}$ 

```

$$\forall i > \ell, \forall m \in [m_i] \begin{cases} z_{i,m} = z'_{i,m} & L_{i,m} \geq 0 \\ z_{i,m} = 0 & U_{i,m} \leq 0 \\ z_{i,m} \geq 0; \quad z_{i,m} \geq z'_{i,m}; \quad z_{i,m} \leq U_{i,m} \cdot a_{i,m}; & \text{else} \\ z_{i,m} \leq z'_{i,m} - L_{i,m} \cdot (1 - a_{i,m}); \quad a_{i,m} \in \{0, 1\} \end{cases} \quad (2d)$$

$$z_{L,c} \leq \max_{c' \neq c} z_{L,c'} \quad (2e)$$

where the max function encoding is defined formally in Tjeng et al. [2019]. Algorithm 3 shows the generation of this MILP. It begins by generating the disjunction over the outputs of N_ℓ (Line 2). Then, for every layer i , it iterates the neurons and, for each, computes a lower and an upper bound on the affine function (Lines 5–6) by calling a MILP solver with the constraints up to layer $i - 1$. Then, it adds the constraints of layer i (Equations (2c) and (2d)) and continues to the next layer.

Refinement. We next explain how BaVerLy performs refinement, in case MILP_SUF is feasible. This failure can arise either from a genuinely non-robust ϵ -ball or from a spurious counterexample caused by excluding the constraints of the first layers N_1, \dots, N_ℓ . If MILP_SUF over a batch B is feasible, by Theorem 4.1, the assignment identifies $x \in B$ (where $I_x = 1$) and $y \in \text{bounds}[x][\ell]$ that is an adversarial example (i.e., $\arg\max(N_L \circ \dots \circ N_{\ell+1}(y)) \neq c$). For this x , BaVerLy performs refinement: it runs MIPVerify on the entire network N and $B_\epsilon^\infty(x)$ (Line 22). Note that refinement of B to larger sets containing x is not viable, because an adversarial example is found within the bounds of x , independently of the bounds of the other inputs in the batch (because of our disjunction encoding). To expedite MIPVerify, BaVerLy passes it the bounds for x up to layer ℓ .

4.3 Constructing Batches by Hierarchical Clustering

In this section, we describe our binary tree used for constructing the batches (in Line 11).

Goal. The goal of the binary tree is to provide an efficient approach to construct a batch in every iteration. To boost the batch verification, the inputs in the batch should have ϵ -balls whose network's computations are as similar as possible. This is because the closer the computations, the smaller the bounds of the last layers whose analysis is joined and the fewer ReLUs that become unstable in the batch. However, identifying the ϵ -balls with the closest computations requires analyzing each ϵ -ball separately, defeating the purpose of the batch verification. Instead, as described before, we approximate their closeness by the Hamming distance of the activation patterns of the inputs at the center of the ϵ -balls.

Activation patterns. The activation pattern (AP) of an input x is a boolean vector whose size equals the number of ReLU neurons in the network. An entry i in the vector is 1 or 0 depending on whether the respective neuron is active (i.e., its input is positive). Formally:

Definition 4.2 (Activation Patterns). Given a neural network N with n ReLU neurons and an input x , the *activation pattern* of x through N is a boolean vector $r^x \in \{0, 1\}^n$, where r_i^x is the state of the i^{th} ReLU when x fed into N . If the state is active (i.e., the input of the neuron i is positive given x), $r_i^x = 1$, otherwise, $r_i^x = 0$.

Unlike image similarity metrics, AP depends on the network. Thus, it captures the similarity of inputs with respect to how the network perceives them. We rely on the *Hamming distance* for measuring the distance of APs. Formally, the distance of two inputs is the number of different bits in their activation patterns: $\text{dist}(x, y) = |\{j \in [n] \mid r_j^x \neq r_j^y\}|$. The lower the Hamming distance of two APs over inputs x and y , the fewer ReLU neurons that have distinct states. Consequently, the fewer unstable neurons that stem from grouping these inputs' ϵ -balls and the lower the batch verification's complexity.

H-Cluster. Given the activation patterns of the inputs in S , BaVerLy clusters the inputs using *Hierarchical Clustering (H-Cluster)* with the complete-linkage criteria. H-Cluster greedily constructs a *dendrogram*, a diagrammatic representation of the cluster hierarchy. H-Cluster begins by forming a pairwise-distance matrix $D_{|S| \times |S|}$ of the inputs in S by the Hamming distance of their AP (i.e., $D[x, y] = \text{dist}(x, y)$). Then, H-Cluster builds the dendrogram bottom-up. Initially, it forms a list of clusters, each contains one input. At each step, it merges the clusters with the minimal distance (follows by the complete linkage criteria). The distance of two clusters is the maximal distance of any two inputs in the clusters: $\text{dist}(A, B) = \max_{x \in A, y \in B} \text{dist}(x, y)$. Figure 6a exemplifies a dendrogram.

Binary tree. Given the dendrogram over the activation patterns, BaVerLy constructs a binary tree \mathcal{T} over the inputs. The tree enables it to construct a batch of up to a given size and remove a batch with logarithmic complexities in the size of the input set S , on average. For every activation pattern r^x in the dendrogram, BaVerLy introduces a respective leaf node labeled by the input x . For every split in the dendrogram, BaVerLy introduces an inner node, and the relation between the nodes follows exactly the structure of the dendrogram. Every inner node is marked by the number of leaves in its subtree. This enables BaVerLy to easily construct a cluster up to a certain size. The number of nodes in \mathcal{T} is $O(2|S|)$ and its depth ranges between $O(\log_2(|S|))$ and $O(|S|)$. The lower bound is obtained when at every iteration of H-Cluster, all clusters are merged with some cluster, resulting in a full and complete binary tree. The upper bound is obtained when the first iteration of H-Cluster merges two inputs, and afterwards every iteration merges the largest cluster with a singleton cluster. Figure 6b illustrates a binary tree constructed by BaVerLy. We next describe how BaVerLy constructs a batch and how it removes a batch. To form a batch of up to size k , BaVerLy runs a pre-order traversal from the root of \mathcal{T} . When it reaches a node whose number of leaves is at most k , it forms a batch that consists of all its leaves (by continuing the pre-order traversal) and returns it. Pruning a batch is obtained by removing the inner node that BaVerLy used to construct the batch and updating the batch sizes in each node along the path back to the root. Thus, this operation's average complexity is logarithmic in the tree size, which is $|S|$.

4.4 Adaptive Selection of Batch Size via a Multi-Armed Bandit

In this section, we describe how BaVerLy learns the optimal batch size (Line 10). This step provides another advantage of verification of a large set of ϵ -balls: not only BaVerLy can scale the analysis using mini-batch verification but also if the set S is large it can dynamically learn the optimal mini-batch size. In particular, it may begin from larger mini-batches, for the relatively close inputs

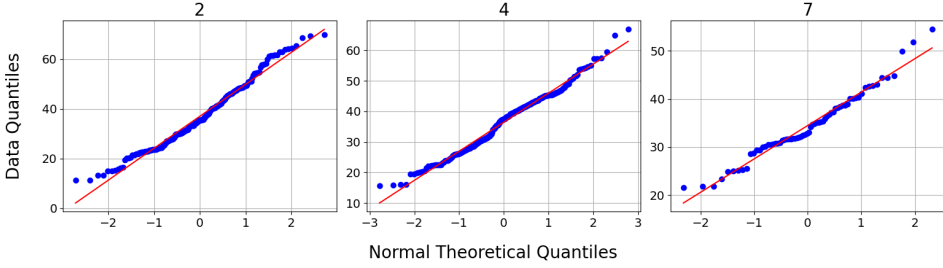


Fig. 7. A quantile-quantile plot comparing the empirical velocity distributions for batch sizes 2, 4, and 7 against the theoretical Gaussian distributions with the same mean and variance.

in S , and as the inputs become farther apart, it can dynamically reduce the mini-batch size. We next describe the mechanism for predicting the batch size via a multi-armed bandit (MAB) agent and how it leverages the verification of previous batches to predict the next batch size.

Multi-armed bandit. The multi-armed bandit (MAB) is a reinforcement learning problem where an agent iteratively selects an arm from a fixed set of arms $\{1, \dots, K\}$. Each arm has an unknown distribution for reward. After selecting an arm, a random reward is sampled from the arm's distribution and added to the agent's total reward. The agent's goal is to maximize their total reward. During the selection process, the agent learns the distributions of the arms. Consequently, they face the known *exploration-exploitation* trade-off: at each iteration the agent can choose the arm with the highest expected reward (*exploitation*) or an arm that may have better rewards (*exploration*). The Thompson Sampling is a strategy for selecting the arms that effectively balances exploration and exploitation [Thompson 1933]. Specifically, we focus on the Gaussian mean-variance bandits (MVTs) algorithm proposed by Zhu and Tan [2020], in which the arms' distributions are Gaussian. In this case, the Thompson Sampling solves the Risk-Averse MAB problem, namely it balances reward maximization and variance minimization (via a risk tolerance factor ρ added to the agent's goal). In each iteration, the agent samples from the arms' distributions and selects the arm that optimizes the mean-variance objective function. Given the reward, it updates the arms' distributions.

MAB for batch sizes. We define an arm for every mini-batch size up to `MAX_BATCH_SIZE`, which is a hyper-parameter. After the MAB agent selects an arm k , BaVerLy constructs a batch of up to that size and verifies it. We define the reward as the velocity of this batch. Recall that velocity is distance divided by time. We define a batch B 's distance as the number of inputs in B whose ϵ -balls are analyzed jointly (i.e., the inputs whose robustness state is updated in Line 30). The batch's time is the total runtime invested in proving the batch, i.e., the bound computation for all $x \in B$ from layer N_1 to layer N_ℓ , plus the time of the batch verification. This time excludes the refinement time (i.e., the time spent to certify individual ϵ -balls who failed during the batch verification), because it is independent of the batch effort, and we account for those failures in the distance calculation. Our MAB leverages MVTs, since in practice the batch velocity of every batch size is approximately Gaussian-distributed. Figure 7 shows a quantile-quantile plot demonstrating that the velocity distributions are close to Gaussian, for different batch sizes, on an MNIST fully connected classifier with five hidden layers of 100 neurons each.

Unifying arms. The higher the number of arms, the longer it takes for MAB to converge to the arms' distributions. To expedite its convergence, we partition the arms into *buckets*. Each bucket consists of values $\{i, \dots, i + \text{BUCKET_SIZE} - 1\}$. If an arm is selected, the maximal value in the bucket is used as the predicted batch size. If a batch of size k is verified, its velocity is used for updating

the distribution of the arm containing this value $k \in \{i, \dots, i + \text{BUCKET_SIZE} - 1\}$. For example, for $\text{MAX_BATCH_SIZE} = 8$ and $\text{BUCKET_SIZE} = 2$, there are four arms: $\{1, 2\}$, $\{3, 4\}$, $\{5, 6\}$, $\{7, 8\}$. If MAB selects the arm $\{5, 6\}$, it recommends using a batch size of 6. If after this recommendation, BaVerLy builds a batch of size 3, this batch's velocity is used to update the distribution of the arm $\{3, 4\}$.

4.5 Choosing The Layer for Batch Verification

We next explain how BaVerLy chooses the layer N_ℓ for the batch verification. Selecting N_ℓ is challenging due to the scalability-precision trade-off introduced by our batch verification, which is faster but adds overapproximation error. Although the batch's input layer N_ℓ does not add overapproximation error (due to our disjunction encoding), the following layers exhibit overapproximation error, because the real-valued bounds of every neuron consider all the batch's ϵ -balls. The further the layer from N_ℓ , the higher the overapproximation error. This may suggest to favor N_ℓ closer to the output layer. However, BaVerLy analyzes each ϵ -ball separately up to layer N_ℓ . Thus, the closer N_ℓ to the output layer the lower the reduction in the overall analysis time compared to one by one verification. This may suggest to favor N_ℓ closer to the input layer.

For convolutional neural networks, we choose N_ℓ as the last convolutional layer (before the fully connected layers). The motivation is that the output of this layer tends to be sufficiently discriminative across different classes. Also, computing the bounds of convolutional layers is relatively fast, since their neurons get as input only part of the neurons in the previous layer.

For fully connected networks, we have not observed a single global layer that is effective for splitting. As known, the goal of neurons in neural networks is to extract from previous neurons the information relevant for the classification. The better their extracted information the tighter the bounds. For some networks, the bounds at earlier layers are sufficiently tight for batch verification. Thus, splitting in an early layer enables BaVerLy to prove robustness. For other networks, splitting in an early layer leads to spurious counterexamples and triggers more refinements. To estimate the best N_ℓ , we rely on initialization via sampling [Bergstra and Bengio 2012]. For each layer l , we estimate its effectiveness by sampling an input from S and verifying its ϵ -ball when $\ell = l$. We define ℓ as the layer with the minimal analysis time. Our estimate considers batches of size $k = 1$ to eliminate the influence of the similarity of the inputs in the batch. Our approach has several advantages. First, it is computationally efficient, since it focuses on batches of size one and relies on the verification of only $L - 1$ ϵ -balls. Second, it advances the task of group verification, since it determines the robustness status of the sampled inputs. Third, it does not rely on an offline mechanism or hyper-parameter tuning, which may not generalize well to an unseen network. Our estimate approach is independent of BaVerLy's analysis and can be improved by other mechanisms, e.g., online learning of automated reasoning strategies for a set of similar problems [Wu et al. 2023].

Algorithm 4 shows how BaVerLy picks the layer to split. It takes as input the network N , the set of inputs S , the class c , the ϵ , and the dictionary `is_robust`. It maintains a dictionary `layers_times` mapping a layer to its analysis time for a single ϵ -ball of a sampled input. For each layer, BaVerLy samples an input x and verifies its ϵ -ball when splitting in this layer (Line 5–Line 6). If a spurious counterexample is discovered, it analyzes the ϵ -ball without splitting (Line 7). Then, it updates the status in `is_robust` and removes x from S . Lastly, it returns the layer with the minimal runtime.

4.6 Complexity Analysis of BaVerLy

In this section, we analyze the asymptotic complexity of our approach. We begin with the analysis time of a single batch verification, followed by the overall complexity analysis of BaVerLy.

Batch verification. The verification of a batch B of size k includes (1) computing the bounds of each input in B up to layer ℓ (Line 13–Line 16), (2) the batch verification (Line 18), and (3) the verification

Algorithm 4: learnSplitLayer($N, S, c, \epsilon, \text{is_robust}$)

```

1 layers_times = []                                // Dictionary mapping layers to runtimes
2 for  $l \in \{1, \dots, L-1\}$  do
3      $x = \text{uniform}(S)$                                 // An input sample
4     start_time = current_time()
5     bounds = MILPBounds( $N_l \circ \dots \circ N_1, x, \epsilon$ )    // Compute bounds up to  $N_l$ 
6     cex = MILPSolve(MILPBatch( $N_L \circ \dots \circ N_{l+1}, \{\text{bounds}[l]\}, c$ ))    // Verify the rest
7     if  $\text{cex} \neq \perp$  then  $\text{cex} = \text{MIPVerify}(N, x, \epsilon, \text{bounds})$     // Refine
8     is_robust[ $x$ ] =  $\text{cex} == \perp$  ? Robust : Non-Robust
9     layers_times[l] = current_time() - start_time
10     $S = S \setminus \{x\}$ 
11  $\ell = \arg \min(\text{layers\_times})$                 // Choose the layer with the shortest runtime
12 return  $\ell, S, \text{is\_robust}$ 

```

of ϵ -balls suspected as not robust (Line 22). The asymptotic time complexity of a MILP is exponential in the number of binary variables. In our setting, this number is the sum of the ReLU neurons, k (for the disjunction, Equation (2a)) and $d_{\text{out}} - 1$ (for checking if class c might not have the maximal score, Equation (2e)). The complexity is thus $T(B) = O(k \cdot 2^{\sum_{i=1}^{\ell} m_i} + (2^{k+d_{\text{out}}+\sum_{i=\ell+1}^L m_i}) + r \cdot 2^{d_{\text{out}}+\sum_{i=1}^L m_i})$, where m_i is the number of ReLU neurons in layer i and r is the number of ϵ -balls suspected as non-robust. We remind that verifying a batch after removing an ϵ -ball that is suspected as non-robust (Line 28) does not incur overhead. For comparison, the asymptotic time complexity of verifying the ϵ -balls one by one with MIPVerify is $O(k \cdot 2^{d_{\text{out}}+\sum_{i=1}^L m_i})$. If $r = 0$, BaVerLy reduces MIPVerify's complexity by a factor of $k \cdot 2^{\sum_{i=1}^{\ell} m_i}$, which is added as an additive term, and multiplies by 2^k (which is independent on the network size).

BaVerLy's complexity. BaVerLy begins by passing the inputs in S through N , computing their activation patterns, and storing them in a dictionary. The time complexity is negligible (compared to our analysis) and the memory complexity is $O(|S|)$. The H-cluster incurs a runtime complexity of $O(|S|^3)$ and the resulting binary tree has a size of $O(2|S|)$. The time complexity of operations on this tree is $O(\log|S|)$ on average and $O(|S|)$ in the worst-case. The time complexity of the operations on the MAB agent depends on the number of arms. Since it is a very small number, the time complexity is $O(1)$. The dominant factor in BaVerLy's runtime is the verification (known to be NP-hard [Katz et al. 2017]). Let B_1, \dots, B_p be all batches in BaVerLy's run, their time complexity is $\sum_{i=1}^p T(B_i)$.

5 Evaluation

In this section, we present the experimental results of our approach. We begin by discussing our implementation and evaluation setup. We then describe our experiments showing that: (1) BaVerLy expedites the approach of verifying local robustness ϵ -ball by ϵ -ball on average by 2.3x and up to 4.1x, (2) BaVerLy determines robustness for all inputs, whereas shared certification verification [Fischer et al. 2023] determines robustness only for 63% of the inputs and shows a lower maximal speedup (1.21x), (3) the more inputs in the set, the higher the speedup in the analysis time per input: 100 inputs already enable a 2x speedup, and (4) learning the optimal mini-batch size by our multi-armed bandit boosts the performance of BaVerLy by 2.5x.

Implementation. We implemented BaVerLy in the Julia programming language (version 1.11.1), as a module wrapper for MIPVerify [Tjeng et al. 2019]. We extended MIPVerify to support our

refinement, where some of the bounds are given and need not be computed. We used Gurobi (version 12.0.1) [Gurobi Optimization, LLC 2023] as the MILP solver. For the batch verification (Lines 18 and 28) and MIPVerify (Line 22), we set Gurobi's MIPFocus flag to 1 to guide it to focus on finding a feasible solution rather than the optimal one, and we set its SolutionLimit flag to 1 so that Gurobi would terminate when finding the first feasible solution. These adaptations fit our setting since our goal is to determine whether an ϵ -ball is robust or not, which translates to determining whether these MILPs are feasible or not.

Evaluation setup. We conducted our experiments on Ubuntu 20.04.2 LTS OS on a dual AMD EPYC 7742 64-Core Processor server with 1TB RAM and 128 cores. We compared BaVerLy to MIPVerify [Tjeng et al. 2019] on which we build our MILP encoding. Given a set of inputs, MIPVerify verifies their ϵ -balls one by one. Its analysis time is the total analysis time over all ϵ -balls. Gurobi is used by BaVerLy and MIPVerify and it is parallelized over 8 threads and has the same flags' values. We evaluated BaVerLy on the MNIST dataset [Deng 2012], consisting of 28×28 grayscale images of handwritten digits, and the CIFAR-10 dataset [Krizhevsky 2012], consisting of 32×32 RGB images representing ten classes of common animals and vehicles. For MNIST, we adopted four network architectures from Mirman et al. [2018]: the convolutional networks ConvSmall and ConvMed, and the fully connected (FC) networks 5×100 and 6×100 . They contain 3,604, 5,704, 500, and 600 ReLU neurons, respectively. ConvSmall has two convolutional layers with ReLU, a fully connected layer of 100 ReLU neurons, and a fully connected layer with ten neurons for assigning the scores to each class. ConvMed is similar to ConvSmall but has slightly different padding and its first fully connected has 1000 ReLU neurons. The 5×100 has five hidden layers and the 6×100 network has six hidden layers, each with 100 ReLU neurons. We trained our MNIST models using the PGD adversarial defense [Madry et al. 2018] with a perturbation limit of $\epsilon = 0.2$, for the convolutional networks, and a limit of $\epsilon = 0.1$, for the fully connected networks. For CIFAR-10, we trained a ConvMed network with 7,144 ReLU neurons using PGD with $\epsilon = 0.001$. We incorporated techniques to improve generalization and stability, including L_1 regularization, adaptive learning rate scheduling, and Xavier uniform weight initialization. Training was performed using Adam [Kingma and Ba 2014], for 6 epochs with a batch size of 128. Additionally, we executed standard scaling to transform pixel values for improved performance and added a corresponding normalization layer when loading the models. The natural accuracy of the networks is 96% for MNIST ConvMed, 93% for ConvSmall and 5×100 , 91% for 6×100 and 48% for CIFAR-10 ConvMed (similar to the accuracies of the networks evaluated by Fischer et al. [2023]). For the convolutional networks, the split layer ℓ is the last convolutional layer. For the fully connected networks, ℓ is chosen by `learnSplitLayer`. The maximum batch size is `MAX_BATCH_SIZE` = 4 for the ConvMed networks and `MAX_BATCH_SIZE` = 8 for the rest. The bucket size for unifying batch sizes is `BUCKET_SIZE` = 2 and our MAB's ρ is 100.

5.1 Performance Analysis

In this section, we evaluate BaVerLy's effectiveness in group verification and compare to MIPVerify.

Group verification on large sets. We begin with an experiment on large sets of inputs. In this experiment, we consider all networks. For each, we run BaVerLy on sets S of the first 100 inputs of the same class (for several classes) and different values of ϵ . We compare its analysis time to MIPVerify's analysis time. We remind that both verifiers are complete (i.e., correctly determine whether an ϵ -ball is robust or not). Table 1 shows the certification rate, the total analysis time of both approaches and BaVerLy's speedup. The certification rate is the number of inputs whose ϵ -balls are verified as robust (by both approaches) divided by the number of correctly classified inputs. Results show that BaVerLy's speedup is 2.3x on average and up to 4.1x. The highest speedup is obtained on MNIST 5×100 , where BaVerLy reduces the analysis time from 24 hours to 6 hours.

Table 1. BaVerLy vs. MIPVerify over different networks and ϵ on sets with 100 inputs of the same class.

Dataset	Network	ϵ	c	Cert. Rate	MIPVerify [hours]	BaVerLy [hours]	Speedup
MNIST	ConvMed	0.03	0	99/99	27.22	8.00	3.4
			1	100/100	36.42	14.08	2.6
			2	96/98	27.61	9.99	2.7
			3	95/98	27.08	9.82	2.7
	ConvSmall	0.05	0	95/98	2.04	1.36	1.5
			1	98/100	2.30	0.96	2.4
			2	78/91	1.90	1.33	1.4
			3	78/89	2.07	1.38	1.5
	5×100	0.03	0	96/98	13.87	4.92	2.8
			1	96/100	23.63	5.75	4.1
			2	79/90	8.80	5.70	1.5
			3	78/93	10.88	6.23	1.7
	6×100	0.03	0	88/97	11.50	5.46	2.1
			1	92/100	33.85	12.79	2.6
			2	75/88	15.10	8.77	1.7
			3	94/98	14.37	5.43	2.6
CIFAR-10	ConvMed	0.001	0	37/39	2.64	1.07	2.4
			1	61/62	4.18	1.97	2.1
			2	31/31	1.74	0.87	2.0
			3	39/41	2.89	1.60	1.8

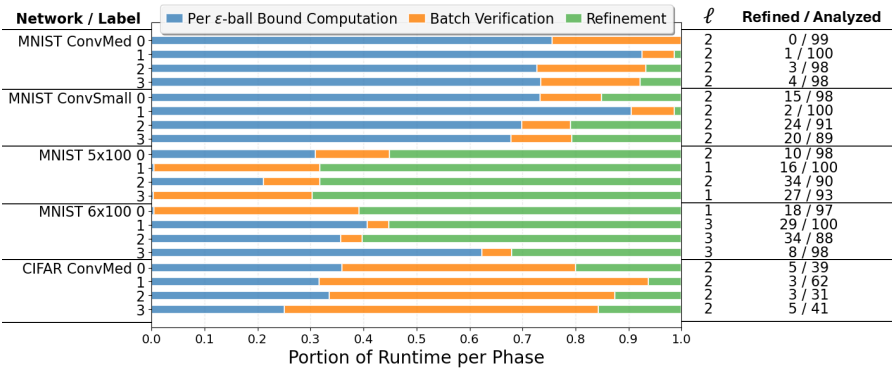


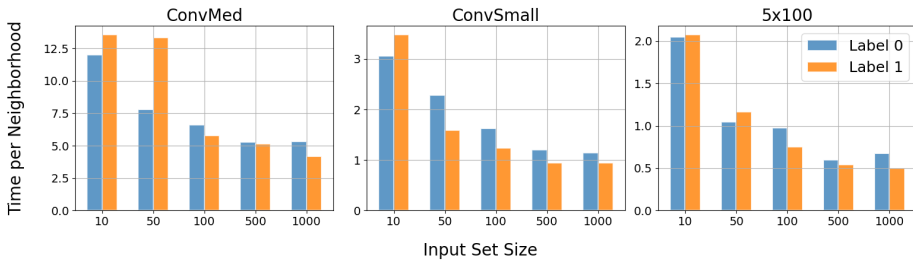
Fig. 8. Runtime breakdown and refinement frequency for the experiments in Table 1.

Figure 8 shows the runtime breakdown and refinement frequency. It shows that most ϵ -balls are verified within a batch. The MNIST convolutional networks have the shortest refinement phase, while the fully connected networks have the shortest bound computation and longest refinement.

Group verification on small sets. Next, we evaluate BaVerLy on small sets of inputs. We consider different networks and ϵ values. For each, we ran each approach on the first 100 test set images, consisting of different classes (BaVerLy ran separately on each class). For such sets, BaVerLy almost

Table 2. BaVerLy vs. MIPVerify over different networks and ϵ on sets with the first 100 test set images.

Dataset	Network	ϵ	Cert. Rate	MIPVerify [hours]	BaVerLy [hours]	Speedup
MNIST	ConvMed	0.03	98/100	31.98	11.96	2.6
	ConvSmall	0.1	69/88	27.86	19.33	1.4
	5×100	0.03	84/92	11.71	4.85	2.4
	6×100	0.03	80/92	18.85	11.80	1.6
CIFAR-10	ConvMed	0.001	49/50	4.03	2.43	1.6

Fig. 9. Verification time per ϵ -ball (in seconds) for different set sizes and the first two classes of MNIST.

does not benefit from learning the optimal mini-batch size and does not benefit from clustering inputs with similar computations of the networks. Thus, this experiment is challenging for BaVerLy. Table 2 shows the total analysis time (over all classes), for BaVerLy and MIPVerify. It further shows the certification rate and BaVerLy's speedup. BaVerLy achieves up to a 2.6x speedup compared to MIPVerify. MNIST ConvMed has the best speedup (and the highest certification rate, 98/100).

Shared certification. We next discuss the empirical differences between BaVerLy and shared certification [Fischer et al. 2023]. Shared certification relies on preprocessing (which takes multiple hours) for generating templates which can expedite the analysis on unseen ϵ -balls. It focuses on incomplete verification, i.e., the robustness of ϵ -balls can remain unknown. For example, in our experiment over sets with the first 100 test set images, for ConvSmall and $\epsilon = 0.1$, BaVerLy precisely determines the certification rate, which is 69/88 (78%). However, verification that relies on the DeepZ abstract domain [Singh et al. 2018], like shared certification, can only prove that 56 ϵ -balls are robust (certification rate of 63%). Further, the highest speedup of shared certification over DeepZ (on which it builds) is 1.21x, while BaVerLy's speedup over MIPVerify is at least 1.4x and up to 2.6x.

Analysis time per ϵ -ball. Next, we show that as the input set size increases, the average analysis time per ϵ -ball decreases. This experiment shows the effectiveness of clustering inputs in batches, especially for larger sets where the likelihood of finding inputs with similar network computations increases. Additionally, the larger the input set, the better our MAB agent in predicting optimal mini-batch sizes. In this experiment, we focus on the MNIST networks and $\epsilon = 0.00001$. We consider the first two classes of MNIST and for each we construct input sets of different sizes from 10 to 1000 (for class 0, up to 980, which is its test set's size). Figure 9 shows the average analysis time per ϵ -ball as a function of the size of the set. Results show that, compared to the set with 10 inputs, BaVerLy obtains up to a 4x speedup for 5×100, up to a 3.5x speedup for ConvSmall and up to a

Table 3. BaVerLy vs. a variant that randomly chooses the mini-batch sizes. The times of the random variant are averaged over three repetitions. The input set is the full test set of MNIST's first class.

Dataset	Network	ϵ	BaVerLy [m]	BaVerLy w/ random size [m]	Speedup
MNIST	ConvMed	0.00001	142.55	639.73	4.5
	ConvSmall	0.00001	35.79	53.64	1.5
	5×100	0.00001	11.02	18.15	1.6

Table 4. BaVerLy vs. a variant with a given split layer ℓ , on sets of 50 MNIST images of the same class.

Network	ϵ	c	BaVerLy [h] (ℓ)	$\ell = 1$ [h]	$\ell = 2$ [h]	$\ell = 3$ [h]	$\ell = 4$ [h]	$\ell = 5$ [h]
5×100	0.03	0	2.98 (1)	2.75	2.94	3.60	4.62	N/A
		1	2.92 (1)	2.71	4.34	5.36	7.03	N/A
		2	2.93 (1)	2.89	3.33	3.35	3.63	N/A
6×100	0.03	0	2.83 (1)	2.75	2.72	3.17	3.59	4.35
		1	6.28 (3)	7.99	7.16	6.23	8.56	10.69
		2	5.08 (1)	5.02	4.76	4.41	4.59	5.38

3.3x speedup for ConvMed, for the set with 1000 inputs. In fact, for all networks, a 2x speedup is observed already for 100 inputs. This shows the effectiveness of our batches and learning.

5.2 Ablation Study

In this section, we show the effectiveness of BaVerLy's components.

MAB effectiveness. We study the importance of learning the mini-batch size using a multi-armed bandit (MAB). We consider a variant that randomly selects the mini-batch sizes. We set MAX_BATCH_SIZE = 16 to make the learning more challenging for our MAB agent. We run both approaches on the MNIST classifiers, where the input set is 980 test inputs of class 0. Table 3 shows the analysis time and the speedup of BaVerLy. Results show that our MAB accelerates BaVerLy's verification time by 4.5x for ConvMed, by 1.5x for ConvSmall and by 1.6x for 5×100.

Split layer. We next show the effectiveness of BaVerLy in learning the split layer N_ℓ (Algorithm 4). We compare to a variant that fixes ℓ . We evaluate on MNIST 5×100 and 6×100 and three input sets, each containing 50 images of the same class (0, 1 or 2). Table 4 shows the verification time in hours and the layer that BaVerLy chose. The results show the importance of selecting a good ℓ and that the optimal ℓ varies between the networks and input sets. The results also show that BaVerLy selects the optimal split layer for most networks and classes with a overhead of 7.2 minutes, on average. Even when a sub-optimal ℓ is selected, the overhead does not exceed 40.2 minutes.

Similarity metrics. We next show the effectiveness of the activation pattern similarity. We compare BaVerLy to variants that replace the activation pattern similarity with SSIM [Wang et al. 2004] and LPIPS [Zhang et al. 2018] (using AlexNet activations). We evaluate on CIFAR-10 ConvMed with an input set containing 100 inputs of the same class, for different classes and $\epsilon = 0.001$. Figure 10a shows the analysis time. It shows that the activation pattern similarity is better by 1.13x, on average.

Disjunction. We next show the importance of our disjunction encoding (Equation (1)). We evaluate a variant that abstracts ϵ -balls in the input layer ($\ell = 0$) with the minimal bounding box. It does not refine ϵ -balls, because it cannot easily identify a suspect non-robust ϵ -ball. We ran this variant

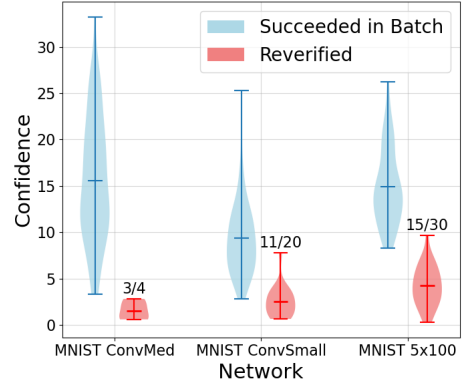
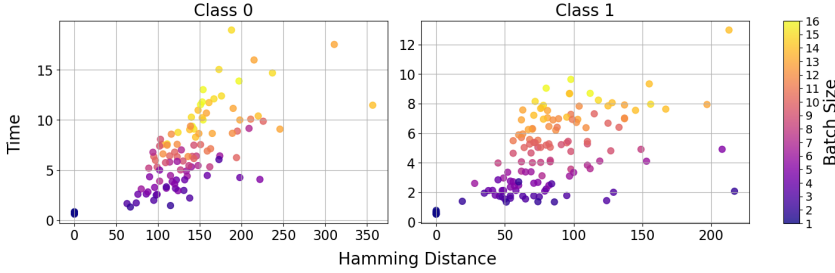
(a) Analysis time on CIFAR-10 ConvMed, $\epsilon = 0.001$.

c	AP [m]	SSIM [m]	LPIPS [m]
0	64.3	72.3	69.3
1	118.6	135.3	154.8
2	52.2	57.4	58.2

(b) BaVerLy without disjunction encoding, $\ell = 0$, and a 10 hour limit on MNIST ConvSmall, $\epsilon = 0.05$.

Similarity metric	#Analyzed	#Robust
AP	75	1
SSIM	74	2
LPIPS	76	2

Fig. 10. BaVerLy with different similarity metrics.

Fig. 11. The confidence of ϵ -balls verified within a batch and ϵ -balls that required a separate analysis.Fig. 12. The analysis time of BaVerLy (in minutes) for different batches on MNIST ConvSmall, $\epsilon = 0.05$.

with the three similarity metrics, on MNIST ConvSmall, $\epsilon = 0.05$, and an input set consisting of 100 images of class 0, and a 10 hour limit. Figure 10b reports the number of analyzed inputs (whose ϵ -ball is included in a batch) and the number of ϵ -balls that were proven robust. On average, 25% ϵ -balls could not be analyzed within 10 hours, and at most two ϵ -balls were proven robust. In contrast, BaVerLy determines robustness for all 98 ϵ -balls within 1.36 hours (Table 1).

Error case analysis. We next show an empirical difference between ϵ -balls that were proven within a batch and those that were refined. The difference is the network's *confidence* in their central input, i.e., the gap between the highest and second highest scores. Figure 11 shows a violin plot over the confidence distribution of MNIST inputs of class 3, for different networks, for ϵ -balls that were proven in a batch (in blue) and refined (in red). Above the red violin, we show the number of non-robust ϵ -balls and the number of refined (suspected non-robust) ϵ -balls. The figure shows that the confidence is lower for refined ϵ -balls and that at least half of them are indeed non-robust.

Batch size vs. Hamming distance. BaVerLy constructs a batch by determining its size, with the MAB agent, and then choosing the most similar inputs. An alternative is to choose the most similar inputs, with a MAB agent that selects the maximal pairwise distance, and then determine the batch size. We next show that this alternative is ineffective since the analysis time is not correlated to the maximal pairwise Hamming distance. We consider a variant that randomly selects a mini-batch size (up to 16) and then constructs a batch of exactly this size (if possible), to draw balanced statistics for all sizes. We evaluate on MNIST ConvSmall, $\epsilon = 0.05$, and input sets consisting of all 980 and

1135 test set inputs of class 0 and 1. Figure 12 shows the analysis time of each batch as a function of the maximal pairwise Hamming distance. Each batch is colored by its size. The results show that the analysis time of every distance has a wide range, whereas batches of the same size have similar analysis time. For example, for class 1, the analysis of batches with distance 70 takes 1-9 minutes, whereas for batches of size 8 it takes 4-6 minutes. That is, the analysis time is related to the batch size (we remind that batches are evaluated by their velocity, which depends on the analysis time).

6 Related Work

In this section, we discuss related work for boosting neural network verification.

Boosting by previous analysis. Several neural network verifiers boost their analysis by relying on previous analysis. Verifiers targeting continuous verification (incremental verification) leverage the intermediate analysis results of a network to expedite the analysis of its variants obtained by further training or fine-tuning [Cheng and Yan 2021; Ugare et al. 2023; Wei and Liu 2023]. FANC proposes *proof sharing* for generating intermediate layer templates that capture the property being verified and adapt them for similar networks to expedite their verification [Ugare et al. 2022]. Shared certifications extend this concept to boost the analysis on unseen ϵ -balls by generating abstract templates of intermediate analysis results during preprocessing [Fischer et al. 2022]. DeepAbstract analyzes a set of inputs during preprocessing to identify similar neurons by their activation patterns [Ashok et al. 2020]. These neurons are abstracted when verifying ϵ -balls, thereby boosting their analysis. Similarly, AccMILP analyzes a set of inputs during preprocessing to identify neurons with less impact on the network's accuracy [Zheng et al. 2025]. Accordingly, the verification of an ϵ -ball performs linear relaxation only to these neurons.

Boosting by learning. Several neural network verifiers employ learning to expedite their analysis. Wan et al. [2020] boost existing verifiers by learning a prioritization over classes by their likelihood to be the classes of adversarial examples. VeeP expedites local robustness verification of a network classifier in semantic feature neighborhoods by relying on active learning to partition the verification process into smaller steps [Kabaha and Drachler-Cohen 2022]. Similarly to BaVerLy, it predicts the next step by computing the analyzer's velocity on previous steps. Learning has also been proposed to expedite the analysis of verifiers in other domains. Brázdil et al. [2015] rely on machine learning to improve the verification of Markov decision processes (MDPs), efficiently analyzing probabilistic reachability and temporal properties without exhaustively exploring the entire state space. Dimitrakopoulos et al. [2023] reduce hardware verification efforts using a multi-armed bandit to automatically select the most promising test sequences. Wu et al. [2023] learn optimal strategies for expediting the automated reasoning of a set of similar problems. They sample problem candidates, train a cost model to predict the runtime of a strategy for a given problem, and use it to dynamically select efficient strategies for future problems.

7 Conclusion

We present BaVerLy, a verifier that analyzes the robustness of a neural network in a set of L_∞ ϵ -balls. BaVerLy is sound and complete. The key idea is to identify the ϵ -balls for which the network has similar computations and group their analysis in a mini-batch. BaVerLy relies on a multi-armed bandit to predict the optimal mini-batch size. BaVerLy begins the batch analysis in a middle layer and encodes the inputs to this layer precisely. This encoding also enables an effective refinement. We evaluated BaVerLy on fully connected and convolutional networks for MNIST and CIFAR-10. Experimental results show that BaVerLy scales the verification on average by 2.3x and up to 4.1x, reducing verification time from 13 hours to 5 hours, on average. Our results also show the importance of learning the optimal batch sizes: it scales BaVerLy by 2.5x.

Acknowledgements

We thank Yuval Shapira for his invaluable insights and the anonymous reviewers for their feedback. This research was supported by the Israel Science Foundation (grant No. 2607/25).

Data-Availability Statement

The data and source code that support the findings of this study are openly available. The reproducible artifact associated with this paper is described in [Tzour-Shaday and Drachsler-Cohen \[2025\]](#), and the implementation and experimental scripts can be accessed and reused at <https://github.com/Saarts21/BaVerLy>.

References

- Marios Anthimopoulos, Stergios Christodoulidis, Lukas Ebner, Andreas Christe, and Stavroula Mougiakakou. 2016. Lung Pattern Classification for Interstitial Lung Diseases Using a Deep Convolutional Neural Network. *IEEE Transactions on Medical Imaging* 35, 5 (2016), 1207–1216. <https://doi.org/10.1109/TMI.2016.2535865>
- Pranav Ashok, Vahid Hashemi, Jan Křetínský, and Stefanie Mohr. 2020. DeepAbstract: Neural Network Abstraction for Accelerating Verification. In *Automated Technology for Verification and Analysis*, Dang Van Hung and Oleg Sokolsky (Eds.). Springer International Publishing, Cham, 92–107. https://doi.org/10.1007/978-3-030-59152-6_5
- Riadh Ayachi, Mouna Afif, Yahia Said, and Mohamed Atri. 2020. Traffic Signs Detection for Real-World Application of an Advanced Driving Assisting System Using Deep Learning. *Neural Processing Letters* 51 (02 2020). <https://doi.org/10.1007/s11063-019-10115-8>
- Mrinal R. Bachute and Javed M. Subhedar. 2021. Autonomous Driving Architectures: Insights of Machine Learning and Deep Learning Algorithms. *Machine Learning with Applications* 6 (2021), 100164. <https://doi.org/10.1016/j.mlwa.2021.100164>
- Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural Machine Translation by Jointly Learning to Align and Translate. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). <http://arxiv.org/abs/1409.0473>
- Stanley Bak and Parasara Duggirala. 2017. Simulation-Equivalent Reachability of Large Linear Systems with Inputs. 401–420. https://doi.org/10.1007/978-3-319-63387-9_20
- Mislav Balunovic, Maximilian Baader, Gagandeep Singh, Timon Gehr, and Martin Vechev. 2019. Certifying Geometric Robustness of Neural Networks. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2019/file/f7fa6aca028e7ff4ef62d75ed025fe76-Paper.pdf
- James Bergstra and Yoshua Bengio. 2012. Random search for hyper-parameter optimization. *J. Mach. Learn. Res.* 13, null (Feb. 2012), 281–305. <https://doi.org/10.5555/2503308.2188395>
- Mariusz Bojarski, David W. del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. 2016. End to End Learning for Self-Driving Cars. *ArXiv abs/1604.07316* (2016). <https://api.semanticscholar.org/CorpusID:15780954>
- Tomáš Brázdil, Krishnendu Chatterjee, Martin Chmelík, Vojtěch Forejt, Jan Křetínský, Marta Kwiatkowska, David Parker, and Mateusz Ujma. 2015. Verification of Markov Decision Processes using Learning Algorithms. *arXiv:1402.2967* [cs.LO] <https://arxiv.org/abs/1402.2967>
- Chih-Hong Cheng and Rongjie Yan. 2021. Continuous Safety Verification of Neural Networks. (2021), 1478–1483. <https://doi.org/10.23919/DAT51398.2021.9473994>
- Francesco Croce and Matthias Hein. 2019. Sparse and Imperceptible Adversarial Attacks. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*.
- Li Deng. 2012. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine* 29, 6 (2012), 141–142. <https://doi.org/10.1109/MSP.2012.2211477>
- G. Dimitrakopoulos, E. Kallitsounakis, Z. Takakis, A. Stefanidis, and C. Nicopoulos. 2023. Multi-Armed Bandits for Autonomous Test Application in RISC-V Processor Verification. In *2023 12th International Conference on Modern Circuits and Systems Technologies (MOCASST)*. 1–5. <https://doi.org/10.1109/MOCASST57943.2023.10176659>
- Andre Esteva, Brett Kuprel, Roberto Novoa, Justin Ko, Susan Swetter, Helen Blau, and Sebastian Thrun. 2017. Dermatologist-level classification of skin cancer with deep neural networks. *Nature* 542 (01 2017). <https://doi.org/10.1038/nature21056>
- European Commission. 2020. *White Paper on Artificial Intelligence - A European approach to excellence and trust*. European Commission. <https://digital-strategy.ec.europa.eu/en/consultations/white-paper-artificial-intelligence-european-approach-excellence-and-trust>
- Claudio Ferrari, Mark Niklas Muller, Nikola Jovanovic, and Martin Vechev. 2022. Complete Verification via Multi-Neuron Relaxation Guided Branch-and-Bound. *arXiv:2205.00263* [cs.LG] <https://arxiv.org/abs/2205.00263>

- Marc Fischer, Christian Sprecher, Dimitar I. Dimitrov, Gagandeep Singh, and Martin Vechev. 2022. Shared Certificates for Neural Network Verification. , 127–148 pages. https://doi.org/10.1007/978-3-031-13185-1_7
- Marc Fischer, Christian Sprecher, Dimitar I. Dimitrov, Gagandeep Singh, and Martin Vechev. 2023. Shared Certificates for Neural Network Verification. arXiv:2109.00542 [cs.LG] <https://arxiv.org/abs/2109.00542>
- Timon Gehr, Matthew Mirman, Dana Drachler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin Vechev. 2018. AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation. In *2018 IEEE Symposium on Security and Privacy (SP)*. 3–18. <https://doi.org/10.1109/SP.2018.00058>
- Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. 2015. Explaining and Harnessing Adversarial Examples. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). <http://arxiv.org/abs/1412.6572>
- Alex Graves and Navdeep Jaitly. 2014. Towards End-To-End Speech Recognition with Recurrent Neural Networks. In *Proceedings of the 31st International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 32)*, Eric P. Xing and Tony Jebara (Eds.). PMLR, Beijing, China, 1764–1772. <https://proceedings.mlr.press/v32/graves14.html>
- Chuan Guo, Jacob Gardner, Yurong You, Andrew Gordon Wilson, and Kilian Weinberger. 2019. Simple Black-box Adversarial Attacks. In *Proceedings of the 36th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, 2484–2493. <https://proceedings.mlr.press/v97/guo19a.html>
- Gurobi Optimization, LLC. 2023. Gurobi Optimizer Reference Manual. <https://www.gurobi.com>
- Yujia Huang, Huan Zhang, Yuan Yuan Shi, J Zico Kolter, and Anima Anandkumar. 2021. Training Certifiably Robust Neural Networks with Efficient Local Lipschitz Bounds. arXiv:2111.01395 [cs.LG] <https://arxiv.org/abs/2111.01395>
- Andrew Ilyas, Logan Engstrom, Anish Athalye, and Jessy Lin. 2018. Black-box Adversarial Attacks with Limited Queries and Information. In *Proceedings of the 35th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 80)*, Jennifer Dy and Andreas Krause (Eds.). PMLR, 2137–2146. <https://proceedings.mlr.press/v80/ilyas18a.html>
- Omar Javed and Mubarak Shah. 2002. Tracking and Object Classification for Automated Surveillance. In *Computer Vision – ECCV 2002*, Anders Heyden, Gunnar Sparr, Mads Nielsen, and Peter Johansen (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 343–357.
- Anan Kabaha and Dana Drachler-Cohen. 2022. Boosting Robustness Verification of Semantic Feature Neighborhoods. In *Static Analysis: 29th International Symposium SAS (Auckland, New Zealand)*. Springer-Verlag, Berlin, Heidelberg, 299–324. https://doi.org/10.1007/978-3-031-22308-2_14
- Anan Kabaha and Dana Drachler-Cohen. 2024. Verification of Neural Networks’ Global Robustness. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 130 (April 2024), 30 pages. <https://doi.org/10.1145/3649847>
- Fazle Karim, Somshubra Majumdar, and Houshang Darabi. 2021. Adversarial Attacks on Time Series. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 43, 10 (2021), 3309–3320. <https://doi.org/10.1109/TPAMI.2020.2986319>
- Guy Katz, Clark Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. 2017. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In *Computer Aided Verification*, Rupak Majumdar and Viktor Kunčák (Eds.). Springer International Publishing, Cham, 97–117.
- Guy Katz, Derek A. Huang, Duligur Ibeling, Kyle Julian, Christopher Lazarus, Rachel Lim, Parth Shah, Shantanu Thakoor, Haoze Wu, Aleksandar Zeljić, David L. Dill, Mykel J. Kochenderfer, and Clark Barrett. 2019. The Marabou Framework for Verification and Analysis of Deep Neural Networks. In *Computer Aided Verification*, Isil Dillig and Serdar Tasiran (Eds.). Springer International Publishing, Cham, 443–452. https://doi.org/10.1007/978-3-030-25540-4_26
- Diederik Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. *International Conference on Learning Representations* (12 2014).
- Alex Krizhevsky. 2012. Learning Multiple Layers of Features from Tiny Images. *University of Toronto* (05 2012).
- Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems*, F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger (Eds.), Vol. 25. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf
- Klas Leino, Zifan Wang, and Matt Fredrikson. 2021. Globally-Robust Neural Networks. arXiv:2102.08452 [cs.LG] <https://arxiv.org/abs/2102.08452>
- Diego Manzananas Lopez, Sung Woo Choi, Hoang-Dung Tran, and Taylor T. Johnson. 2023. NNV 2.0: The Neural Network Verification Tool. In *Computer Aided Verification*, Constantin Enea and Akash Lal (Eds.). Springer Nature Switzerland, Cham, 397–412.
- J MacQueen. 1967. Some methods for classification and analysis of multivariate observations. In *Proceedings of 5-th Berkeley Symposium on Mathematical Statistics and Probability/University of California Press*.
- Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. 2018. Towards Deep Learning Models Resistant to Adversarial Attacks. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver*,

- BC, Canada, April 30 - May 3, 2018, *Conference Track Proceedings*. OpenReview.net. <https://openreview.net/forum?id=rJzIBfZAb>
- Matthew Mirman, Timon Gehr, and Martin Vechev. 2018. Differentiable Abstract Interpretation for Provably Robust Neural Networks. In *International Conference on Machine Learning*. 3575–3583. <https://files.sri.inf.ethz.ch/website/papers/icml18-diffai.pdf>
- Jeet Mohapatra, Tsui-Wei Weng, Pin-Yu Chen, Sijia Liu, and Luca Daniel. 2020. Towards Verifying Robustness of Neural Networks Against A Family of Semantic Perturbations. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.
- Christoph Müller, François Serre, Gagandeep Singh, Markus Püschel, and Martin Vechev. 2021. Scaling Polyhedral Neural Network Verification on GPUs. arXiv:2007.10868 [cs.LG] <https://arxiv.org/abs/2007.10868>
- Matan Ostrovsky, Clark Barrett, and Guy Katz. 2022. An Abstraction-Refinement Approach to Verifying Convolutional Neural Networks. In *Automated Technology for Verification and Analysis*, Ahmed Bouajjani, Lukáš Holík, and Zhilin Wu (Eds.). Springer International Publishing, Cham, 391–396. https://doi.org/10.1007/978-3-031-19992-9_25
- Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. 2016. You Only Look Once: Unified, Real-Time Object Detection. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*. IEEE Computer Society, 779–788. <https://doi.org/10.1109/CVPR.2016.91>
- Yuval Shapira, Eran Avneri, and Dana Drachslers-Cohen. 2023. Deep Learning Robustness Verification for Few-Pixel Attacks. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 90 (apr 2023), 28 pages. <https://doi.org/10.1145/3586042>
- Yuval Shapira, Naor Wiesel, Shahar Shabelman, and Dana Drachslers-Cohen. 2024. Boosting Few-Pixel Robustness Verification via Covering Verification Designs. In *Computer Aided Verification: 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24–27, 2024, Proceedings, Part II* (Montreal, QC, Canada). Springer-Verlag, Berlin, Heidelberg, 377–400. https://doi.org/10.1007/978-3-031-65630-9_19
- Gagandeep Singh, Timon Gehr, Matthew Mirman, Markus Püschel, and Martin Vechev. 2018. Fast and Effective Robustness Certification. In *International Conference on Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.), Vol. 31. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2018/file/f2f446980d8e971ef3da97af089481c3-Paper.pdf
- Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin Vechev. 2019a. An abstract domain for certifying neural networks. *Proc. ACM Program. Lang.* 3, POPL, Article 41 (jan 2019), 30 pages. <https://doi.org/10.1145/3290354>
- Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin Vechev. 2019b. Robustness Certification with Refinement. In *International Conference on Learning Representations*. <https://openreview.net/forum?id=HJgeEh09KQ>
- Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to sequence learning with neural networks. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2* (Montreal, Canada) (NIPS'14). MIT Press, Cambridge, MA, USA, 3104–3112.
- Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. 2013. Intriguing properties of neural networks. (12 2013).
- William R. Thompson. 1933. On the Likelihood that One Unknown Probability Exceeds Another in View of the Evidence of Two Samples. *Biometrika* 25, 3/4 (1933), 285–294. <http://www.jstor.org/stable/2332286>
- Naftali Tishby and Noga Zaslavsky. 2015. Deep Learning and the Information Bottleneck Principle. *2015 IEEE Information Theory Workshop, ITW 2015* (03 2015). <https://doi.org/10.1109/ITW.2015.7133169>
- Vincent Tjeng, Kai Xiao, and Russ Tedrake. 2019. Evaluating Robustness of Neural Networks with Mixed Integer Programming. *ICLR* (2019).
- Hoang-Dung Tran, Stanley Bak, Weiming Xiang, and Taylor T. Johnson. 2020. Verification of Deep Convolutional Neural Networks Using ImageStars. In *Computer Aided Verification*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer International Publishing, Cham, 18–42. https://doi.org/10.1007/978-3-030-53288-8_2
- Hoang-Dung Tran, Diago Manzananas Lopez, Patrick Musau, Xiaodong Yang, Luan Viet Nguyen, Weiming Xiang, and Taylor T. Johnson. 2019. Star-Based Reachability Analysis of Deep Neural Networks. In *Formal Methods – The Next 30 Years*, Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira (Eds.). Springer International Publishing, Cham, 670–686. https://doi.org/10.1007/978-3-030-30942-8_39
- Saar Tzour-Shaday and Dana Drachslers-Cohen. 2025. Reproduction Package for Article ‘Mini-Batch Robustness Verification of Deep Neural Networks’. Zenodo. <https://doi.org/10.5281/zenodo.16892960>
- Shubham Ugare, Debangshu Banerjee, Sasa Misailovic, and Gagandeep Singh. 2023. Incremental Verification of Neural Networks. *Proceedings of the ACM on Programming Languages* 7, PLDI (June 2023), 1920–1945. <https://doi.org/10.1145/3591299>
- Shubham Ugare, Gagandeep Singh, and Sasa Misailovic. 2022. Proof transfer for fast certification of multiple approximate neural networks. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 75 (April 2022), 29 pages. <https://doi.org/10.1145/3527319>
- Wenjie Wan, Zhaodi Zhang, Yiwei Zhu, Min Zhang, and Fu Song. 2020. Accelerating Robustness Verification of Deep Neural Networks Guided by Target Labels. *ArXiv abs/2007.08520* (2020).

- Fu Wang, Peipei Xu, Wenjie Ruan, and Xiaowei Huang. 2023. Towards verifying the geometric robustness of large-scale neural networks. In *Proceedings of the Thirty-Seventh AAAI Conference on Artificial Intelligence and Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence and Thirteenth Symposium on Educational Advances in Artificial Intelligence (AAAI'23/IAAI'23/EAAI'23)*. AAAI Press, Article 1704, 9 pages. <https://doi.org/10.1609/aaai.v37i12.26773>
- Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. 2018a. Efficient Formal Safety Analysis of Neural Networks. , 6369–6379 pages. <https://proceedings.neurips.cc/paper/2018/hash/2ecd2bd94734e5dd392d8678bc64cdab-Abstract.html>
- Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. 2018b. Formal security analysis of neural networks using symbolic intervals. In *Proceedings of the 27th USENIX Conference on Security Symposium* (Baltimore, MD, USA) (*SEC'18*). USENIX Association, USA, 1599–1614.
- Shiqi Wang, Huan Zhang, Kaidi Xu, Xue Lin, Suman Jana, Cho-Jui Hsieh, and Zico Kolter. 2021. Beta-CROWN: efficient bound propagation with per-neuron split constraints for neural network robustness verification. In *Proceedings of the 35th International Conference on Neural Information Processing Systems (NIPS '21)*. Curran Associates Inc., Red Hook, NY, USA, Article 2289, 13 pages.
- Zhou Wang, A.C. Bovik, H.R. Sheikh, and E.P. Simoncelli. 2004. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing* 13, 4 (2004), 600–612. <https://doi.org/10.1109/TIP.2003.819861>
- Jr. Ward, Joe H. 1963. Hierarchical Grouping to Optimize an Objective Function. *J. Amer. Statist. Assoc.* 58, 301 (1963), 236–244. <https://doi.org/10.1080/01621459.1963.10500845> arXiv:<https://www.tandfonline.com/doi/pdf/10.1080/01621459.1963.10500845>
- Tianhao Wei and Changliu Liu. 2023. Online Verification of Deep Neural Networks under Domain Shift or Network Updates. arXiv:2106.12732 [cs.LG] <https://arxiv.org/abs/2106.12732>
- Wayne L. Winston. 1991. Operations Research: Applications and Algorithms. (1991). <https://doi.org/10.1002/net.3230180310>
- Haoze Wu, Christopher Hahn, Florian Lonsing, Makai Mann, Raghuram Ramanujan, and Clark Barrett. 2023. Lightweight Online Learning for Sets of Related Problems in Automated Reasoning. In *2023 Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 1–11. https://doi.org/10.34727/2023/ISBN.978-3-85448-060-0_10
- Haoze Wu, Omri Isac, Aleksandar Zeljić, Teruhiro Tagomori, Matthew Daggitt, Wen Kokke, Idan Refaeli, Guy Amir, Kyle Julian, Shahaf Bassan, Pei Huang, Ori Lahav, Min Wu, Min Zhang, Ekaterina Komendantskaya, Guy Katz, and Clark Barrett. 2024. Marabou 2.0: A Versatile Formal Analyzer of Neural Networks. In *Computer Aided Verification*, Arie Gurfinkel and Vijay Ganesh (Eds.). Springer Nature Switzerland, Cham, 249–264. https://doi.org/10.1007/978-3-031-65630-9_13
- Xiaoyong Yuan, Pan He, Qile Zhu, and Xiaolin Li. 2019. Adversarial Examples: Attacks and Defenses for Deep Learning. *IEEE Transactions on Neural Networks and Learning Systems* 30, 9 (2019), 2805–2824. <https://doi.org/10.1109/TNNLS.2018.2886017>
- Mohammad Zahrawi and Khaled Shaalan. 2023. Improving video surveillance systems in banks using deep learning techniques. *Scientific Reports* 13 (05 2023), 16. <https://doi.org/10.1038/s41598-023-35190-9>
- Richard Zhang, Phillip Isola, Alexei A. Efros, Eli Shechtman, and Oliver Wang. 2018. The Unreasonable Effectiveness of Deep Features as a Perceptual Metric. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR*. Computer Vision Foundation / IEEE Computer Society, 586–595.
- Fei Zheng, Qingguo Xu, Zhou Lei, and Huaikou Miao. 2025. AccMILP: An Approach for Accelerating Neural Network Verification Based on Neuron Importance. In *Engineering of Complex Computer Systems*, Guangdong Bai, Fuyuki Ishikawa, Yamine Ait-Ameur, and George A. Papadopoulos (Eds.). Springer Nature Switzerland, Cham, 88–107.
- Duo Zhou, Christopher Brix, Grani A Hanasusanto, and Huan Zhang. 2024. Scalable Neural Network Verification with Branch-and-bound Inferred Cutting Planes. <https://doi.org/10.48550/ARXIV.2501.00200> arXiv:2501.00200 [cs.LG]
- Qiuyu Zhu and Vincent Y. F. Tan. 2020. Thompson sampling algorithms for mean-variance bandits. In *Proceedings of the 37th International Conference on Machine Learning (ICML '20)*. JMLR.org, Article 1075, 10 pages.

A Running Example

We next describe a running example of Algorithm 1, given an MNIST 3×100 fully connected classifier with 3 hidden layers, ten MNIST images $S = \{x_1, \dots, x_{10}\}$, class $c = 0$ and $\epsilon = 0.1$. BaVerLy begins by identifying that all inputs are classified as c . Then, it determines whether to split in layer $\ell = 1$ or $\ell = 2$ as follows. First, it samples an input x_5 and verifies its ϵ -ball when splitting the network at layer 1. Then, it samples x_3 and verifies its ϵ -ball when splitting the network at layer 2. Since the verification runtime of x_3 is shorter, it sets $\ell = 2$. Both ϵ -balls are robust, BaVerLy updates their status in `is_robust` and removes these inputs from S . BaVerLy continues by computing the activation patterns and constructing the binary tree using H-Cluster. Then, it initializes the \mathcal{MAB} agent and begins iterations while the tree is not empty. In the first iteration, the \mathcal{MAB} agent returns $k = 6$. BaVerLy performs a pre-order search and returns the mini-batch $B = \{x_2, x_4, x_6, x_7, x_{10}\}$. Then, it computes the bounds up to the layer $\ell = 2$ for every input in B . Then, it performs a batch analysis from layer 3 till the output layer. The MILP solver returns a counterexample, where $I_{x_2} = 1$. Thus, BaVerLy checks whether $B_\epsilon^\infty(x_2)$ is robust and expedites its analysis by leveraging the already computed bounds from the input layer till layer ℓ . The MILP solver finds a counterexample, indicating that x_2 is *Non-Robust*. Then, BaVerLy removes x_2 from B , adds the constraint $I_{x_2} = 0$ and calls the MILP solver again. The solver returns there is no counterexample, thus BaVerLy determines that all inputs $\{x_4, x_6, x_7, x_{10}\}$ are robust. It then updates their status in `is_robust` and updates the distribution of batch size 5, based on the velocity of B . The velocity is $\frac{4}{25}$, where the denominator is the total time of the analysis, except for the additional analysis time of x_2 . Then, BaVerLy begins another iteration. The \mathcal{MAB} agent returns $k = 4$. Thus, BaVerLy adds all remaining inputs to the mini-batch $B = \{x_1, x_8, x_9\}$. BaVerLy continues similarly to the previous iteration and identifies that all ϵ -balls are robust, thus it updates their status. It updates the distribution of batch size 3 based on the velocity $\frac{3}{23}$. At the end of this iteration, \mathcal{T} is empty, all inputs in S are determined as robust or not and BaVerLy terminates.

B Proof

THEOREM 4.1. *Equation (1) is feasible if and only if there exists $y \in [l_i, u_i]$ for some $i \in [k]$.*

PROOF. In the first direction, we assume that the MILP is feasible. Since $\forall i \in [k] : I_i \in \{0, 1\}$, Equation (1a) implies that there exists $i \in [k]$ such that $I_i = 1$, and $\forall j \neq i \in [k] : I_j = 0$. From assigning the binary values in Equation (1b), it holds that $y \geq l_i$ and for every $j \neq i$, $y \geq 0$. Additionally, from assigning them in Equation (1c), it holds that $y \leq u_i$ and for every $j \neq i$, $y \leq u_M$. Because $l_i \geq 0$ and $u_i \leq u_M$, we obtain $y \in [l_i, u_i]$.

In the second direction, we assume that there exists $y' \in [l_i, u_i]$ for some $i \in [k]$. Consider the assignment of $I_i = 1$, $\forall j \neq i \in [k] : I_j = 0$, and $y = y'$. We show that it satisfies the above MILP. Clearly, Equation (1a) is satisfied. By the assumption, it holds that $l_i \leq y' \leq u_i$. Hence, $y \geq l_i \cdot 1 = l_i \cdot I_i$ and $y \leq u_i \cdot 1 + u_M \cdot 0 = u_i \cdot I_i + u_M \cdot (1 - I_i)$. For every $j \neq i$, we set $I_j = 0$. Since $y' \geq l_i \geq 0$, it follows that $y \geq l_j \cdot 0 = l_j \cdot I_j$. Additionally, $y' \leq u_i \leq u_M$, thus $y \leq u_j \cdot 0 + u_M \cdot 1 = u_j \cdot I_j + u_M \cdot (1 - I_j)$. Therefore, Equation (1b) and Equation (1c) are true for every $j \in [k]$. \square