# Automated machine learning for neural network verification

König, H.M.T.

# Automated Machine Learning for Neural Network Verification

Proefschrift

ter verkrijging van
de graad van doctor aan de Universiteit Leiden,
op gezag van rector magnificus prof.dr.ir. H. Bijl,
volgens besluit van het college voor promoties
te verdedigen op donderdag 9 oktober 2025
klokke 13:00 uur

door

Hendrik Matthias Tilman König
geboren te Saarbrücken, Duitsland
in 1993

**Promotor:**
Prof.dr. H.H. Hoos

**Co-promotor:**
Dr. J.N. van Rijn

**Promotiecomissie:**
Prof.dr. M.M. Bonsangue
Prof.dr. K.J. Batenburg
Prof.dr. T.H.W. Bäck
Prof.dr. J.P. Katoen (RWTH Aachen University)
Prof.dr. M. Kwiatkowska (University of Oxford)



This research was partially supported by TAILOR, a project funded by EU Horizon 2020 research and innovation program under GA No. 952215.

Backcover: Andreas Lisander Strauch

"Technical skill is mastery of complexity while creativity is mastery of simplicity."

E.C. Zeeman

# Contents

# Chapter 1

# Introduction

Neural networks have become increasingly prominent ever since Alan Turing first proposed his idea of *unorganised machines* – computer programs based on trainable networks of largely randomly connected, neuron-like elements [105]. Nowadays, neural networks can be found in various applications, ranging from healthcare to generating artworks, and have enabled the rise of big AI companies, such as OpenAI or Tesla. These neural networks typically consist of millions or even billions of parameters and are commonly referred to as *deep neural networks*.

With the increased adaption of deep neural networks comes the call for safety and trustworthiness of the systems in which they are employed. However, deep neural networks are highly complex and generally suffer from poor explainability; *i.e.*, it often remains unclear how their output was reached. At the same time, they are inherently fragile, and their behaviour is sometimes unexpected and, even more concernedly, unintended (see, *e.g.*, [101]). In some cases, this unintended behaviour can lead to severe consequences, *e.g.*, in the case of the misclassification of traffic signs. Therefore, it becomes necessary to provide formal guarantees about their behaviour; these guarantees can be obtained via *formal verification*. In general, formal software verification seeks to prove or disprove the correctness of a computer program with respect to a certain pre-defined *property* or formal *specification*, using mathematically rigorous techniques.

The most commonly studied verification property of deep neural networks is the *local robustness* property [37, 89]. Local robustness means that a trained neural network produces the same (correct) output when small perturbations are applied to its inputs. Informally speaking, a local robustness property could specify that the image of a speed limit sign is not confused with that of a yield sign due to a small input perturbation,

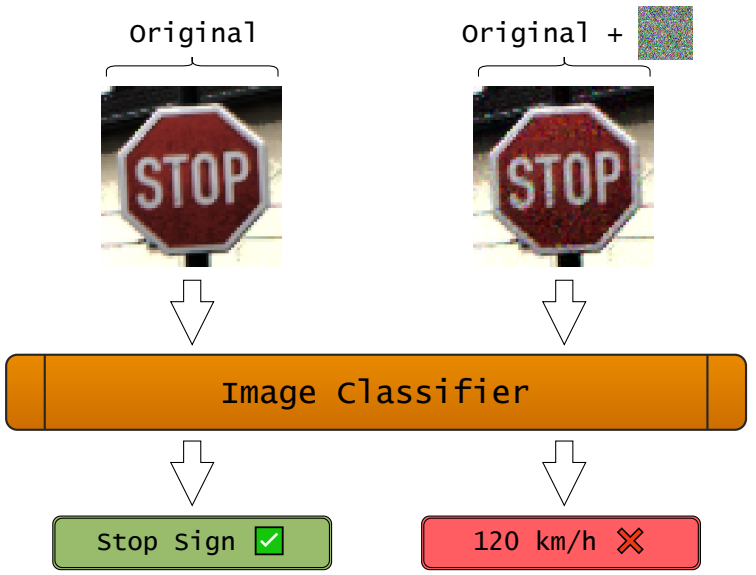



**Figure 1.1:** An example of an adversarial attack. The initial image is accurately identified as a stop sign. However, applying a specific perturbation to this image can lead the image classification model to produce an incorrect prediction, despite the image evidently showing a stop sign. Source: `https://kennysong.github.io/adversarial.js`

as illustrated in Figure 1.1. This specific phenomenon of misclassification due to minor input variations is canonically referred to as *adversarial attack*. Notice that the local robustness property is flexible in the sense that the degree of perturbations as well as output specification can be adjusted to the given use case.

The key challenge of the verification task is to formally describe the behaviour of the neural network model. However, in a deep learning setting, we typically do not know the concept underlying the learning task. For example, we do not know, which pixel values or features make an image belong to a certain class in an image classification setting. Hence, if we cannot define the task the network is supposed to learn, we also cannot prove whether the network correctly learned the intended concept.

Instead, to formally prove its correctness with respect to a given property, we must encode the neural network. There are several ways to encode the network or, in other words, formally specify the network and its function, enabling us to reason over these encodings with respect to a given correctness property. However, at the time of writing this dissertation, there exists a plethora of encoding techniques as well as reasoning methods. Furthermore, the neural network verification task has been shown to be NP-complete [55]. In this work, we seek to provide a better overview of state-of-the-art

neural network verification methods (with a focus on local robustness properties), to understand better the strengths and weaknesses of existing algorithms, and to present meta-algorithmic approaches to reduce the computational costs involved with tackling neural network verification tasks.

## 1.1 Research Questions

Neural network verification with respect to local robustness is a highly diverse research area, and existing methods rely on a broad range of techniques. This raises the question which verification algorithm is most suitable for solving specific types of instances of the verification problem, and what constitutes the state of the art in neural network verification overall, also taking into account different hardware specifications, as some methods rely on CPUs, while others utilise GPU acceleration. There might exist a single verifier dominating in performance over other methods, or it might depend on the exact problem type under consideration. In essence, we seek to answer the following research question:

**RQ1** (Chapter 3) What constitutes the state of the art in neural network verification?

In general, different problem types require different solving approaches, or well-calibrated adaptations of the same approach. This raises the automated algorithm configuration problem. In the context of neural network verification, this becomes especially relevant since some verification approaches rely on mixed integer linear programming (MIP), where it is well known that state-of-the-art solvers (*e.g.* CPLEX) employed by these systems are highly sensitive to the setting of their hyper-parameters. At the same time, configuring a MIP solver embedded into a neural network verification engine introduces new challenges and considerations, such as the heterogeneity of problem instances, which makes it hard to select a single configuration that works well on every instance. Moreover, this introduces the following research question:

**RQ2** (Chapter 4) How can we improve the performance of a MIP-based verification system, leveraging automated algorithm configuration techniques?

Given the inherent complexity of the neural network verification task, solving these problems remains a resource-intensive task even when using state-of-the-art and carefully tuned algorithms. The complexity is further amplified in a portfolio setting, where multiple algorithms run in parallel, introducing inefficiencies through the allocation of computational budget to less effective solvers that run concurrently with

the optimal one until a solution is found. Additionally, there exists the possibility that all algorithms in the portfolio may fail to solve certain instances. This introduces the more general problem of spending compute resources on instances that eventually turn out to be unsolvable within a set cutoff time, *i.e.*, the maximum allowable running time after which the algorithm is terminated. Consequently, a verification system would operate more efficiently if compute resources were allocated towards problem instances that can be solved within the given cutoff time. This leads to the following research question:

**RQ3** (Chapter 5) To which extent can we predict the running time of a given verification algorithm for a specific problem instance?

So far, we have considered the task of adapting an appropriate neural network verification method to a given problem instance (or set of problem instances), where a problem instance is composed of a neural network and verification property. However, when performing verification of a neural network model (or any other kind of performance assessment), we are typically interested in finding a model that achieves optimal performance. In a verification context, we typically measure robust accuracy. This introduces the model selection problem, which is concerned with selecting the best-performing model from a set of candidates on the basis of a predefined performance criterion. Therefore, we are also interested in efficiently performing robust model selection by leveraging meta-algorithmic approaches. Thus, we arrive at the following research question:

**RQ4** (Chapter 6) How can we efficiently select the neural network model from a given set of models that achieves the highest certified robust accuracy?

In summary, this thesis seeks to improve the state of the art in neural network verification systems by leveraging recent advances in meta-algorithmic approaches, such as automated algorithm configuration, portfolio construction, running time prediction and model selection techniques for optimised resource allocation.

## 1.2 Contributions of this thesis

The core technical content of this thesis has been published in the form of research papers, with each thesis chapter aligning with a specific paper.

In Chapter 3, we investigate the highly diverse landscape of neural network verification algorithms with respect to local robustness properties and present a detailed

overview of current algorithmic approaches. To enable a principled analysis, we define several criteria for defining the state of the art, and perform an empirical performance analysis of selected methods. For this performance analysis, we created a new and diverse benchmark consisting of neural network verification problem instances and divided this benchmark into subcategories based on different neural network activation functions. In addition, we introduce specific measures capturing not only the stand-alone performance of a given verification algorithm but also their performance in relation to others. Using these *complementarity metrics*, we show that no single best algorithm dominates performance across all verification problem instances and illustrate the potential of leveraging algorithm portfolios. Furthermore, we show that some activation functions are highly under-supported by existing verification methods. The research presented in this chapter has given rise to the following research articles:

- Matthias König, Annelot W Bosman, Holger H Hoos, and Jan N van Rijn. Critically Assessing the State of the Art in Neural Network Verification. *Journal of Machine Learning Research*, 25(12):1–53, 2024.

The paper above is an extension of the following workshop paper:

- Matthias König, Annelot W Bosman, Holger H Hoos, and Jan N van Rijn. Critically Assessing the State of the Art in CPU-based Local Robustness Verification. In *Proceedings of the Workshop on Artificial Intelligence Safety 2023 (SafeAI 2023) co-located with the Thirty-Seventh AAAI Conference on Artificial Intelligence (AAAI2023)*, pages 1–9, 2023. **[Best Paper Award]**

In Chapter 4, we present a concrete approach to leverage algorithm portfolios for neural network verification in combination with automated algorithm configuration. Specifically, we consider neural network verification based on mixed integer linear programming (MIP) encodings, where the verification property is treated as a minimisation problem and solved by a commercial MIP solver. We show that by using automated algorithm configuration and portfolio construction techniques, the performance of a MIP-based verification system can be substantially improved in terms of running time as well as the total number of solved problem instances within a given time budget. The research presented in this chapter has given rise to the following research articles:

- Matthias König, Holger H Hoos, and Jan N van Rijn. Speeding up neural network robustness verification via algorithm configuration and an optimised mixed integer linear programming solver portfolio. *Machine Learning*, 111(12):4565–4584, 2022.

- Matthias König, Holger H Hoos, and Jan N van Rijn. Speeding Up Neural Network Verification via Automated Algorithm Configuration. In *ICLR Workshop on Security and Safety in Machine Learning Systems*, pages 1–4, 2021.

In Chapter 5, we introduce novel features describing instances of the neural network verification problem. These features take into account the given instance as well as internal mechanics of the verification algorithm used. We focus on several state-of-the-art verification algorithms and show that our features enable the reliable prediction of timeouts; *i.e.*, cases in which a specific instance cannot be solved within the given time budget. This prediction is performed by a supervised machine learning model trained on these features. Using this timeout prediction model, we can substantially reduce the computational costs demanded by the verification system via early termination of verification queries that would otherwise result in a timeout. The research presented in this chapter has given rise to the following research article:

- Konstantin Kaulen, Matthias König, and Holger H Hoos. Dynamic algorithm termination for branch-and-bound-based neural network verification. In *To appear in Proceedings of the 39th AAAI Conference on Artificial Intelligence (AAAI-25)*, pages 1–9, 2025.

Lastly, in Chapter 6, we consider the task of robust model selection. Specifically, this task involves selecting the neural network model from a given set of candidate models that shows the highest degree of adversarial robustness. Towards this end, we propose a racing algorithm that leverages the estimated likelihood of an instance to be robust and prioritises those during the model evaluation procedure. This enables an early elimination of candidate models after verifying only a small number of input instances. We show that our approach reduces the computational burden of selecting the most robust neural network model by up to two orders of magnitude on standard benchmarks from the literature, compared to an exhaustive evaluation (*i.e.*, standard) approach. The research presented in this chapter has given rise to the following research article:

- Matthias König, Holger H Hoos, and Jan N van Rijn. Accelerating Adversarially Robust Model Selection for Deep Neural Networks via Racing. In *Proceedings of the 38th AAAI Conference on Artificial Intelligence (AAAI-24)*, pages 21267–21275, 2024.

Altogether, the contributions of this thesis enable the scaling of state-of-the-art neural network verification algorithms to problem instances that were previously

unsolved as well as the usage of the these algorithms in a more resource-efficient manner.

## 1.3 Other Work by the Author

From the following papers, the work presented by König et al. [65] is directly related to the contents of this thesis, as it applies the concept of automated algorithm configuration to linear bounding techniques for incomplete neural network verification. The remaining papers are not directly related.

- Matthias König, Xiyue Zhang, Holger H Hoos, Marta Kwiatkowska, and Jan N van Rijn. Automated Design of Linear Bounding Functions for Sigmoidal Nonlinearities in Neural Networks. In *Proceedings of the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases*, 2024.
- Bruno Veloso, Luciano Caroprese, Matthias König, Sónia Teixeira, Giuseppe Manco, Holger H Hoos, and João Gama. Hyper-Parameter Optimization for Latent Spaces in Dynamic Recommender Systems. In *Proceedings of the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases*, pages 249–264, 2021.
- Matthias König, Holger H Hoos, and Jan N van Rijn. Towards Algorithm-Agnostic Uncertainty Estimation: Predicting Classification Error in an Automated Machine Learning Setting. In *ICML Workshop on Automated Machine Learning*, pages 1–6, 2020.

# Chapter 2

# Preliminaries

In this chapter, we will introduce the most important concepts and methods on which this thesis are built, including adversarial attacks, existing approaches to formal verification as well as meta-algorithmic techniques, such as automated algorithm configuration and running time prediction. Notice that the research articles presented in Chapter 1.2 are, in part, based in the content of this chapter.

## 2.1 Adversarial Attacks

In recent years, deep learning methods based on neural networks have been increasingly adopted within various safety-critical domains and use contexts, ranging from manoeuvre advisory systems in unmanned aircraft to face recognition systems in mobile phones ([50, 2, 53]). Concurrently, it is now well known that neural networks are vulnerable to *adversarial attacks* [101], where a given input is manipulated, often in subtle ways, such that it is misclassified by the network.

Adversarial examples or negatives are network inputs that are indistinguishable from regular inputs, but cause the network to produce misclassifications [101]. These adversarial examples can be created by applying a hardly perceptible perturbation to the original instance that maximises model error while staying close to the initial example. The most prevalent distance metrics used to evaluate adversarial distortion are the $l_1$ [13, 16], $l_2$ [101] and $l_\infty$ [37, 89] norms. Prominent methods for creating adversarial examples include *projected gradient descent* (PGD) [77] and *fast gradient sign method* (FGSM) [37], which, in essence, seek to identify the input pixels that have the largest impact on the model prediction, and modify those until the model produces

a misclassification. Notice that adversarial examples could also occur due to natural distortions, such as signal noise or changing conditions in the environment.

In the case of image recognition tasks, the perturbation required to trigger a misclassification, whether it is adversarially crafted or arises accidentally, can be so small that it remains virtually undetectable to the human eye.

## 2.2 Formal Neural Network Verification

Various methods have been proposed to establish the robustness of neural networks against adversarial attacks. Some of these methods perform *heuristic* attacks, where adversarial examples are found using approximate, iterative strategies to solve the underlying optimisation problem [37, 67, 14]; however, these methods are empirical in the sense that they do not paint a full picture of the robustness of a given network to adversarial attacks, as one defense mechanism might still be circumvented by another, possibly new class of attacks.

In light of this, several approaches have been developed to more thoroughly verify neural networks [94, 6, 27, 55, 26, 35, 115, 12, 104, 8]. These *formal* verification methods can assess the robustness of a given network in a principled fashion, which means that they yield provable guarantees on certain properties of input-output combinations. For a classifier, a property can be that instances, which are in close distance to a certain input $x$, belong to the same class as $x$.

This specific type of assessment refers to *local robustness verification*, a broadly studied verification task, in which a network is systematically tested against various input perturbations under pre-defined norms, where the most commonly considered norm is the $l_\infty$-norm [37, 89], representing the maximum discrepancy between two inputs. The $l_\infty$-norm is most suitable for adversarial testing, as it ensures that perturbations are minimally perceptible yet uniformly applied across all input dimensions. It should be noted that recent work has proposed to move beyond pixel-based perturbations, which account for realistic sensor errors, to *semantic* perturbations; *i.e.*, linear transformations representing changes in contrast, luminosity, scaling, rotation, and other factors [83, 39]. This specific type of verification, though, falls outside the scope of this thesis.

In general, formal verification algorithms can be characterised by three criteria: *soundness, completeness* and computational cost. A sound algorithm will only report that a property holds if the property actually holds. An algorithm that is complete will correctly state that a property holds whenever it holds. While it is favourable to produce verifiers that can certify every given instance in a dataset, there is a trade-off

between the completeness of a verification algorithm and its scalability in terms of computational complexity. The neural network verification problem is highly complex and has been proven to be NP-complete [55]. This complexity arises from the need to determine whether any input from a given domain can produce an output that violates specified constraints, which translates into solving a non-convex optimization problem due to the presence of non-linear activation functions in the neural network. Therefore, it is not surprising that for large networks and/or instance sets the problem quickly becomes practically intractable.

Consequently, some verification algorithms forego completeness to improve computational efficiency by resorting to approximations [6, 26, 35, 115, 12]. These approximations, however, do not always return the actual solution to a given verification problem but can result in mismatches or cases where the solution remains unknown. Other incomplete methods seek to add random noise to the inputs during training to smooth a neural network classifier and then derive the certified robustness for this smoothed classifier [69, 19]. While these approaches scale to larger network architectures, their robustness guarantees remain probabilistic, *i.e.*, the method estimates the probability that the predicted label remains most likely even under small perturbations to the input. In contrast to incomplete verification methods, however, this does not provide sound verification, as there is no formal guarantee that the robustness property actually holds. Furthermore, randomised smoothing has been found to come at the cost of classifier accuracy [82]. As can be seen from this example, increased scalability of a verification method usually comes at the cost of performance loss in other areas.

### 2.2.1 Problem Definition

Formally, the local robustness verification problem we study in the remainder of this thesis can be described as follows. Consider a feed-forward neural network classifier comprising $n$ layers with a $k_0$-dimensional input and a $k_n$-dimensional output. Let $\boldsymbol{x} \in \mathcal{D}_{\boldsymbol{x}} \subseteq \mathbb{R}^{k_0}$ denote the input and $\boldsymbol{y} \in \mathcal{D}_{\boldsymbol{y}} \subseteq \mathbb{R}^{k_n}$ denote the class label associated with that input (in a certain representation), where $\mathcal{D}_{\boldsymbol{x}}$ and $\mathcal{D}_{\boldsymbol{y}}$ represent the domains of possible values for $\boldsymbol{x}$ and $\boldsymbol{y}$, respectively. The neural network represents a function $\boldsymbol{f}$ that aims to model the relation between input $\boldsymbol{x}$ and the associated class label $\boldsymbol{y}$, expressed as $\hat{\boldsymbol{y}} = \boldsymbol{f}(\boldsymbol{x})$, where $\hat{\boldsymbol{y}}$ represents the output of the neural network based on input $\boldsymbol{x}$. Note that the model can also produce misclassifications, in those cases $\hat{\boldsymbol{y}} \neq \boldsymbol{y}$.

To address the verification problem, we need to validate the specified conditions for input-output relationships of $\boldsymbol{f}$. In the context of local robustness verification for

image classification networks, the goal is to determine whether input instances within distance $\epsilon$ of a specific input $\boldsymbol{x_0}$ belong to the same output class as $\boldsymbol{x_0}$. This problem can be formulated as follows (based on [73]):

$$\forall \boldsymbol{x} : \|\boldsymbol{x} - \boldsymbol{x}_0\|_p \leq \epsilon \implies \boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{f}(\boldsymbol{x}_0)$$

Here, $\boldsymbol{x}$ represents a possible input, for which we can compute the distance to $\boldsymbol{x_0}$ based on an $\ell_p$ distance metric. If the neural network always predicts correctly within the pre-defined distance $\epsilon$, the network is verified to be robust with respect to the verification property. Otherwise, $\boldsymbol{x}$ has been found to be an adversarial example, rendering the neural network unsafe, *i.e.*, non-robust.

### 2.2.2 Incomplete Verification

The most scalable and efficient incomplete verification approaches attempt to produce sound bounds for the output nodes of a given network by identifying linear relationships between each hidden or input neuron and the output neurons. Therefore, in the case of ReLU activation functions, each ReLU neuron whose inputs span over both the positive and negative domain must be relaxed, which typically involves approximating the ReLU function by means of linear bounds; see *e.g.*, [119]. These neurons are referred to as being *unstable*. There also exist relaxations for other commonly used non-linearities, such as sigmoid or hyperbolic tangent [65, 119].

To find a linear relationship between the input and the output of a network, the output bounds of the last layer are back-propagated, such that the input bounds of each layer are replaced recursively by the output bounds from the previous layer [99, 119]. These techniques are referred to as *linear bound propagation* (LBP) methods. Once the linear connections have been established, the easier linear optimisation problem can be solved to obtain the final bounds of the output layer using Linear Programming (LP) solvers. Alternatively, it has been proposed to employ *symbolic interval propagation* (SIP) to calculate linear bounding equations of the output [8, 110]. In comparison to LBP techniques, SIP-based approaches propagate input variables symbolically and, thus, are able to identify inter-dependencies between them, which may lead to overall tighter bounds.

Finally, adversarial attacks can also be considered an incomplete verification approach, as the existence of an adversarial example proves the violation of the given property [37].

### 2.2.3 Complete Verification

Early work on complete verification of neural networks utilised *satisfiability modulo theories (SMT)* solvers [55, 56, 90, 91, 92], which determine whether a set of logic constraints is satisfiable [84]. The resulting verification problems are NP-complete and challenging to solve in practice. Some SMT-based verification algorithms, such as those proposed by Katz et al. [55, 56], employ the well-known simplex algorithm [21] for assigning values to the SMT variables.

Alternatively, it is possible to formulate the verification task as a constraint optimisation problem using *mixed integer programming (MIP)* [8, 25, 75, 104]. MIP solvers essentially optimise an objective function subject to a set of constraints. Generally, optimisation problems are well studied, and by approaching verification tasks from that angle, techniques and insights from well-developed areas of computer science and operations research can be leveraged. MIP-based verification algorithms assign variables to each node in the network and, more specifically, encode non-linearities by means of binary variables indicating whether a node is in an active or inactive state; for ReLU nodes, this means whether a the pre-activation value of the node exceeds a value of zero. Approaches differ in the way perturbations are encoded into the program as well as in the specification of their objective function. MIP problems, similar to SMT problems, can be challenging to solve and tend to be computationally expensive (in terms of CPU time and memory). Further details on MIP-based verification will be provided in Chapter 4.

To overcome the computational complexity of SMT and MIP, it has been proposed to use the well-known *branch-and-bound* algorithm [68] for solving the verification problem, regardless of whether it is modelled as MIP or SMT [11, 12, 22, 27, 109]. Neural network verification algorithms based on branch-and-bound consist of two main steps: (*i*) branching and (*ii*) bounding. Branching involves splitting the domain of one or more variables (based on the nodes in the network) into sub-problems of the original problem; for instance, a ReLU node can be split into its positive and negative domain, which creates new sub-problems for each of these cases, respectively. These (relaxed) sub-problems are then solved by cheap but incomplete verification algorithms, such as LBP techniques, which determine a lower bound to the verification problem, while upper bounds are found via falsification algorithms [24]. By repeating these steps, the bounds, *i.e.*, the upper and lower limits on the possible value of a solution to the verification problem, are tightened in each iteration. There exist many different branching schemes and bounding algorithms, which vary in tightness of the bounds

and performance in terms of running time.

To formulate the constraints used in the above-mentioned methods, the non-linear activation functions of a neural network are usually relaxed. This is mostly done by approximating the original non-linear activation function by at least two linear functions, forming upper and lower bounds [27, 99, 112, 113, 119]. Employing the linear bounds as relaxation to the activation function increases the feasible region of each variable in the model, and as the nodes in each layer are dependent on the previous layer, the bounds on each consecutive layer become looser. The approximation, thus, provides a trade-off, as loose and fast bounds lead to large feasible regions while obtaining tight bounds tends to be computationally expensive. The way in which non-linearities are approximated presents a key distinguishing factor between complete verification algorithms.

Alternatively, *symbolic interval propagation* has been proposed to compute bounds on the output range of the network for a given input and use those as additional constraints [8, 40, 110, 111]. The output range of the output layer is obtained by propagating the bounds through the network, which renders it unnecessary to encode the entire network and use computationally expensive solvers. Symbolic interval propagation lends itself as an incomplete verification method, but it can also complement complete methods, improving their efficiency by reducing the size of the feasible region of the problem, compared to the looser approximation described above.

Other bound approximation methods include *polyhedra*, *zonotope* and *star-set abstraction*. Polyhedra abstraction produces one lower bound for the approximation based on the trained network, instead of two bounds, as used in symbolic interval propagation, where the latter results in tighter bounds [99, 119]. Zonotope abstraction is similar to polyhedra abstraction and is able to model dependencies between the zonotope representation of different network layers [35, 98]. In contrast to polyhedron transformations, the zonotope transformations scale polynomially, and optimisation is efficient. Star sets are a generalisation of the zonotope abstraction, as they are not restricted to being symmetric [4]. They are similar to zonotopes; however, optimisation is less efficient, as it requires solving a linear program.

Throughout this thesis, we will use and study several verification algorithms stemming from various paradigms. Notice that we focus on complete verifiers. An overview of all considered methods as well as their reference and location in this thesis are presented in Table 2.1.

**Table 2.1:** Overview of verification methods considered in this thesis.

| Verifier | Reference | Paradigm | Chapter |
|---|---|---|---|
| BaB | Bunel et al. [12] | Branch-and-bound | 3 |
| BaDNB | DePalma et al. [22] | Branch-and-bound | 3,5 |
| $\alpha\beta$-CROWN | Wang et al. [111] | Branch-and-bound | 3,5,6 |
| Marabou | Katz et al. [56] | SMT | 3 |
| MIPVerify | Tjeng et al. [104] | MIP | 4 |
| MN-BaB | Ferrari et al. [29] | Branch-and-bound | 3 |
| Neurify | Wang et al. [110] | SIP | 3 |
| nnenum | Bak et al. [4] | Star-set abstraction | 3 |
| Venus | Botoeva et al. [8] | MIP | 4 |
| VeriNet | Henriksen & Lomuscio [40] | SIP | 3,5,6 |

## 2.3 Automated Algorithm Configuration

In general, the algorithm configuration problem can be described as follows: Given an algorithm $A$ (also referred to as the *target algorithm*) with parameter configuration space $\Theta$ (arising from the domains of individual parameters), a set of problem instances $\Pi$ and a cost metric $c : \Theta \times \Pi \to \mathbb{R}$, find a configuration $\theta^* \in \Theta$ that minimises cost $c$ across the instances in $\Pi$:

$$\theta^* \in \arg\min_{\theta \in \Theta} \sum_{\pi \in \Pi} c(\theta, \pi) \tag{2.1}$$

The general workflow of the algorithm configuration procedure starts with picking a configuration $\theta \in \Theta$ and an instance $\pi \in \Pi$. Next, the configurator initialises a run of algorithm $A$ with configuration $\theta$ on instance $\pi$ with CPU time cutoff $k$ and measures the resulting cost $c(\theta, \pi)$. The configurator uses this information about the performance of the target algorithm to find a configuration that performs well on the training instances. Once its configuration budget (e.g., time budget) is exhausted, it returns the current incumbent, *i.e.*, the best configuration found so far. Finally, when running the target algorithm with configuration $\theta^*$, it should result in lower cost (such as average running time) across the benchmark set.

Automated algorithm configuration has been shown to work effectively in the context of SAT solving [43, 47], scheduling [18], mixed-integer programming [44, 76], answer set solving [34], AI planning [107] and machine learning [103, 31].

## 2.4 Algorithm Portfolios & Selection

In cases where the performance of an algorithm varies greatly from one instance to another and where the performance of several different algorithms (or algorithm configurations) complements each other across an instance distribution, one can make use of *algorithm portfolios* [36, 42]. Such portfolios combine multiple algorithms (or algorithm configurations) in such a way that a much broader range of performance characteristics can be exploited, compared to running a single algorithm on its own. Added this paragraph with formal description.

A widely recognised approach for portfolio construction is Hydra [116], which has been shown to work effectively, *e.g.*, in the context of MIP-based neural network verification [61]. Hydra employs a greedy algorithm that scores a configuration based on its actual performance if it surpasses the portfolio on the current instance; otherwise, it assigns the portfolio's performance cost. Consequently, potential configurations are evaluated solely based on their contribution to improving the portfolio, guiding the configurator to prioritise instances on which the current portfolio underperforms. The portfolio construction proceeds as follows: Let $P_i$ denote the portfolio after iteration $i$, starting with $P_0 := \{\}$, which is the empty portfolio. In each iteration $i$, the configurator identifies a new configuration $\theta_i$, which is then added to the portfolio. In the first iteration, this results in $P_1 := \{\theta_1\}$. From the second iteration onward, the performance metric is adjusted such that if the incumbent configuration underperforms the portfolio on a specific instance, its score is replaced with the portfolio's performance. Following each iteration, the newly generated configuration $\theta_i$ is evaluated, and the portfolio is updated according to a predefined portfolio updating strategy, yielding $P_i = P_{i-1} \cup \{\theta_i\}$. Notice that this framework can be extended to allow adding multiple configurations per iteration, however, this requires adjustments to the performance metric and the updating strategy.

Algorithm portfolios can employ all algorithms in a parallel fashion or, alternatively, provide the basis for per-instance algorithm selection mechanisms. The latter are based on instance-specific features, which are used to train a statistical model subsequently used for selecting the algorithm to be run on a given problem instance, *e.g.*, based on performance predictions for each individual algorithm in the portfolio [118]. In the former case, all algorithms are run in parallel on a given problem instance, and the portfolio terminates once one algorithm has returned a solution. This implicitly ensures that we always benefit from the best-performing algorithm in the portfolio; however, it comes at the cost of increased use of parallel resources when compared to

per-instance selection from a portfolio of algorithms. Thus, when evaluating a parallel algorithm portfolio, it is important to ensure that all algorithms together do not exceed the global computing budget used by a single baseline algorithm.

## 2.5 Running Time Prediction

In the context of NP-complete problems, such as SAT, MIP or TSP, it has been shown that running time can vary drastically depending on the instance and algorithm used [48]. While there is only very little understanding of the reasons for this behaviour, it is possible to predict running times of previously unseen SAT, MIP and TSP problem instances reasonably well based on cheaply computable instance features by fitting a statistical model on the running time of a given algorithm [48]. Running time prediction has various applications that seek to minimise costs and improve the efficiency of machine learning systems; those range from algorithm selection (as mentioned in the previous section) to scheduling [58]. Furthermore, running time prediction provides insights into the relationship between instance characteristics and algorithm running time and, thus, informs us about instance complexity.

Following the notation of [48], we characterise a problem instance using a list of $m$ features $\mathbf{z} = [z_1, \ldots, z_m]$, selected from a specified feature space $\mathcal{F}$. These features are typically computed by specialised code to extract relevant characteristics for any given problem instance. In the context of neural network verification, such features could be, for example, the number of unstable nodes or the prediction margin.

Consider $\mathbb{R}$ the space of real numbers indicating an algorithm's performance measure, such as the running time in seconds on a given machine. For an algorithm $A$ and a distribution of instances described by a feature space $\mathcal{F}$, we are interested in modelling the running time as a function $f : \mathcal{F} \to \mathbb{R}$ that maps the feature vector $\mathbf{z} \in \mathcal{F}$ to the performance measure of the algorithm.

To model the performance of an algorithm $A$ on an instance set $\Pi$, we run $A$ on all instances $\pi_i \in \Pi$ and record the resulting performance values $y_i$. We capture the $m$-dimensional feature vector $\mathbf{z}_i$ of the instance used in the $i$-th run to be used as our vector of predictor variables. The training data for our performance model is then $\{(\mathbf{z}_1, y_1), \ldots, (\mathbf{z}_n, y_n)\}$, where $y_i$ (with $1 \leq i \leq n$) is the performance (*e.g.*, running time) of $A$ on instance $\pi_i$. Lastly, we use $\mathbf{y}$ for the vector of performance values $[y_1, \ldots, y_n]$.

# Chapter 3

# Critically Assessing the State of the Art in Neural Network Verification

Neural network verification with respect to local robustness is a highly diverse research area, and existing methods rely on a broad range of techniques. At the same time, neural networks differ in terms of their architecture, such as the number of hidden layers and nodes, the type of non-linearities, *e.g.*, ReLU, Sigmoid or Tanh, and the type of operations they employ, *e.g.*, pooling or convolutional layers. This diversity, both in terms of verification approaches and neural network design, makes it non-trivial for researchers or practitioners to assess and decide which method is most suitable for verifying a given neural network [15]. This challenge is amplified by the fact that the neural network verification community does not (yet) use commonly agreed evaluation protocols, which makes it difficult to draw clear conclusions from the literature regarding the capabilities and performance of existing verifiers. More precisely, existing studies use different benchmarks and, so far, have not provided an in-depth performance comparison of a broad range of verification algorithms, as we will further outline in Section 3.1.

Recently, a competition series has been initiated, in which several verifiers were applied to different benchmarks (*i.e.*, networks, properties and datasets) and compared in terms of various performance measures, including the number of verified instances as well as running time [87]. While the results from these competitions have provided

valuable insights into the general progress in neural network verification, several questions remain unexplored. Most importantly, the ranking of algorithms based on their aggregated performance scores makes it difficult to assess in detail the strengths or weaknesses of verifiers on different instances. Indeed, looking at the competition results, one easily gets the impression that a single approach dominates 'across the board' — an assumption that is known to be inaccurate for other problems involving formal verification tasks; see, *e.g.*, [117] or [52] for SAT.

In this chapter, we focus exclusively on local robustness verification in image classification against perturbations under the $l_\infty$-norm. This scenario represents a widely studied verification task, with a large number of networks being publicly available and many verifiers providing off-the-shelf support. Notice that most verification tasks can be translated into local robustness verification queries [95]; we, therefore, believe that our findings are broadly applicable. Moreover, we seek to go beyond existing benchmarking approaches and shed light on previously unanswered questions regarding the state of the art in local robustness verification from a practitioner's point of view – a perspective that complements the insights from the VNN competition, where the participating tools are carefully adapted to the given benchmarks by their developers. Our contributions in this chapter are as follows and, altogether, seek to answer RQ1 of this thesis:

- We analyse the current state of practice in benchmarking verification algorithms;
- we perform a systematic benchmarking study of several, carefully chosen GPU- and CPU-based verification methods based on a *newly assembled* and diverse set of networks, including 38 CIFAR and 41 MNIST networks with different activation functions, representing a much larger number of networks than typically considered, each verified against several robustness properties, for which we expended a total of approximately 1 GPU and 16 CPU years in running time;
- we present a categorisation of verification benchmarks based on verifier compatibilities with different layer types and operations;
- we quantify verifier performance in terms of the number of solved instances, running time, as well as marginal contribution and Shapley value, showing that top-performing verification algorithms strongly complement rather than consistently dominate each other in terms of performance, a finding that we also show to hold for the results of the 2022 VNN Competition – *e.g.*, while the verifiers nnenum and PeregriNN achieved competitive performance in the *FC*

category of the competition, the former solved many instances unsolved by the latter and vice versa.

## 3.1 Common Practices in Benchmarking Neural Network Verifiers

As explained in Chapter 2, formal verification algorithms can be either *complete* or *incomplete* [71]. An algorithm that is incomplete does not guarantee to report a solution for every given instance; however, incomplete verification algorithms are typically *sound*, which means they will report that a property holds only if the property actually holds. On the other hand, an algorithm that is sound and complete, when given sufficient resources to be run to completion, will correctly state that a property holds *whenever* it holds, and, in particular, will determine accurately when the property does not hold. In this study, we focus on complete algorithms, as those arguably represent the most ambitious form of neural network verification, making them preferable over incomplete methods, especially in safety-critical applications. Furthermore, we focus on the verification of real-valued networks, which are typically considered in the verification literature, although there exist methods for the verification of other network types; see, *e.g.*, the work of [88] or [49] on binarised networks.

Considering the diversity in neural network verification problems, it is quite natural to assume that a single best algorithm does not exist, *i.e.*, a method that *always* outperforms all others. It is still hard to identify to what extent a method contributes to the state of the art, mainly because verification methods are typically evaluated *(i)* on a small number of benchmarks, which have often been created for the sole purpose of evaluating the method at hand, and *(ii)* against baseline methods for which it is often unclear how they were chosen, leading to several methods claiming state-of-the-art performance without having been directly compared. We note that in the context of local robustness verification, a benchmark most often represents a neural network classifier trained on the MNIST or CIFAR-10 dataset, respectively.

As previously mentioned, a competition series has been established with the goal of providing an objective and fair comparison of the state-of-the-art methods in neural network verification, in terms of scalability and speed [87]. The VNN competition was held every year since 2020, with different protocols (*e.g.*, for running experiments, scoring, etc.), benchmarks and participants. Here, we focus on the 2022 edition. Within VNN 2022, a total of 12 benchmarks were considered, of which 6 represented test cases

for local robustness verification of image classification networks. Notice that one of these benchmarks considers bias field perturbations, which are reduced to a standard $l_\infty$-norm specification. Benchmarks were proposed by the participants themselves and included a total of 13 CIFAR, 2 MNIST and 2 (Tiny)ImageNet networks, which differed in terms of architecture components, such as non-linearities (*e.g.*, ReLU, Tanh, Sigmoid) and layer operations (*e.g.*, convolutional or pooling layers, skip connections). Networks were trained on the CIFAR-10, CIFAR-100, MNIST, TinyImageNet and ImageNet datasets, respectively. Moreover, each benchmark was composed of random image subsets, excluding images that were misclassified by the given network, along with varying perturbation radii.

This competition overcame several of the previously reported limitations with regard to the evaluation of network verifiers. Most notably, it covered a relatively large and diverse set of neural networks. Moreover, thanks to the active participation from the community, 12 verification algorithms were included in the competition. At the same time, we see room for further research into the performance of neural network verifiers.

First and foremost, the competition seeks to determine the current state of the art; however, the competition ranking and scores do not sufficiently quantify the extent to which an algorithm actually contributes to the state of the art. In other words, it is in the nature of competitions to determine a winner, at least implicitly suggesting that a single approach generally outperforms all competitors. However, some verification algorithms might have limited but distinct areas of strength, which cannot be identified through aggregated performance measures, such as the total number of verified instances. Although the competition report [87] shows that individual verifier performance differs among benchmarks, it remains unclear whether all algorithms solve the same set of instances in the given benchmark, or if they complement each other. Similarly, it does not reveal whether or not methods are correlated in their performance.

Furthermore, in our study, we conducted both a joint and separate analysis of CPU- and GPU-based methods. This choice was motivated by the inherent challenges that arise when attempting to compare these two types of algorithms. Indeed, the competition results suggest that GPU-based methods are more efficient than CPU-based algorithms [87]; however, GPU resources are typically more expensive to run. Additionally, while CPU-based methods can run a single verification query on each CPU core, allowing for multiple instances to be solved in parallel on the same machine, GPU-based methods utilise the full GPU when solving a single verification query. In fact, running multiple queries in parallel, each utilising a single CPU core, might be

**Table 3.1:** Overview of reviewed verification methods and their eligibility for inclusion in our assessment based on their (*i*) completeness **and** (*ii*) presence in the top five ranking of the 2021 or 2022 VNN Competition **or** (*iii*) support through DNNV. Check marks indicate that a verifier satisfies the criterion, while cross marks indicate that it does not. If a verifier satisfies the inclusion criteria but is superseded by another, more recent method, the former is not included.

| Verifier | Complete? | In VNN Comp? | In DNNV? | GPU/GPU? | Reference |
|---|---|---|---|---|---|
| BaB | ✓ | ✗ | ✓ | CPU | [12] |
| BaDNB | ✓ | ✓ | ✗ | GPU | [22] |
| $\alpha\beta$-CROWN | ✓ | ✓ | ✗ | GPU | [111] |
| ERAN[1] | ✓ | ✓ | ✓ | GPU | [96] |
| Marabou | ✓ | ✓ | ✓ | CPU | [56] |
| MIPVerify[2] | ✓ | ✗ | ✓ | CPU | [104] |
| MN-BaB | ✓ | ✓ | ✗ | GPU | [29] |
| Neurify | ✓ | ✗ | ✓ | CPU | [110] |
| nnenum | ✓ | ✓ | ✓ | CPU | [4] |
| Planet[3] | ✓ | ✗ | ✓ | CPU | [27] |
| Reluplex[4] | ✓ | ✗ | ✓ | CPU | [55] |
| VeriNet | ✓ | ✗ | ✓ | CPU | [40] |

[1]Superseded by MN-BaB.
[2]Local robustness verification not supported via DNNV.
[3]Superseded by BaB.
[4]Superseded by Marabou.

a more efficient approach than running each query sequentially, while utilising all cores. Thus, overall, it remains challenging to set up a comparison between CPU- and GPU-based verification algorithms in an unbiased manner, which is why we present both a direct comparison and a separate analysis.

Finally, the competition approaches the state of the art from the perspective of a tool developer, where the developer is given access to the benchmarks beforehand and can adapt their implementations as well as hyperparameter settings accordingly. On the other hand, in this study, we assess the state of the art from the perspective of a practitioner, who typically uses a verification tool out of the box, is bounded by the limitations of the implementations, and might also not be able to tune the hyperparameters of these tools. We believe that both these perspectives on the state of the art are valid and give complementary insights.

## 3.2 Verification Algorithms under Assessment

We consider eight complete neural network verification algorithms in this study; each of these was chosen because it fulfilled one of the following conditions: it was (*i*) ranked among the top five verification methods according to the 2021 and 2022 VNN competitions or (*ii*) supported by the recently published DNNV framework [95]. Table 3.1 presents an overview of all methods we reviewed and their eligibility for inclusion based on the criteria specified above. Notice that some verification methods, such as Neurify [110] or BaDNB [22], did not participate in the 2022 edition of the VNN competition. On the other hand, it can be assumed that these methods also contribute to the state of the art in neural network verification. For example, BaDNB, which is part of the OVAL framework, reached third place in the 2021 edition of the competition [3] but did not compete in 2022. Altogether, we consider our set of algorithms to be representative of recent and important developments in the area of complete neural network and, more specifically, local robustness verification.

All methods were employed with their default hyperparameter settings, as they would likely be used by practitioners. In other words, one aspect of our study is to capture the situation someone using existing tools "out of the box" might face. We note that the performance of a verifier might improve if its hyperparameters were optimised specifically for the given benchmark; however, most verifiers have dozens of hyperparameters (or employ combinatorial solvers that come with their own, extensive set of hyperparameters), which makes this a non-trivial task, requiring additional expertise and resources.

### 3.2.1 CPU-Based Methods

The CPU-based verification algorithms we considered are the following.

**BaB.** The algorithm proposed by Bunel et al. [12] restates the verification problem as a global optimisation problem, which is then solved using branch-and-bound search. It further incorporates algorithmic improvements to branching and bounding procedures such as *smart branching*; *i.e.*, before splitting, it computes fast bounds on each of the possible subdomains and chooses the one with the tightest bounds. This method supports ReLU-based networks; for the remainder of this chapter, we refer to it as BaBSB.

**Marabou.** The Marabou framework [56] employs SMT solving techniques, specifically the lazy search technique for handling non-linear constraints. Furthermore, Marabou employs deduction techniques to obtain information on the activation func-

tions that can be used to simplify them. The core of the SMT solver is simplex-based, which means that the variable assignments are made using the simplex algorithm. Marabou supports ReLU and Sigmoid activation functions as well as MaxPooling operations.

**Neurify.** The verification algorithm proposed by Wang et al. [110] relies on symbolic interval propagation to create over-approximations, followed by a refinement strategy based on symbolic gradient information. The constraint refinement aims to tighten the bounds of the approximation of activation functions. Neurify can process networks containing ReLU activation functions.

**nnenum.** The verifier proposed by Bak et al. [4] utilises star sets to represent the values each layer of a neural network can attain. By propagating these through the network, it checks whether one or more of the star sets results in an adversarial example. This verifier can handle networks with ReLU activation functions.

**VeriNet.** The verifier developed by Henriksen & Lomuscio [40] combines symbolic intervals with gradient-based adversarial local search for finding counter-examples. The authors further propose a splitting heuristic for interval propagation based on the influence of a given node on the bounds of the network output. VeriNet supports networks containing ReLU, Sigmoid and Tanh activation functions.

### 3.2.2 GPU-Based Methods

Next, we present the GPU-based verification algorithms we considered.

**BaDNB.** The BaDNB verifier introduced by DePalma et al. [22] builds on earlier versions of the BaB framework; however, it uses a novel dual formulation of the MIP, which it solves via branch-and-bound. The novel formulation allows for extensive parallelisation on GPUs. Furthermore, it employs a bounding heuristic which significantly reduces the number of branches necessary for solving the verification problem. BaDNB is limited to ReLU-based networks and MaxPooling operations.

**Beta-CROWN.** $\alpha\beta$-CROWN [111] is a bound propagation method combined with neuron-split constraints, which divides the original problem into sub-problems based on the activation function's range. $\alpha\beta$-CROWN leverages neuron-split constraints, while, in general, other bound propagation methods are not able to handle this type of constraint. Using the framework presented by Bunel et al. [12], the verifier is complete and can be efficiently parallelised using GPUs. $\alpha\beta$-CROWN can handle ReLU, Sigmoid and Tanh activations as well as MaxPooling layers.

**MN-BaB.** The MN-BaB verifier [29] builds on the multi-neuron constraints un-

**Table 3.2:** Instance set size for each benchmark category. Solvable instances are those solved by at least one (*i.e.*, any) or all of the considered verifiers. We considered any instance that was found to be *sat* or *unsat* as solved. The number of *sat* and *unsat* instances, respectively, can be found in brackets. The column "Verifiers employed" lists (1) BaBSB, (2) Marabou, (3) Neurify, (4) nnenum, (5) VeriNet, (6) BaDNB, (7) $\alpha\beta$-CROWN or (8) MN-BaB as the matching suitable algorithm(s) to the respective category.

| **CPU methods** | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Category | MNIST | | | | | CIFAR | | | | | Verifiers employed |
| | Total | Solvable | | | | Total | Solvable | | | | |
| | | Any | (*sat*/*unsat*) | All | (*sat*/*unsat*) | | Any | (*sat*/*unsat*) | All | (*sat*/*unsat*) | |
| ReLU | 2 500 | 1 913 | (169/1 744) | 42 | (38/4) | 2 500 | 972 | (946/26) | 0 | (0/0) | (1),(2),(3),(4),(5) |
| ReLU + MaxPool | 400 | 5 | (0/5) | 0 | (0/0) | 100 | 0 | (0/0) | 0 | (0/0) | (2) |
| Tanh | 600 | 556 | (29/527) | 0 | (0/0) | 600 | 0 | (0/0) | 0 | (0/0) | (5) |
| Sigmoid | 600 | 581 | (37/544) | 0 | (0/0) | 600 | 0 | (0/0) | 0 | (0/0) | (2),(5) |
| **GPU methods** | | | | | | | | | | | |
| ReLU | 2 500 | 2 308 | (128/2 180) | 948 | (53/895) | 2 500 | 2 364 | (2 262/102) | 1 048 | (1 048/0) | (6),(7),(8) |
| ReLU + MaxPool | 400 | 128 | (40/88) | 84 | (25/59) | 100 | 64 | (64/0) | 0 | (0/0) | (6),(7),(8) |
| Tanh | 600 | 319 | (28/291) | 0 | (0/0) | 600 | 497 | (494/3) | 0 | (0/0) | (7),(8) |
| Sigmoid | 600 | 307 | (35/272) | 304 | (0/0) | 600 | 547 | (481/66) | 0 | (0/0) | (7),(8) |

derlying the ERAN toolkit [85, 96, 99, 97, 98] as well as GPU-enabled linear bound propagation in a branch-and-bound framework. MN-BaB uses different verification modes, including input-domain splitting with bound propagation and full MIP encodings for complete verification. It is capable of handling various activation functions and layer operations such as ReLU, Sigmoid, Tanh, and MaxPooling.

## 3.3 Setup for Empirical Evaluation

In the following, we will present an overview of how we set up our benchmark study, *i.e.*, how we selected problem instances and verification algorithms. Furthermore, we will provide details on the software we used and the execution environment in which our experiments were carried out.

### 3.3.1 Problem Instances

For our assessment, we compiled a high-quality set of problem instances for local robustness verification. Following best practices in other research areas, such as optimisation [41, 5], the benchmark should be *representative* and *diverse*, where the former refers to how well the difficulty of the benchmark is aligned with that of real-world instances from the same problem class, and the latter means that the instance set should cover a wide range of difficulties.

Overall, our benchmark is comprised of 79 image classification networks, of which

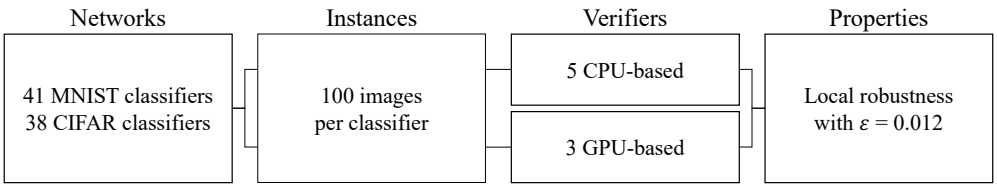



**Figure 3.1:** Schematic overview of the setup of experiments.

38 are trained on the CIFAR-10 dataset and 41 are trained on the MNIST dataset. To ensure the representativeness of our benchmark set, all networks were sampled from the neural network verification literature, *i.e.*, networks used in existing work on local robustness verification and provided in public repositories; in other words, the characteristics of the networks in our benchmark are assumed to match those of networks generally used for evaluating verification algorithms. We further want our instance set to be diverse. Therefore, we paid special attention to ensure that the networks we considered differ in size, *i.e.*, the number of hidden layers and nodes, as well as the type of non-linearities (*e.g.*, ReLU or Tanh) and layer operations (*e.g.*, pooling or convolutional layers) they employ. Notice that some of the networks we considered were also used in the 2022 VNN Competition. A full overview of the networks used in our study and their respective sources is provided in Table 3.3 and Table 3.3.

Of each network, we verified 100 local robustness properties; more precisely, we sampled 100 images from the dataset on which the network has been trained and verified for local robustness with the perturbation radius $\epsilon$ set at $\{0.004, 0.005, 0.008, 0.01, 0.012, 0.02, 0.025, 0.03, 0.04\}$. To avoid over-aggregation, we firstly focused our analysis on a single value of $\epsilon$, where $\epsilon = 0.012$, which represents a radius larger than $1/255$, the smallest $\epsilon$-ball distance used in existing literature [71], and centred around commonly chosen values for $\epsilon$ [114, 8, 109].

Lastly, we split our benchmark set into different categories based on verifier compatibilities. This means a verifier is only applied to categories it can process. The categories as well as the instance set size for each category are shown in Table 3.2. Notice that, in general, the ground truth for any given problem instance is not known a priori. At the same, even state-of-the-art verifiers are known to sometimes produce different results for the same instance [10]. As some of the considered verifiers do not return counterexamples by default, we treated these instance as *unsat*.

**Table 3.3:** Considered neural networks trained on the MNIST dataset, along with their training method, employed activation function and source repository.

| Network | Training | Activation | Source |
|---|---|---|---|
| cnn_max_mninst2[1] | Standard | ReLU | Marabou |
| cnn_max_mninst3[1] | Standard | ReLU | Marabou |
| convBig$_A$ | DiffAI | ReLU | ERAN |
| convMed$_A$ | PGD, $\epsilon = 0.1$ | ReLU | ERAN |
| convMed$_B$ | PGD, $\epsilon = 0.1$ | Sigmoid | ERAN |
| convMed$_C$ | PGD, $\epsilon = 0.1$ | Tanh | ERAN |
| convMed$_D$ | PGD, $\epsilon = 0.3$ | ReLU | ERAN |
| convMed$_E$ | PGD, $\epsilon = 0.3$ | Sigmoid | ERAN |
| convMed$_F$ | PGD, $\epsilon = 0.3$ | Tanh | ERAN |
| convMed$_G$ | Standard | ReLU | ERAN |
| convMed$_H$ | Standard | Sigmoid | ERAN |
| convMed$_I$ | Standard | Tanh | ERAN |
| convnet[1] | Standard | ReLU | ERAN |
| convSmall$_A$ | DiffAI | ReLU | ERAN |
| convSmall$_B$ | PGD | ReLU | ERAN |
| convSmall$_C$ | Standard | ReLU | ERAN |
| convSuper | DiffAI | ReLU | ERAN |
| ffnn_6×500$_A$ | PGD, $\epsilon = 0.1$ | ReLU | ERAN |
| ffnn_6×500$_B$ | PGD, $\epsilon = 0.1$ | Sigmoid | ERAN |
| ffnn_6×500$_C$ | PGD, $\epsilon = 0.1$ | Tanh | ERAN |
| ffnn_6×500$_D$ | PGD, $\epsilon = 0.3$ | ReLU | ERAN |
| ffnn_6×500$_E$ | PGD, $\epsilon = 0.3$ | Sigmoid | ERAN |
| ffnn_6×500$_F$ | PGD, $\epsilon = 0.3$ | Tanh | ERAN |
| ffnn_6×500$_G$ | Standard | ReLU | ERAN |
| ffnn_6×500$_H$ | Standard | Sigmoid | ERAN |
| ffnn_6×500$_I$ | Standard | Tanh | ERAN |
| mnist-net | Standard | ReLU | Venus |
| mnist-net_256×2 | Standard | ReLU | VNN-COMP |
| mnist-net_256×4 | Standard | ReLU | VNN-COMP |
| mnist-net_256×6 | Standard | ReLU | VNN-COMP |
| mnist_3×100 | Standard | ReLU | ERAN |
| mnist_3×50 | Standard | ReLU | ERAN |
| mnist_4×1024 | Standard | ReLU | ERAN |
| mnist_5×100 | Standard | ReLU | ERAN |
| mnist_6×100 | Standard | ReLU | ERAN |
| mnist_6×200 | Standard | ReLU | ERAN |
| mnist_9×100 | Standard | ReLU | ERAN |
| mnist_9×200 | Standard | ReLU | ERAN |
| mnist_conv[1] | Standard | ReLU | ERAN |
| mnist_nn | Standard | ReLU | VeriNet |
| rsl18a-linf01 | SDP | ReLU | MIPVerify |

[1]Employs MaxPooling layers

**Table 3.4:** Considered neural networks trained on the CIFAR-10 dataset, along with their training method, employed activation function and source repository.

| Network | Training | Activation | Source |
|---|---|---|---|
| cifar_base_kw | [113], $\epsilon = 1/255$ | ReLU | OVAL |
| cifar_deep_kw | [113], $\epsilon = 1/255$ | ReLU | OVAL |
| cifar_wide_kw | [113], $\epsilon = 1/255$ | ReLU | OVAL |
| cifar_base_kw_simp | [113], $\epsilon = 1/255$ | ReLU | Marabou |
| cifar_deep_kw_simp | [113], $\epsilon = 1/255$ | ReLU | Marabou |
| cifar_wide_kw_simp | [113], $\epsilon = 1/255$ | ReLU | Marabou |
| cifar-net | Standard | ReLU | Venus |
| cifar_conv[1] | Standard | ReLU | ERAN |
| cifar_4×100 | Standard | ReLU | ERAN |
| cifar_6×100 | Standard | ReLU | ERAN |
| cifar_7×1024 | Standard | ReLU | ERAN |
| cifar_9×200 | Standard | ReLU | ERAN |
| cifar_4×100 | Standard | ReLU | ERAN |
| cifar10_2_255 | COLT, $\epsilon = 2/255$ | ReLU | VNN-COMP |
| cifar10_8_255 | COLT, $\epsilon = 8/255$ | ReLU | VNN-COMP |
| cifar10_2_255_simplified | COLT, $\epsilon = 2/255$ | ReLU | VNN-COMP |
| cifar10_8_255_simplified | COLT, $\epsilon = 8/255$ | ReLU | VNN-COMP |
| convBig$_B$ | PGD, $\epsilon = 2/255$ | ReLU | ERAN |
| convMed$_J$ | PGD, $\epsilon = 2/255$ | ReLU | ERAN |
| convMed$_K$ | PGD, $\epsilon = 2/255$ | Sigmoid | ERAN |
| convMed$_L$ | PGD, $\epsilon = 2/255$ | Tanh | ERAN |
| convMed$_M$ | PGD, $\epsilon = 8/255$ | ReLU | ERAN |
| convMed$_N$ | PGD, $\epsilon = 8/255$ | Sigmoid | ERAN |
| convMed$_O$ | PGD, $\epsilon = 8/255$ | Tanh | ERAN |
| convMed$_P$ | Standard | ReLU | ERAN |
| convMed$_Q$ | Standard | Sigmoid | ERAN |
| convMed$_R$ | Standard | Tanh | ERAN |
| convSmall$_E$ | DiffAI | ReLU | ERAN |
| convSmall$_F$ | Standard | ReLU | ERAN |
| ffnn_6×500$_J$ | PGD, $\epsilon = 2/255$ | ReLU | ERAN |
| ffnn_6×500$_K$ | PGD, $\epsilon = 2/255$ | Sigmoid | ERAN |
| ffnn_6×500$_L$ | PGD, $\epsilon = 2/255$ | Tanh | ERAN |
| ffnn_6×500$_M$ | PGD, $\epsilon = 8/255$ | ReLU | ERAN |
| ffnn_6×500$_N$ | PGD, $\epsilon = 8/255$ | Sigmoid | ERAN |
| ffnn_6×500$_O$ | PGD, $\epsilon = 8/255$ | Tanh | ERAN |
| ffnn_6×500$_P$ | Standard | ReLU | ERAN |
| ffnn_6×500$_Q$ | Standard | Sigmoid | ERAN |
| ffnn_6×500$_R$ | Standard | Tanh | ERAN |

[1]Employs MaxPooling layers

### 3.3.2 Evaluation Metrics

In order to assess the performance of the various methods, we compute four performance metrics: the average running time, the number of solved instances, the *relative marginal contribution* and the *relative Shapley value* [33] of each verifier to the parallel portfolio containing all (applicable) verifiers. The first two of these reflect stand-alone performance, while the last two capture performance complementarity between verifiers and their contribution to the overall state of the art. Although these metrics present aggregated measures, they reflect algorithm performance on an instance level and in relation to other methods included in our comparison; a more detailed explanation will be provided in the following paragraphs. Notice that we do not penalise timeouts when computing average running time; *i.e.*, the maximum running time equals the given time limit.

The marginal contribution is computed as follows. Define $V$ as a set of verifiers and let $s(V)$ be the total score of set $V$. Here, the total score $s(V)$ consists of the number of instances verified by at least one verifier in set $V$ within a given cutoff time. We compute the marginal contribution per algorithm to determine how much the total performance of all algorithms (in terms of solved instances) decreases when the given algorithm is removed from the set of all algorithms if they were employed in a parallel algorithm portfolio. Formally, to determine the marginal contribution of any of the verifiers $v$ to portfolio $V$, one needs to know the value of $s(V)$ and $s(V \setminus \{v\})$, where $V \setminus \{v\}$ is the portfolio minus verifier $v$. Thus, the marginal contribution of verifier $v$ is expressed as

$$MC_v(V) = s(V) - s(V \setminus \{v\}) \tag{3.1}$$

Following this terminology, we can define the number of solved instances by verifier $v$ as a set consisting only of verifier $v$, $Solved_v = s(v) - s(\emptyset)$, where $s(\emptyset) = 0$. In other words, the number of solved instances employs a set of size one whereas the marginal contribution employs a set of all verifiers under consideration. The *relative* marginal contribution represents the marginal contribution of a given verifier as a fraction of the sum of every method's absolute marginal contribution.

Lastly, the Shapley value is the average marginal contribution of a verifier over all possible joining orders, where joining order refers to the order in which the verifiers are added to a parallel portfolio. This value complements the previous two metrics, as it does not assume a particular order in which algorithms are added to the portfolio. To be precise, the number of solved instances simply represents a joining order in which the considered algorithm comes first and in which it is the only one added to the

portfolio, whereas the marginal contribution metric assumes a joining order in which it comes last. However, using fixed orders, as is the case for the marginal contribution, might not reveal possible interactions between the given method and other algorithms, *e.g.*, it might understate the importance of a single algorithm given the presence of another algorithm with highly correlated performance. In such a case, both algorithms would be assigned very low marginal contribution, even though one of them should be included in a potential portfolio. Moreover, the fixed joining order leads to the marginal contribution metric being very sensitive to the composition of the portfolio in question; *i.e.*, this metric might change drastically if only a subset of methods would be included in a given portfolio.

This is captured by the Shapley value: Consider a set of verifiers $V$ of size $n$ (*i.e.*, $|V| = n$) and $\Pi^V$ as the set of all permutations of $V$. Notice that each permutation $\pi$ in $\Pi^V$ is of size $n$, which results in set $\Pi^V$ being of size $n!$. Now define $V_v^\pi$ as the set of verifiers where all verifiers joining after $v$ – *i.e.*, appearing after $v$ in permutation $\pi$ – are discarded from $\pi$. The Shapley value of verifier $v$, $\phi_v$, is then calculated as follows:

$$\phi_v(V) = \frac{1}{n!} \cdot \sum_{\pi \in \Pi^V} \left( s(V_v^\pi) - s(V_v^\pi \setminus \{v\}) \right) \tag{3.2}$$

The *relative* Shapley value of a verifier $v$ is obtained by dividing $\phi_v$ by the sum over the (absolute) Shapley values for all verifiers under consideration; it intuitively represents the fraction of the jointly achieved Shapley values over all verifiers that is attributed to verifier $v$.

### 3.3.3 Execution Environment and Software Used

Our experiments were carried out on a cluster of machines equipped with Intel Xeon E5-2683 CPUs with 32 cores, 40 MB cache size and 94 GB RAM, running CentOS Linux 7. Each verification method was limited to using a single CPU core per run. Each query (*i.e.*, attempt to solve a verification problem instance) was given a time budget of 3 600 seconds and a memory budget of 3 GB. Generally, we executed the verification algorithms through the DNNV interface, version 0.4.8. DNNV is a framework that transforms a network and robustness property into a unified format, which can then be solved by a given method [95]. More specifically, DNNV takes as input a network in the ONNX format, along with a property specification, and then translates the network and property to the input format required by the verifier. After running the verifier on the transformed problem, it returns the results in a standardised manner, where the

output is either *sat* if the property was falsified or *unsat* if the property was proven to hold. In cases where a violation is found, DNNV also returns a counter-example to the property and validates it by performing inference with the network. We note that for the VeriNet toolkit, its implementation in DNNV lags behind the standalone implementation of the verifier. While we acknowledge that this could affect observed performance, we still chose to run each CPU method through the DNNV interface to benefit from the broader benchmark support provided by DNNV.

For GPU-accelerated methods, we used machines equipped with NVIDIA GeForce GTX 1080 Ti GPUs with 11 GB video memory. We provided the same time budget but did not impose any memory constraints. The GPU-based methods we considered are not supported by DNNV. Hence, we used the standalone implementations of these algorithms through the $\alpha\beta$-CROWN[1], OVAL$-$BaB[2], and MN-BaB[3] framework, respectively. These methods also return a counter-example to the property in cases where a violation is found.

## 3.4 Results and Discussion

In the following, we provide an in-depth discussion of the results obtained from our experiments. We distinguish between CPU-based algorithms and algorithms that also utilise GPU resources. Table 3.2 shows the categories we devised based on layer types present in the network, along with the resulting instance set sizes as well as information on which verifier has been employed for each category. Moreover, we investigate whether there exists a single algorithm that performs best on all instances within a given category. If we find this to not be the case, we analyse to what extent the algorithms we considered complement each other in performance, *i.e.*, show strong performance on different problem instances.

### 3.4.1 CPU-Based Methods

Table 3.5 contains the results from our experiments using CPU-based verification algorithms. It reports the number of problem instances solved by each verifier per network category (see Table 3.2 for the total number of problem instances per category), the relative marginal contribution, the relative Shapley value and the average running time computed over the subset of *solvable* instances, *i.e.*, instances that could be solved

[1]Commit 7a46097192207dfbb2fa7135857d6bc4ae7d6cd5
[2]Commit 9e1606044759da5693f226ce489e9d4dded21bd6
[3]Commit 2aa12b145bb61342f4c464b64be3467b3a275e46

**Table 3.5:** Performance comparison of CPU-based verification algorithms in terms of the number of solved instances, relative marginal contribution (*RMC*), relative Shapley value ($\phi$) and CPU running time averaged per problem instance, computed for each category for $\epsilon = 0.012$.

| **ReLU** Verifier | MNIST | | | | CIFAR | | | |
|---|---|---|---|---|---|---|---|---|
| | Solved | *RMC* | $\phi$ | Avg. Time [CPU s] | Solved | *RMC* | $\phi$ | Avg. Time [CPU s] |
| BaBSB | 358 | 0.22 | 0.06 | 3 241 | 307 | 0.00 | 0.09 | 2 924 |
| Marabou | 1 001 | 0.19 | 0.16 | 1 801 | 400 | 0.00 | 0.12 | 2 153 |
| Neurify | 871 | 0.25 | 0.14 | 1 964 | 915 | 0.75 | 0.42 | 235 |
| nnenum | 1 754 | 0.17 | 0.31 | 389 | 76 | 0.05 | 0.03 | 3 337 |
| VeriNet | 1 799 | 0.16 | 0.32 | 263 | 841 | 0.20 | 0.34 | 500 |
| **ReLU+MaxPool** Verifier | MNIST | | | | CIFAR | | | |
| | Solved | *RMC* | $\phi$ | Avg. Time | Solved | *RMC* | $\phi$ | Avg. Time |
| Marabou | 5 | 1.00 | 1.00 | 57 | 0 | 0.00 | 0.00 | 3 600 |
| **Tanh** Verifier | MNIST | | | | CIFAR | | | |
| | Solved | *RMC* | $\phi$ | Avg. Time | Solved | *RMC* | $\phi$ | Avg. Time |
| VeriNet | 556 | 1.00 | 1.00 | 55 | 0 | 0.00 | 0.00 | 3 600 |
| **Sigmoid** Verifier | MNIST | | | | CIFAR | | | |
| | Solved | *RMC* | $\phi$ | Avg. Time | Solved | *RMC* | $\phi$ | Avg. Time |
| Marabou | 0 | 0.00 | 0.00 | 3 600 | 0 | 0.00 | 0.00 | 3 600 |
| VeriNet | 581 | 1.00 | 1.00 | 55 | 0 | 0.00 | 0.00 | 3 600 |

by at least one of the considered methods. The relative marginal contribution and the relative Shapley value are calculated based on the number of solved problem instances. We provide absolute values for both the marginal contribution and Shapley value in Table 3.6, 3.8, 3.10 and 3.12. Notice that instances that were not solved within the time limit were attributed the maximum running time, *i.e.*, 3 600 seconds.

On ReLU-based MNIST networks, we found VeriNet to be the best-performing verifier, solving 1 799 out of 2 500 instances, while achieving a relative Shapley value of 0.32. However, taking relative marginal contribution into account, we found that Neurify achieved the highest relative marginal contribution of 0.25 (compared to 0.16 for VeriNet), indicating that it could verify a sizable fraction of instances on which other methods failed to return a solution. Moreover, the relative marginal contribution scores show that each method could solve a sizeable fraction of instances unsolved by

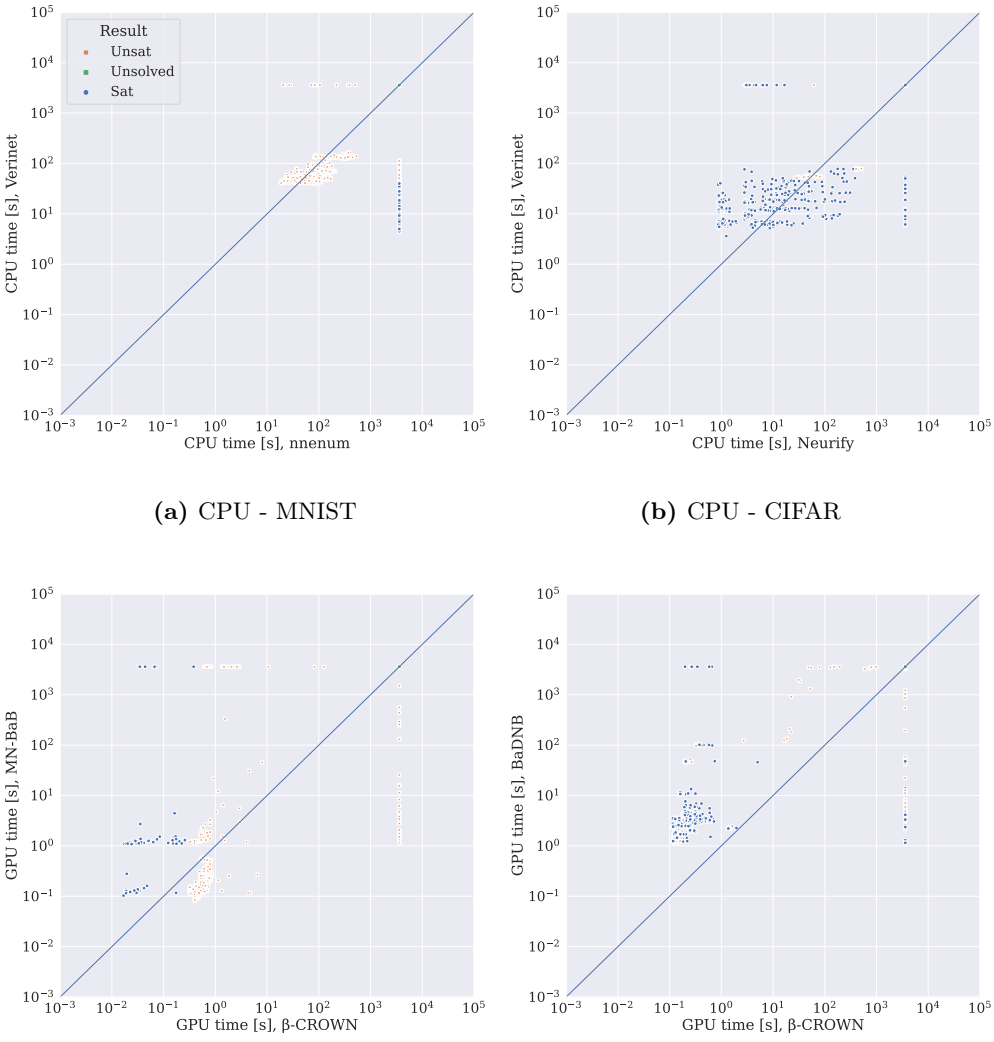
**Table 3.6:** Performance comparison of CPU-based verification algorithms in terms of the number of solved instances, absolute marginal contribution ($MC$), absolute Shapley value ($\phi_{abs}$) and CPU running time averaged per problem instance, computed for each category with $\epsilon$ set at 0.012.

| **ReLU** | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Verifier | MNIST | | | | CIFAR | | | |
| | Solved | $MC$ | $\phi_{abs}$ | Avg. Time [CPU s] | Solved | $MC$ | $\phi_{abs}$ | Avg. Time [CPU s] |
| BaBSB | 358 | 23 | 118 | 3 241 | 307 | 0 | 86 | 2 924 |
| Marabou | 1 001 | 20 | 312 | 1 801 | 400 | 0 | 117 | 2 153 |
| Neurify | 871 | 26 | 265 | 1 964 | 915 | 119 | 411 | 235 |
| nnenum | 1 754 | 18 | 600 | 389 | 76 | 8 | 28 | 3 337 |
| VeriNet | 1 799 | 16 | 618 | 263 | 841 | 31 | 330 | 500 |
| **ReLU+MaxPool** | | | | | | | | |
| Verifier | MNIST | | | | CIFAR | | | |
| | Solved | $MC$ | $\phi_{abs}$ | Avg. Time | Solved | $MC$ | $\phi_{abs}$ | Avg. Time |
| Marabou | 5 | 5 | 5 | 57 | 0 | 0 | 0 | 3 600 |
| **Tanh** | | | | | | | | |
| Verifier | MNIST | | | | CIFAR | | | |
| | Solved | $MC$ | $\phi_{abs}$ | Avg. Time | Solved | $MC$ | $\phi_{abs}$ | Avg. Time |
| VeriNet | 556 | 556 | 556 | 55 | 0 | 0 | 0 | 3 600 |
| **Sigmoid** | | | | | | | | |
| Verifier | MNIST | | | | CIFAR | | | |
| | Solved | $MC$ | $\phi_{abs}$ | Avg. Time | Solved | $MC$ | $\phi_{abs}$ | Avg. Time |
| Marabou | 0 | 0 | 0 | 3 600 | 0 | 0 | 0 | 3 600 |
| VeriNet | 581 | 581 | 581 | 55 | 0 | 0 | 0 | 3 600 |

any other method.

On ReLU-based CIFAR networks, it should first be noted that there is no verification problem instance that can be solved by *all* verifiers, highlighting the structural differences between instances and the sensitivity of the verification approaches to those differences. That said, Neurify slightly outperformed VeriNet in terms of the number of solved instances (915 *vs* 841 out of 2 500). Furthermore, Neurify achieved a much larger relative marginal contribution than VeriNet (0.75 *vs* 0.20), which means that the former could solve a relatively large number of instances which could not be solved by the other methods. Generally, relative marginal contribution scores are much less evenly distributed among verifiers when compared to the MNIST dataset.

Figure 3.2a and 3.2b show an instance-level comparison of the two best-performing algorithms (in terms of relative Shapley value) in the ReLU category for each dataset.




**(a)** CPU - MNIST

**(b)** CPU - CIFAR

**(c)** GPU - MNIST

**(d)** GPU - CIFAR

**Figure 3.2:** Performance comparison of the two top-performing verification methods (in terms of relative Shapley value) in the ReLU category for CPU-based methods on (a) MNIST and (b) CIFAR networks as well GPU-based methods on (c) MNIST and (d) CIFAR networks. Each data point represents an instance, and its position on a given axis represents the performance in terms of running time of the respective solver. The diagonal line represents the point on which both verifiers perform equally well. The verifier represented on the x-axis performs better on instances above the diagonal line, and the verifier represented on the y-axis performs better on instances below the diagonal line. Instances that were not solved within the time limit are displayed with the maximum running time (*i.e.*, 3 600 seconds).

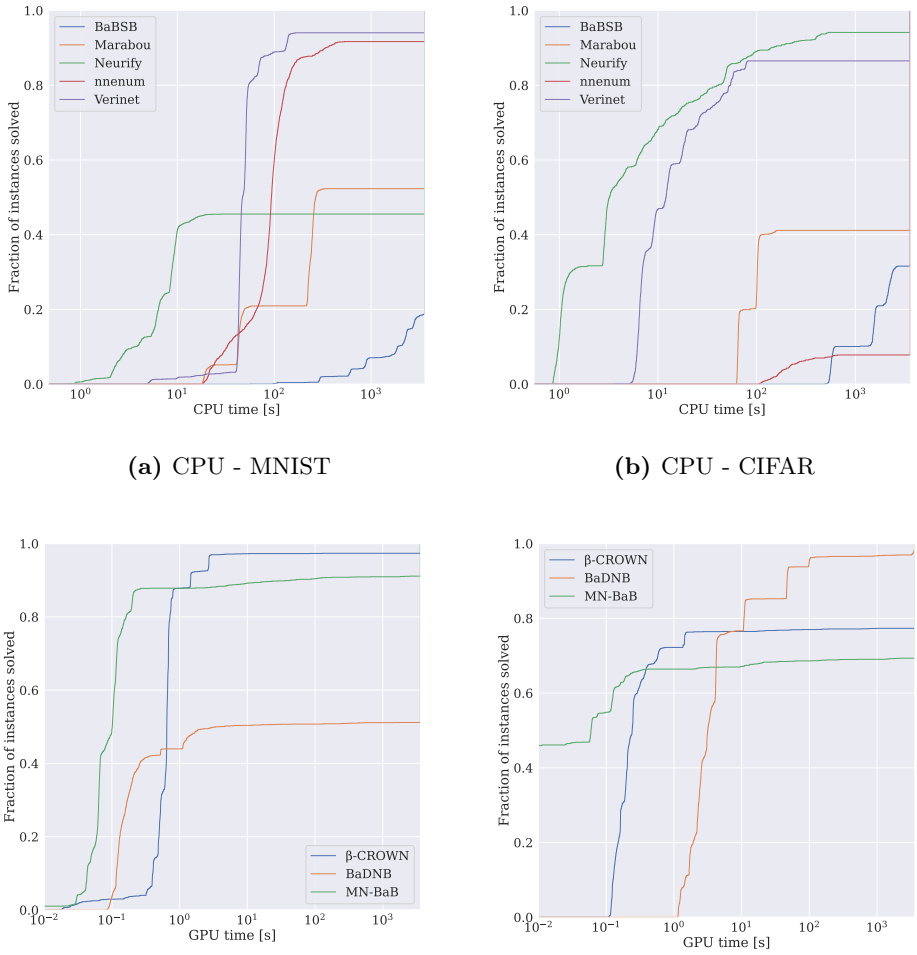
In Figure 3.2a, we see that on MNIST networks, both VeriNet and nnenum solved instances that the other one, in turn, could not solve within the given time budget. Concretely, when considering a parallel portfolio containing both algorithms (see Section 2.4), the number of solved instances slightly increases to 1 817 out of 2 500 (*vs* 1 799 solved by VeriNet and 1 754 solved by nnenum alone), while supplied with similar CPU resources (*i.e.*, 1 800 CPU seconds per verifier, adding up to the same combined maximum running time as running a single verifier with 3 600 CPU seconds).

On CIFAR instances, we found Neurify and VeriNet to also have distinct strengths over each other. This is shown in Figure 3.2b, where both algorithms could solve a substantial amount of instances that the other could not return a solution for. Thus, when combined in a parallel portfolio, 963 instances can be solved (*vs* 915 solved by Neurify and 841 solved by VeriNet alone, out of 2 500 instances), while using the same amount of CPU resources, *i.e.*, 1 800 CPU seconds per verifier. These findings further emphasise the complementarity between the verification algorithms considered in our study. All remaining verifiers achieved much lower relative Shapley values and relative marginal contribution scores, indicating that they would not substantially strengthen the performance of a portfolio already containing Neurify and VeriNet.

Figure 3.3a shows the cumulative distribution function of running times over the MNIST problem instances. As seen in the figure, VeriNet tends to solve these problem instances fastest; however, Neurify tended to show even better performances on those instances it was able to solve. We note that most of the instances unsolved by Neurify represent networks that were trained on images with 3 dimensions, whereas Neurify requires images used as network inputs to have 2 or 4 dimensions.

Figure 3.3b shows a similar plot for the CIFAR problem instances. Here, Neurify solved the largest fraction in less time than other methods. This suggests that Neurify is a very competitive verifier when applicable to the specific network or input format.

For each of the remaining categories, we found that there is only one verifier that could effectively handle the respective problem instances. Specifically, instances from the ReLU+MaxPooling category can be processed by Marabou, although, only a modest number of MNIST instances could be solved in this way. Networks containing Tanh activation functions can, in principle, be verified by VeriNet but the algorithm did nonetheless not solve any CIFAR instances. Lastly, Sigmoid-based networks can be handled by both VeriNet and Marabou, however, only the former could solve MNIST instances within the given time and memory budget.




**(a)** CPU - MNIST

**(b)** CPU - CIFAR

**(c)** GPU - MNIST

**(d)** GPU - CIFAR

**Figure 3.3:** Cumulative distribution of the fraction of instances solved by the considered verification algorithms in the ReLU category as a function of CPU running time. The plots at the top are for CPU-based algorithms, whereas those at the bottom are for GPU-based algorithms, on MNIST and CIFAR.

### 3.4.2 GPU-Based Methods

Table 3.7 summarises the results from our experiments using GPU-based verification algorithms. On ReLU-based MNIST networks, $\alpha\beta$-CROWN outperformed other methods in terms of both the number of solved problem instances as well as the average

**Table 3.7:** Performance comparison of GPU-based verification algorithms in terms of the number of solved instances, relative marginal contribution ($RMC$), relative Shapley value ($\phi$) and average GPU running time, computed for each category for $\epsilon = 0.012$.

| **ReLU** | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Verifier | MNIST | | | | CIFAR | | | |
| | Solved | $RMC$ | $\phi$ | Avg. Time [GPU s] | Solved | $RMC$ | $\phi$ | Avg. Time [GPU s] |
| BaDNB | 1 188 | 0.31 | 0.19 | 1 760 | 2 332 | 0.90 | 0.45 | 116 |
| $\beta$-CROWN | 2 247 | 0.00 | 0.42 | 96 | 1 828 | 0.03 | 0.29 | 814 |
| MN-BaB | 2 103 | 0.69 | 0.39 | 325 | 1 639 | 0.07 | 0.26 | 1 110 |
| **ReLU+MaxPool** | | | | | | | | |
| Verifier | MNIST | | | | CIFAR | | | |
| | Solved | $RMC$ | $\phi$ | Avg. Time | Solved | $RMC$ | $\phi$ | Avg. Time |
| BaDNB | 85 | 0.00 | 0.22 | 1 399 | 0 | 0.00 | 0.00 | 3 600 |
| $\beta$-CROWN | 128 | 1.00 | 0.44 | 0.4 | 0 | 0.00 | 0.00 | 3 600 |
| MN-BaB | 115 | 0.00 | 0.34 | 366 | 64 | 1.00 | 1.00 | 0.008 |
| **Tanh** | | | | | | | | |
| Verifier | MNIST | | | | CIFAR | | | |
| | Solved | $RMC$ | $\phi$ | Avg. Time | Solved | $RMC$ | $\phi$ | Avg. Time |
| $\beta$-CROWN | 319 | 1.00 | 1.00 | 1.16 | 497 | 1.00 | 1.00 | 0.70 |
| MN-BaB | 0 | 0.00 | 0.00 | 3 600 | 0 | 0.00 | 0.00 | 3 600 |
| **Sigmoid** | | | | | | | | |
| Verifier | MNIST | | | | CIFAR | | | |
| | Solved | $RMC$ | $\phi$ | Avg. Time | Solved | $RMC$ | $\phi$ | Avg. Time |
| $\beta$-CROWN | 306 | 0.66 | 0.50 | 13 | 538 | 0.95 | 0.68 | 60 |
| MN-BaB | 305 | 0.33 | 0.50 | 24 | 338 | 0.05 | 0.32 | 1 376 |

running time. At the same time, the relative Shapley values of $\alpha\beta$-CROWN and MN-BaB indicate that these methods complement each other with respect to their performance on this instance set.

On ReLU-based CIFAR networks, Table 3.7 shows that BaDNB outperformed both MN-BaB and $\alpha\beta$-CROWN, with the former solving 2 332 and the latter solving 1 639 and 1 828 out of 2 500 verification problem instances, respectively. Furthermore, both BaDNB and $\alpha\beta$-CROWN achieve large relative Shapley values, suggesting their complementarity in an algorithm portfolio.

Figure 3.2c and 3.2d show the instance-level comparison of the two best-performing algorithms (in terms of relative Shapley value) in the ReLU category for each dataset. Looking at Figure 3.2c, one can see that there is a fairly large number of MNIST instances unsolved by $\alpha\beta$-CROWN but solved by MN-BaB as well as the other way

**Table 3.8:** Performance comparison of GPU-based verification algorithms in terms of the number of solved instances, absolute marginal contribution ($MC$), absolute Shapley value ($\phi_{abs}$) and average GPU running time, computed for each category with $\epsilon$ set at 0.012.

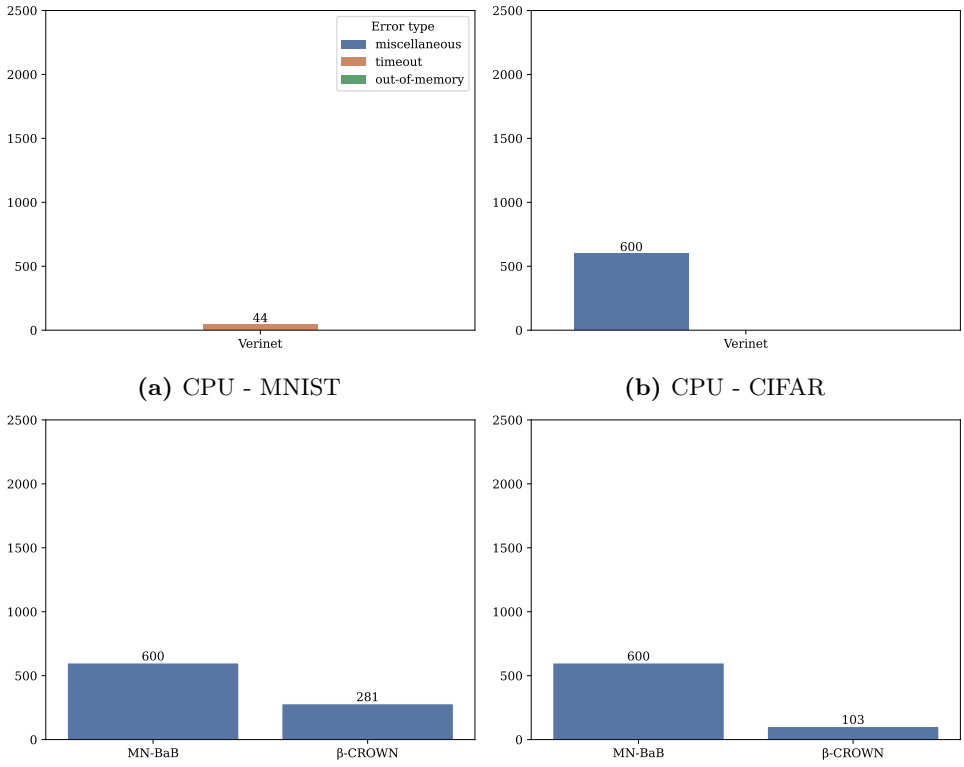
| **ReLU** | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Verifier | MNIST | | | | CIFAR | | | |
| | Solved | $MC$ | $\phi_{abs}$ | Avg. Time [GPU s] | Solved | $MC$ | $\phi_{abs}$ | Avg. Time [GPU s] |
| BaDNB | 1 188 | 8 | 440 | 1 760 | 2 332 | 250 | 1 066 | 116 |
| $\beta$-CROWN | 2 247 | 0 | 966 | 96 | 1 828 | 7 | 693 | 818 |
| MN-BaB | 2 103 | 18 | 903 | 325 | 1 639 | 20 | 604 | 1 110 |
| **ReLU+MaxPool** | | | | | | | | |
| Verifier | MNIST | | | | CIFAR | | | |
| | Solved | $MC$ | $\phi_{abs}$ | Avg. Time | Solved | $MC$ | $\phi_{abs}$ | Avg. Time |
| BaDNB | 85 | 0 | 29 | 1 399 | 0 | 0 | 0 | 3 600 |
| $\beta$-CROWN | 128 | 12 | 56 | 0.4 | 0 | 0 | 0 | 3 600 |
| MN-BaB | 115 | 0 | 44 | 366 | 64 | 64 | 64 | 0.008 |
| **Tanh** | | | | | | | | |
| Verifier | MNIST | | | | CIFAR | | | |
| | Solved | $MC$ | $\phi_{abs}$ | Avg. Time | Solved | $MC$ | $\phi_{abs}$ | Avg. Time |
| $\beta$-CROWN | 319 | 319 | 319 | 1.16 | 497 | 496 | 497 | 0.70 |
| MN-BaB | 0 | 0 | 0 | 3 600 | 0 | 0 | 0 | 3 600 |
| **Sigmoid** | | | | | | | | |
| Verifier | MNIST | | | | CIFAR | | | |
| | Solved | $MC$ | $\phi_{abs}$ | Avg. Time | Solved | $MC$ | $\phi_{abs}$ | Avg. Time |
| $\beta$-CROWN | 306 | 2 | 154 | 13 | 538 | 209 | 374 | 60 |
| MN-BaB | 305 | 1 | 153 | 24 | 338 | 9 | 174 | 1 376 |

around.

On the other hand, BaDNB and $\alpha\beta$-CROWN seem to have distinctive strengths over each other on CIFAR instances, as can be seen in Figure 3.2d: The data points indicating performance on each verification instance are spread out widely around the line of equal performance, showing that there are many instances that one method can solve faster than the other and vice versa.

Concurrently, MN-BaB solves a large fraction of CIFAR instances in less time than other methods, although BaDNB solves more instances overall, which is also reflected in Figure 3.3d. On MNIST instances, MN-BaB solves more instances in less time than $\alpha\beta$-CROWN, although $\alpha\beta$-CROWN solves more instances overall; see also Figure 3.3c.

On MNIST networks containing ReLU activation functions and MaxPooling operations, we again found relatively large Shapley values for both $\alpha\beta$-CROWN and




**(a)** CPU - MNIST **(b)** CPU - CIFAR

**(c)** GPU - MNIST **(d)** GPU - CIFAR

**Figure 3.4:** Frequency of error types returned by the considered verification algorithms on instances in the Tanh category. The total number of instances in this category is 600 for MNIST and 600 for CIFAR.

MN-BaB, as presented in Table 3.7, indicating their potential complementarity in an algorithm portfolio. However, the relative marginal contribution values indicate that there are no instances unsolved by $\alpha\beta$-CROWN that could be solved by other methods. CIFAR instances in this category could only be verified by MN-BaB, due to verifier incompatibilities with the respective network structures unrelated to the MaxPooling operations.

Table 3.7 further shows results for the Tanh category. We found that instances in this category could effectively only be handled the $\alpha\beta$-CROWN verifier. Concretely, MN-BaB returned an error for the instances in this category; see also Figure 3.4 for additional details.

Lastly, networks containing Sigmoid activation functions can be handled by both

BaDNB and $\alpha\beta$-CROWN and achieve perfectly similar relative Shapley values on the MNIST instances in this category, indicating their complementarity in an algorithm portfolio. However, as seen in Table 3.7, this does not hold for CIFAR instances, where $\alpha\beta$-CROWN seems to dominate in performance.

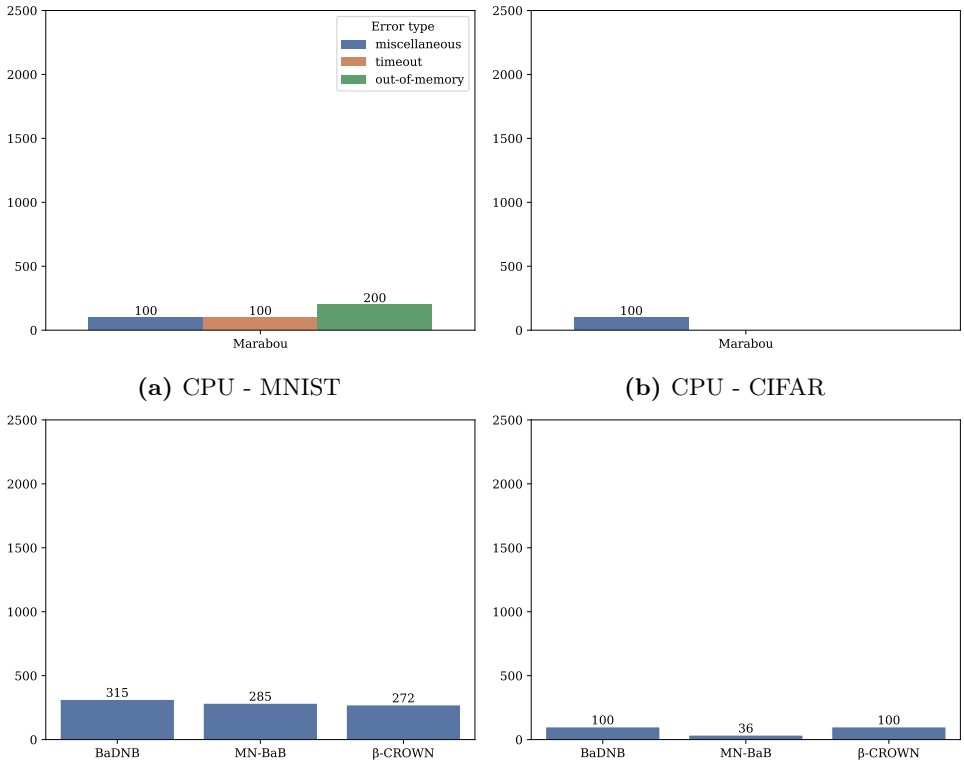
### 3.4.3 Error Analysis

Although the verification methods should, in principle, be able to solve the instances in the category they are applied to, we found many instances left unsolved, not only due to time or memory constraints but also due to other, unexpected issues. Hence, to understand better why certain instances could not be solved by a given verifier, we categorised and counted the errors returned by each verification system. For this analysis, we focused on instances in the ReLU category; results for the remaining categories are presented in Figure 3.5 and 3.6.

The number of instances solved by each method can be found in Table 3.5 for CPU- and Table 3.7 for GPU-based algorithms. The total number of instances in the ReLU category is 2 500 for MNIST and CIFAR, respectively. We distinguish between timeouts, out-of-memory and *miscellaneous* errors, where the latter includes verifier-specific errors of which most are undefined and not trivial to resolve, especially without in-depth knowledge of the verifier at hand.

Figure 3.7a and 3.7b show the errors returned by CPU-based methods for MNIST and CIFAR instances, respectively. On MNIST, most verifiers failed to solve a given instance due to timeouts, except for nnenum, which mostly ran into memory issues, and Neurify, which requires images used as network inputs to have 2 or 4 dimensions, as mentioned in Section 5.1. Notice that when supplied with a larger memory budget, nnenum could not solve substantially more instances, but produced a comparably large number of timeouts instead; more details can be found in Figure 3.8.

Interestingly, we made different observations with regard to the CIFAR instances. Here, each method mostly returned errors related to the network structure (or undefined errors). Besides this, nnenum again failed to verify a sizable fraction of instances due to memory limitations. Overall, we found CIFAR networks to be much less supported by the CPU-based methods we considered (as implemented in the DNNV framework) than MNIST networks, arguably due to the increased complexity of the former. We note that some of these errors could potentially be circumvented by resorting to the standalone implementations of the respective verifiers. However, overall, DNNV provides the broadest support for different network structures and operations [95].




**(a)** CPU - MNIST **(b)** CPU - CIFAR

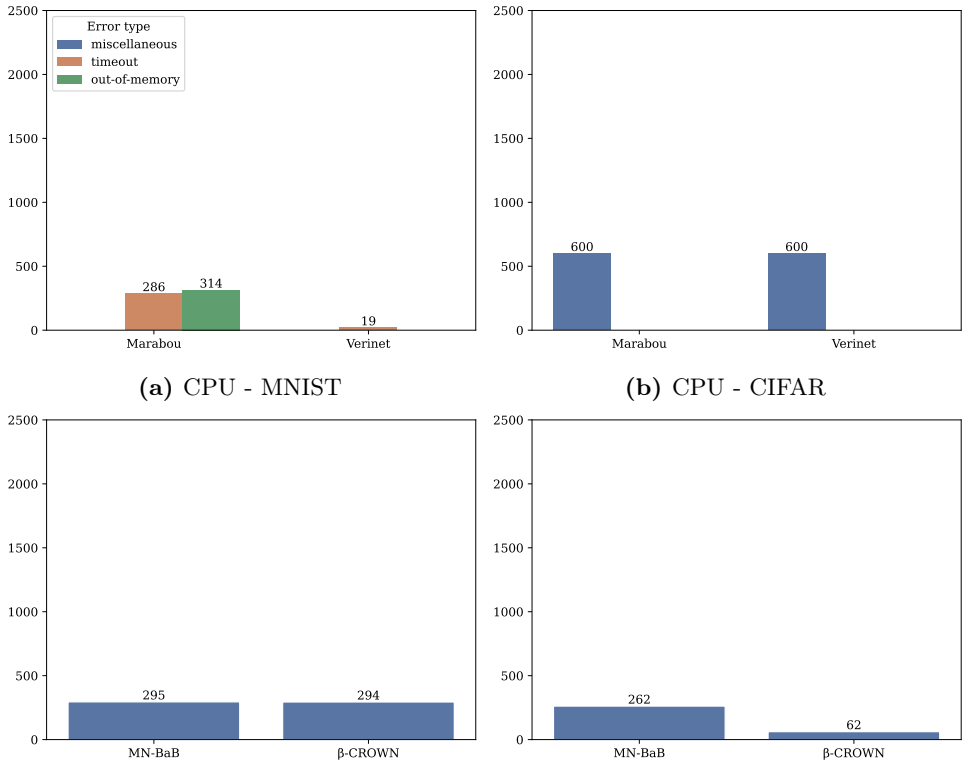
**(c)** GPU - MNIST **(d)** GPU - CIFAR

**Figure 3.5:** Frequency of error types returned by the considered verification algorithms on instances in the ReLU+MaxPool category. The total number of instances in this category is 400 for MNIST and 100 for CIFAR.

On the other hand, GPU-based verifiers show greater support for the considered networks than CPU-based methods. As seen in Figure 3.7c, only BaDNB failed to solve a relatively large number of MNIST instances due to unsupported network structures or other, unspecified technical reasons.

In contrast, BaDNB could solve almost all CIFAR instances, as shown in Figure 3.7d. However, both MN-BaB and $\alpha\beta$-CROWN returned several errors of which most are undefined.

Overall, our results suggest that many verification toolkits only support a limited set of networks. This occurs despite the fact that these networks are provided in onnx format, which should, in principle, be supported by each method considered in this study. Similar findings have been reported in the literature (see, *e.g.*, [80]).




**(a)** CPU - MNIST **(b)** CPU - CIFAR

**(c)** GPU - MNIST **(d)** GPU - CIFAR

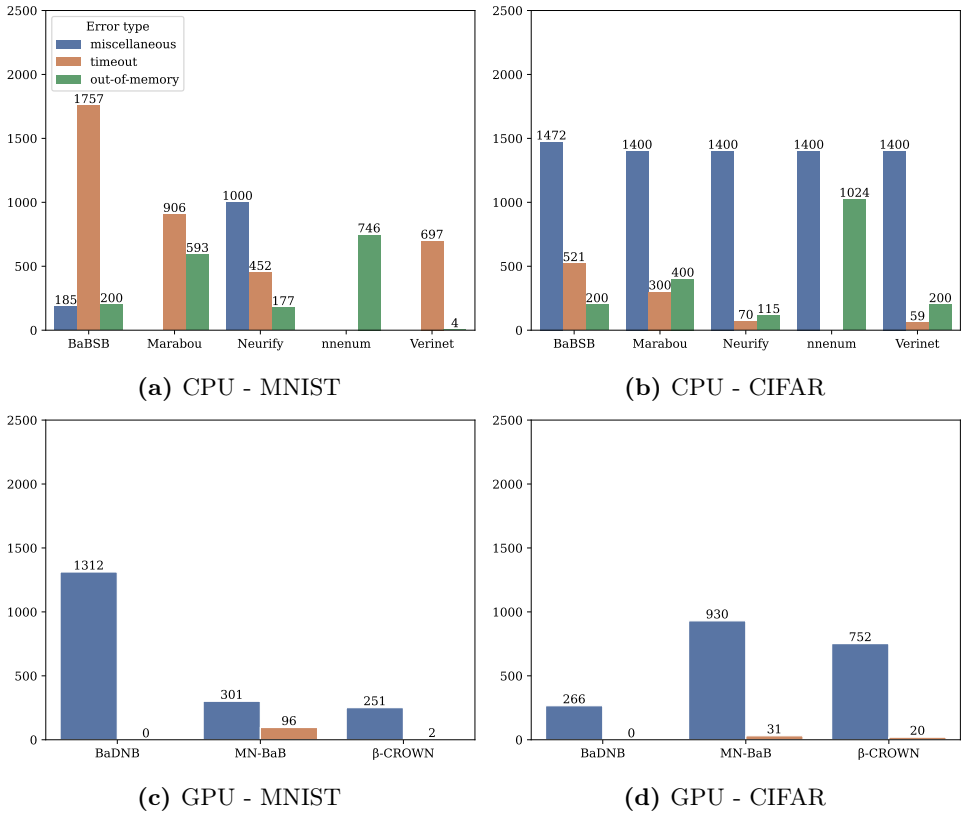
**Figure 3.6:** Frequency of error types returned by the considered verification algorithms on instances in the Sigmoid category. The total number of instances in this category is 600 for MNIST and 600 for CIFAR.

### 3.4.4 Analysis on Broader Set of Perturbation Radii

So far, we have considered a single value of $\epsilon$, but it stands to reason that changing the perturbation radius may affect algorithm behaviour. Therefore, we conducted further analysis on a broader set of perturbation radii, *i.e.*, with $\epsilon$ set to values of 0.004, 0.005, 0.008, 0.01, 0.012, 0.02, 0.025, 0.03 and 0.04.

Table 3.9 shows the results for the CPU-based algorithms on this extended set of problem instances. Overall, we found VeriNet remains the best-performing CPU-based verifier (in terms of solved instances and relative Shapley value) on ReLU-based MNIST networks. With regard to ReLU-based CIFAR networks, Table 3.9 shows that, overall, Neurify remained the best-performing CPU-based method.

However, we observed substantial differences between small and large values of $\epsilon$ in



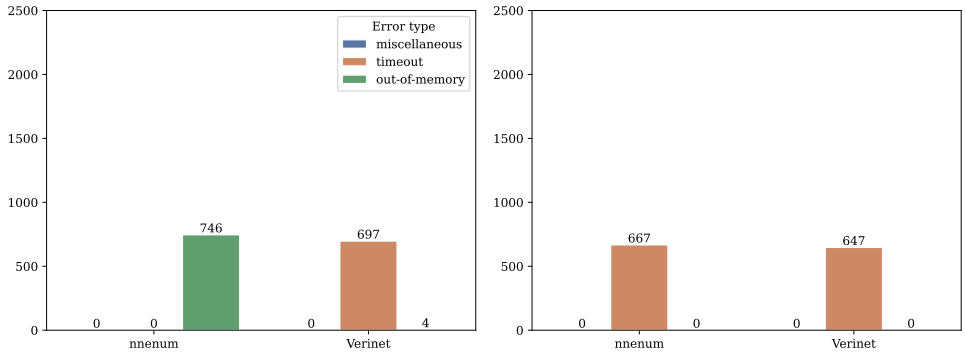

**(a)** CPU - MNIST **(b)** CPU - CIFAR

**(c)** GPU - MNIST **(d)** GPU - CIFAR

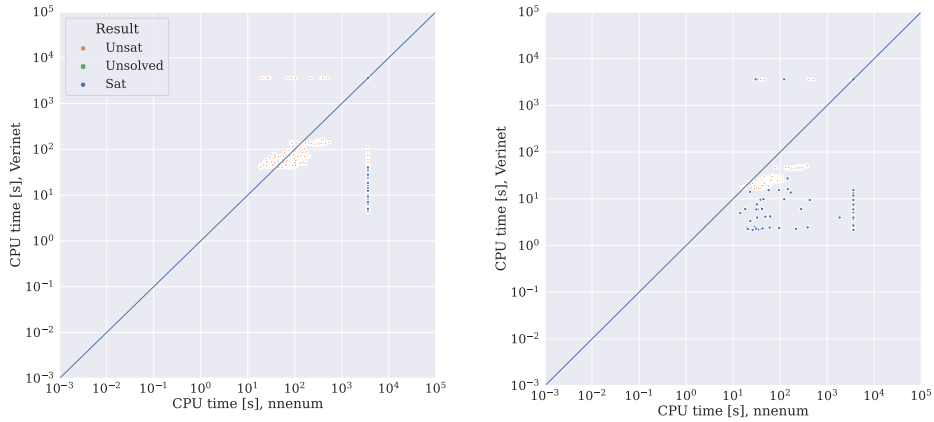
**Figure 3.7:** Frequency of error types returned by the considered verification algorithms on instances in the ReLU category.

the relative marginal contribution for each algorithm. More precisely, we analysed the relative marginal contribution of each verification algorithm for every given value of $\epsilon$ and show this in Figure 3.9c. Interestingly, one can see how the relative marginal contribution of Marabou steeply increases for increasingly larger epsilons, while that of other methods declines. Similarly, the solved instances and relative Shapley value achieved by each method changes as the perturbation radius varies; this is visualised in Figure 3.9a and 3.9e. In terms of both metrics, Marabou is strongly outperformed by most of the other algorithms for small values of $\epsilon$ but ends up achieving competitive or even better performance when $\epsilon$ is large.

An analogous investigation for CIFAR is shown in Figure 3.9d. In contrast to MNIST, one can see that the relative marginal contribution of each method is relatively weakly affected by the perturbation radius and, except for some divergence around




**(a)** Memory limit = 3GB **(b)** Memory limit = 30GB




**(c)** Memory limit = 3GB **(d)** Memory limit = 30GB

**Figure 3.8: Top row**: Frequency of error types returned by the two top-performing verification methods (in terms of the number of solved instances) on instances in the ReLU category, with a memory limit of (a) 3GB or (b) 30GB. **Bottom row**: Performance comparison of the two top-performing verification methods (in terms of relative Shapley value) in the ReLU category for CPU-based methods, with a memory limit of (c) 3GB or (d) 30GB.

$\epsilon = 0.005$, remains at a stable level. This holds for both solved instances and relative Shapley value as shown in Figure 3.9b and 3.9f.

We performed a similar analysis for GPU-based methods and present results, aggregated over all values of $\epsilon$, in Table 3.11. Among these algorithms, $\alpha\beta$-CROWN performed best on MNIST networks in the ReLU category, while BaDNB performed best on CIFAR networks in the same category. However, we found that the relative marginal contribution of each algorithm for every considered value of $\epsilon$ differs substantially

**Table 3.9:** Performance comparison of CPU-based verification algorithms in terms of the number of solved instances, relative marginal contribution ($RMC$), relative Shapley value ($\phi$) and average CPU running time, computed for each category and $\epsilon \in \{0.004, 0.005, 0.008, 0.01, 0.012, 0.02, 0.025, 0.03, 0.04\}$.

| **ReLU** | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Verifier | MNIST | | | | CIFAR | | | |
| | Solved | $RMC$ | $\phi$ | Avg. Time [CPU s] | Solved | $RMC$ | $\phi$ | Avg. Time [CPU s] |
| BaBSB | 3 716 | 0.06 | 0.06 | 3 223 | 2 690 | 0.00 | 0.09 | 2 964 |
| Marabou | 9 457 | 0.44 | 0.19 | 1 721 | 3 651 | 0.01 | 0.12 | 2 145 |
| Neurify | 8 206 | 0.12 | 0.14 | 1 899 | 8 173 | 0.71 | 0.41 | 289 |
| nnenum | 15 144 | 0.16 | 0.30 | 543 | 744 | 0.04 | 0.03 | 3 315 |
| VeriNet | 15 800 | 0.23 | 0.32 | 367 | 7 674 | 0.24 | 0.35 | 486 |
| **ReLU+MaxPool** | | | | | | | | |
| Verifier | MNIST | | | | CIFAR | | | |
| | Solved | $RMC$ | $\phi$ | Avg. Time | Solved | $RMC$ | $\phi$ | Avg. Time |
| Marabou | 316 | 1.00 | 1.00 | 50 | 0 | 0.00 | 0.00 | 3 600 |
| **Tanh** | | | | | | | | |
| Verifier | MNIST | | | | CIFAR | | | |
| | Solved | $RMC$ | $\phi$ | Avg. Time | Solved | $RMC$ | $\phi$ | Avg. Time |
| VeriNet | 4 307 | 1.00 | 1.00 | 59 | 0 | 0.00 | 0.00 | 3 600 |
| **Sigmoid** | | | | | | | | |
| Verifier | MNIST | | | | CIFAR | | | |
| | Solved | $RMC$ | $\phi$ | Avg. Time | Solved | $RMC$ | $\phi$ | Avg. Time |
| Marabou | 0 | 0.00 | 0.00 | 3 600 | 0 | 0.00 | 0.00 | 3 600 |
| VeriNet | 4 728 | 1.00 | 1.00 | 59 | 0 | 0.00 | 0.00 | 3 600 |

between small and large values of $\epsilon$ on MNIST instances, as shown in Figure 3.10c. For example, when $\epsilon = 0.02$, BaDNB and MN-BaB both achieve relative marginal contribution scores close to 0.5 but then strongly converge as $\epsilon$ becomes larger. Notably, these changes are not reflected in the relative Shapley values achieved by each method, where $\alpha\beta$-CROWN and MN-BaB both reach values close to 0.40 for every value of $\epsilon$; see Figure 3.10e for more details.

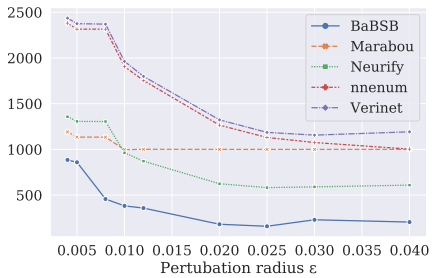
On CIFAR instances, Figure 3.10d indicates that the relative marginal contribution scores are only marginally affected by the chosen perturbation radius. More precisely, BaDNB achieves the largest relative marginal contribution for every value of $\epsilon$, while the relative marginal contributions of $\alpha\beta$-CROWN and MN-BaB only change slightly as the perturbation radius increases. At the same time, the observed Shapley values are mostly stable with regard to the perturbation radius, as shown in Figure 3.10f.

**Table 3.10:** Performance comparison of CPU-based verification algorithms in terms of the number of solved instances, absolute marginal contribution ($MC$), absolute Shapley value ($\phi_{abs}$) and average CPU running time, computed for each category with aggregated $\epsilon \in \{0.004, 0.005, 0.008, 0.01, 0.012, 0.02, 0.025, 0.03, 0.04\}$.

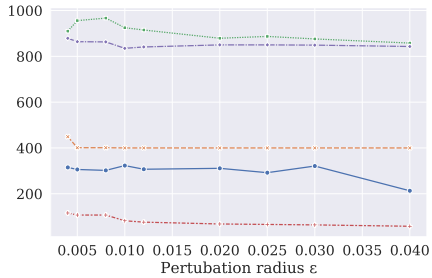
| **ReLU** | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Verifier | MNIST | | | | CIFAR | | | |
| | Solved | $MC$ | $\phi_{abs}$ | Avg. Time [CPU s] | Solved | $MC$ | $\phi_{abs}$ | Avg. Time [CPU s] |
| BaBSB | 3 716 | 103 | 1 062 | 3 223 | 2 690 | 0 | 759 | 2 964 |
| Marabou | 9 457 | 784 | 3 309 | 1 721 | 3 651 | 8 | 1 078 | 2 145 |
| Neurify | 8 206 | 212 | 2 418 | 1 899 | 8 173 | 1 059 | 3 662 | 289 |
| nnenum | 15 144 | 288 | 5 093 | 543 | 744 | 61 | 268 | 3 315 |
| VeriNet | 15 800 | 411 | 5 442 | 367 | 7 674 | 365 | 3 061 | 486 |
| **ReLU+MaxPool** | | | | | | | | |
| Verifier | MNIST | | | | CIFAR | | | |
| | Solved | $MC$ | $\phi_{abs}$ | Avg. Time | Solved | $MC$ | $\phi_{abs}$ | Avg. Time |
| Marabou | 316 | 316 | 316 | 50 | 0 | 0 | 0 | 3 600 |
| **Tanh** | | | | | | | | |
| Verifier | MNIST | | | | CIFAR | | | |
| | Solved | $MC$ | $\phi_{abs}$ | Avg. Time | Solved | $MC$ | $\phi_{abs}$ | Avg. Time |
| VeriNet | 4 307 | 4 307 | 4 307 | 59 | 0 | 0 | 0 | 3 600 |
| **Sigmoid** | | | | | | | | |
| Verifier | MNIST | | | | CIFAR | | | |
| | Solved | $MC$ | $\phi_{abs}$ | Avg. Time | Solved | $MC$ | $\phi_{abs}$ | Avg. Time |
| Marabou | 0 | 0 | 0 | 3 600 | 0 | 0 | 0 | 3 600 |
| VeriNet | 4 728 | 4 728 | 4 728 | 59 | 0 | 0 | 0 | 3 600 |

Lastly, we again compared the performance of the two best-performing CPU as well as GPU methods on an instance-level for all MNIST and CIFAR networks, respectively, from the ReLU category and show the results in Figure 3.11. In each case, we found that one method could solve some instances that were unsolved by the other, irrespective of the perturbation radius. Notice that our findings hold even for a much larger value of $\epsilon$. Specifically, we ran the two best-performing CPU-based algorithms, nnenum and VeriNet, on the MNIST instances for $\epsilon = 0.2$ and present the results in Figure 3.12.

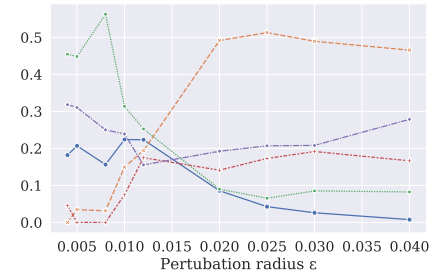
Overall, this clearly demonstrates that our observation of performance complementarity between verification algorithms holds for a broad range of perturbation radii.



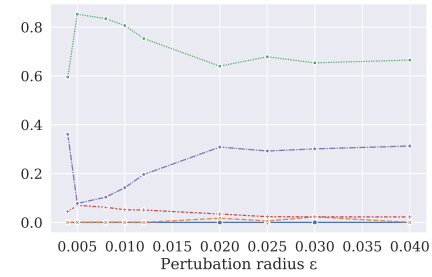

**(a)** MNIST - Instances solved



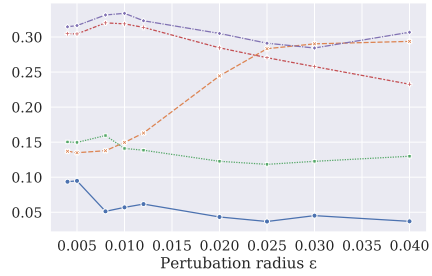

**(b)** CIFAR - Instances solved



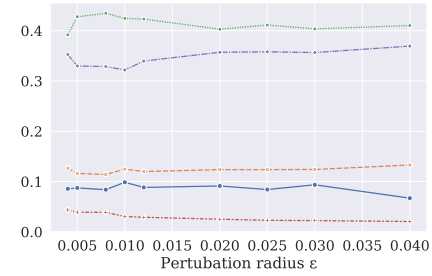

**(c)** MNIST - RMC




**(d)** CIFAR - RMC




**(e)** MNIST - Rel. Shapley Value




**(f)** CIFAR - Rel. Shapley Value

**Figure 3.9:** Performance of CPU-based verifiers for different values of $\epsilon$ in the ReLU category.

### 3.4.5 Joint Analysis of CPU- and GPU-Based Methods

As previously explained, directly comparing CPU- and GPU-based algorithms is a challenging endeavour, due to the different parallelisation schemes as well as the costs associated with running these algorithms. Here, we seek to capture both of these aspects by conducting a cost-calibrated analysis. More concretely, we compared these methods whilst factoring in the price of operating them on a prominent cloud

**Table 3.11:** Performance comparison of GPU-based verification algorithms in terms of the number of solved instances, relative marginal contribution ($RMC$), relative Shapley value ($\phi$) and average GPU running time, computed for each category and aggregated $\epsilon \in \{0.004, 0.005, 0.008, 0.01, 0.012, 0.02, 0.025, 0.03, 0.04\}$.

| **ReLU** | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Verifier | MNIST | | | | CIFAR | | | |
| | Solved | $RMC$ | $\phi$ | Avg. Time [GPU s] | Solved | $RMC$ | $\phi$ | Avg. Time [GPU s] |
| BaDNB | 9 886 | 0.71 | 0.19 | 1 864 | 21 438 | 0.90 | 0.45 | 100 |
| $\beta$-CROWN | 18 955 | 0.02 | 0.42 | 148 | 17 014 | 0.03 | 0.30 | 783 |
| MN-BaB | 17 799 | 0.27 | 0.39 | 363 | 14 675 | 0.07 | 0.25 | 1 174 |
| **ReLU+MaxPool** | | | | | | | | |
| Verifier | MNIST | | | | CIFAR | | | |
| | Solved | $RMC$ | $\phi$ | Avg. Time | Solved | $RMC$ | $\phi$ | Avg. Time |
| BaDNB | 720 | 0.03 | 0.22 | 1 493 | 0 | 0.00 | 0.00 | 3 600 |
| $\beta$-CROWN | 1 127 | 0.96 | 0.46 | 19 | 0 | 0.00 | 0.00 | 3 600 |
| MN-BaB | 966 | 0.01 | 0.32 | 366 | 576 | 1.00 | 1.00 | 0.008 |
| **Tanh** | | | | | | | | |
| Verifier | MNIST | | | | CIFAR | | | |
| | Solved | $RMC$ | $\phi$ | Avg. Time | Solved | $RMC$ | $\phi$ | Avg. Time |
| $\beta$-CROWN | 2 576 | 1.00 | 1.00 | 1.16 | 4 535 | 1.00 | 1.00 | 0.75 |
| MN-BaB | 0 | 0.00 | 0.00 | 3 600 | 0 | 0.00 | 0.00 | 3 600 |
| **Sigmoid** | | | | | | | | |
| Verifier | MNIST | | | | CIFAR | | | |
| | Solved | $RMC$ | $\phi$ | Avg. Time | Solved | $RMC$ | $\phi$ | Avg. Time |
| $\beta$-CROWN | 2 617 | 0.66 | 0.50 | 23 | 4 961 | 0.97 | 0.69 | 46 |
| MN-BaB | 2 601 | 0.33 | 0.50 | 44 | 3 042 | 0.03 | 0.31 | 1 420 |

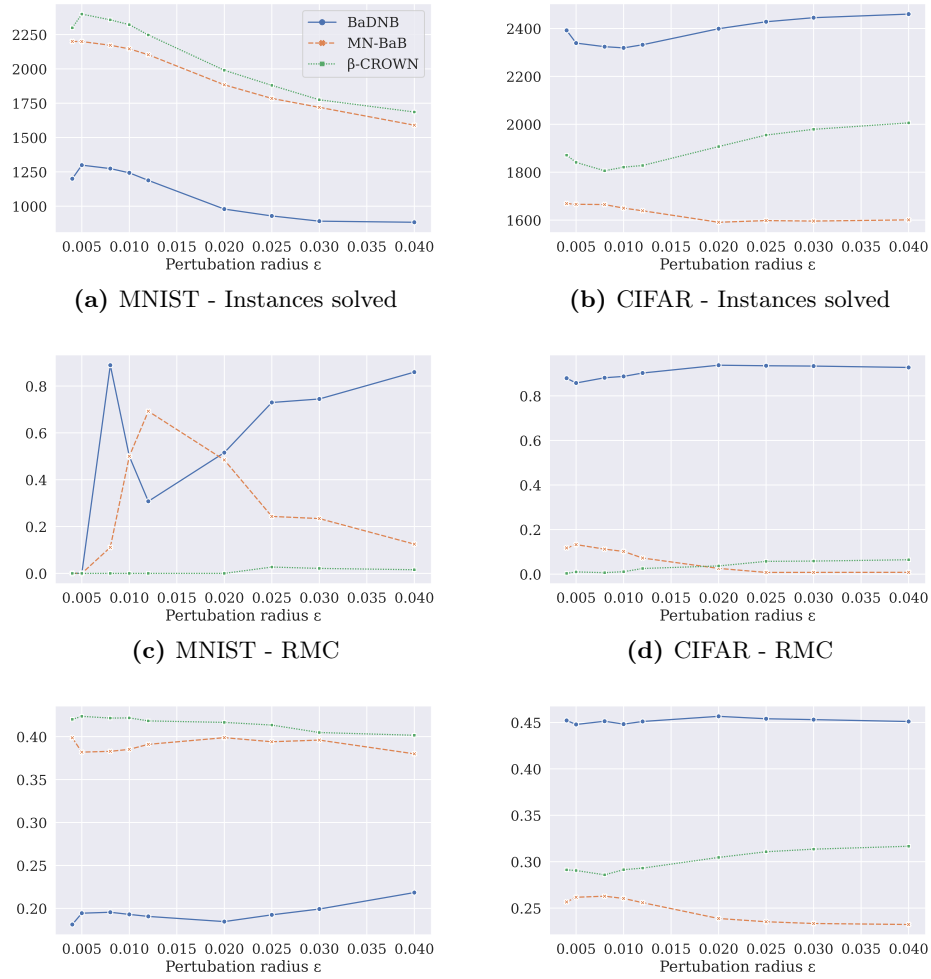
computing platform. To this end, we investigated the price difference between Amazon EC2 CPU instances comparable to the resources allocated in this study.[4] Notice that this hardware is not the exact hardware used in our experiments but is being used here as a substitute for calculating the cost of running similar hardware. Based on this cost difference, we reduced the time budget for GPU-based methods by a factor of 46.9, thereby ensuring that these methods cannot exceed the cost budget given to the CPU-based algorithms. While we carefully calibrated this factor based on existing prices, it must be noted that this analysis is based on many assumptions, and therefore, the comparison between CPU and GPU-based solvers serves only illustrative purposes.

[4]We selected the t2.medium and the g4dn.8xlarge instances, which cost \$0.0464 and \$2.176 per hour, respectively, see `https://aws.amazon.com/ec2/pricing/on-demand/`. Notice that there also exists the even cheaper t2.small instance with only a single CPU core; however, we did not select this machine as it has only 2 GB RAM.

**Table 3.12:** Performance comparison of GPU-based verification algorithms in terms of the number of solved instances, absolute marginal contribution ($MC$), absolute Shapley value ($\phi_{abs}$) and average GPU running time, computed for each category with $\epsilon \in \{0.004, 0.005, 0.008, 0.01, 0.012, 0.02, 0.025, 0.03, 0.04\}$.

| **ReLU** Verifier | MNIST | | | | CIFAR | | | |
|---|---|---|---|---|---|---|---|---|
| | Solved | $MC$ | $\phi_{abs}$ | Avg. Time [GPU s] | Solved | $MC$ | $\phi_{abs}$ | Avg. Time [GPU s] |
| BaDNB | 9 886 | 287 | 3 832 | 1 864 | 21 438 | 2 251 | 9 823 | 100 |
| $\beta$-CROWN | 18 955 | 6 | 8 226 | 148 | 17 014 | 72 | 6 521 | 784 |
| MN-BaB | 17 799 | 110 | 7 700 | 363 | 14 675 | 170 | 5 401 | 1 174 |
| **ReLU+MaxPool** Verifier | MNIST | | | | CIFAR | | | |
| | Solved | $MC$ | $\phi_{abs}$ | Avg. Time | Solved | $MC$ | $\phi_{abs}$ | Avg. Time |
| BaDNB | 720 | 5 | 244 | 1 493 | 0 | 0 | 0 | 3 600 |
| $\beta$-CROWN | 1 127 | 160 | 525 | 19 | 0 | 0 | 0 | 3 600 |
| MN-BaB | 966 | 1 | 365 | 531 | 576 | 576 | 576 | 0.009 |
| **Tanh** Verifier | MNIST | | | | CIFAR | | | |
| | Solved | $MC$ | $\phi_{abs}$ | Avg. Time | Solved | $MC$ | $\phi_{abs}$ | Avg. Time |
| $\beta$-CROWN | 2 576 | 2 576 | 2 576 | 1.16 | 4 535 | 4 535 | 4 535 | 0.75 |
| MN-BaB | 0 | 0 | 0 | 3 600 | 0 | 0 | 0 | 3 600 |
| **Sigmoid** Verifier | MNIST | | | | CIFAR | | | |
| | Solved | $MC$ | $\phi_{abs}$ | Avg. Time | Solved | $MC$ | $\phi_{abs}$ | Avg. Time |
| $\beta$-CROWN | 2 617 | 32 | 1 325 | 23 | 4 961 | 1 983 | 3 472 | 46 |
| MN-BaB | 2 601 | 16 | 1 309 | 44 | 3 042 | 64 | 1 553 | 1 421 |

Results from this analysis can be found in Table 3.13 for $\epsilon = 0.012$ and Table 3.14 for the full range of values of $\epsilon$ we considered. First and foremost, it can be seen that despite the higher costs associated with GPU resources, GPU-based verification tools (in particular $\beta$-CROWN, MN-BaB) are in many scenarios the most cost-efficient verifiers. However, the results also show that there exist scenarios in which CPU-based methods complement GPU-based methods in their performance. More concretely, Table 3.14 shows that the CPU-based verifier Marabou achieved the largest relative marginal contribution among all methods on MNIST networks from the ReLU category, indicating that it could solve a sizeable number of instances, which none of the other CPU- or GPU-based methods were able to solve within the same budget. In addition, the CPU-based verifier VeriNet achieved competitive marginal contribution and Shapley values. Furthermore, in the Tanh category, VeriNet was able to solve a large fraction of




**(a)** MNIST - Instances solved

**(b)** CIFAR - Instances solved

**(c)** MNIST - RMC

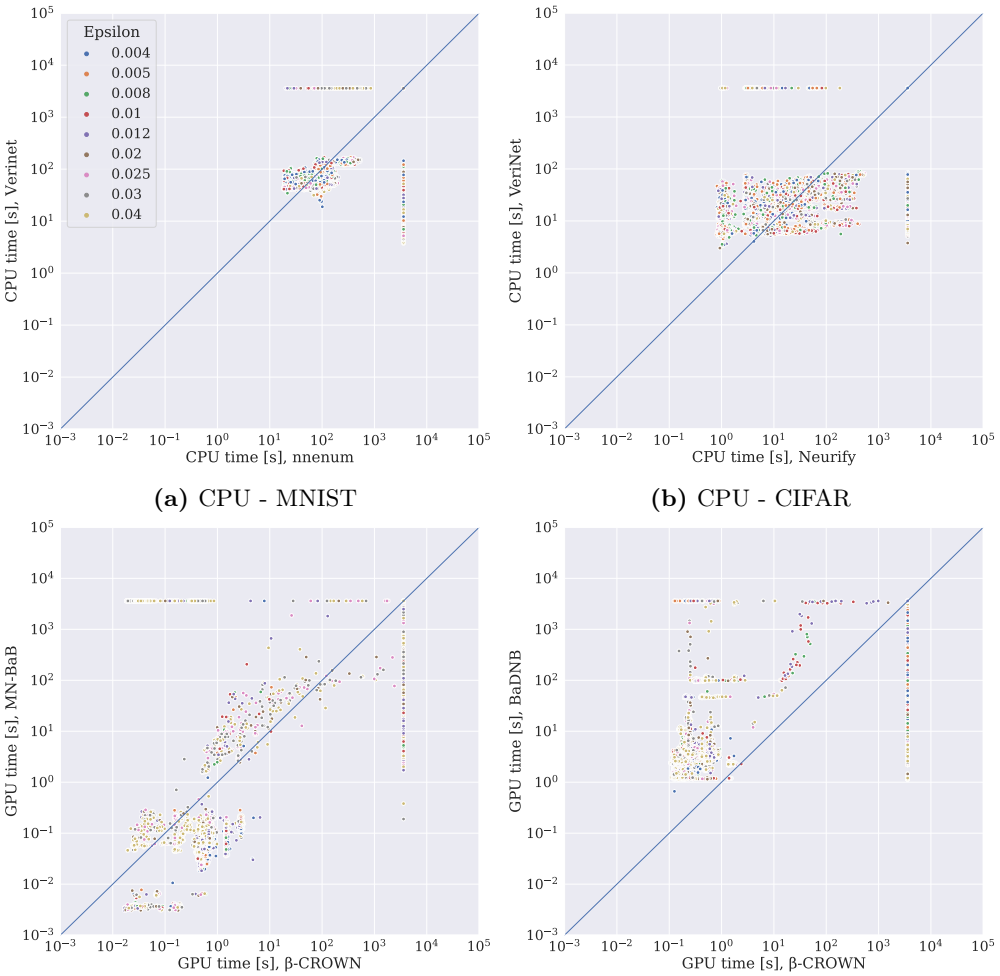
**(d)** CIFAR - RMC

**(e)** MNIST - Rel. Shapley value

**(f)** CIFAR - Rel. Shapley value

**Figure 3.10:** Performance of GPU-based verifiers for different values of $\epsilon$ in the ReLU category.

instances for which $\beta$-CROWN failed to return a solution; this observation holds when analysing both a single value of $\epsilon$ as well as the whole set of considered perturbation radii.




**(a)** CPU - MNIST

**(b)** CPU - CIFAR

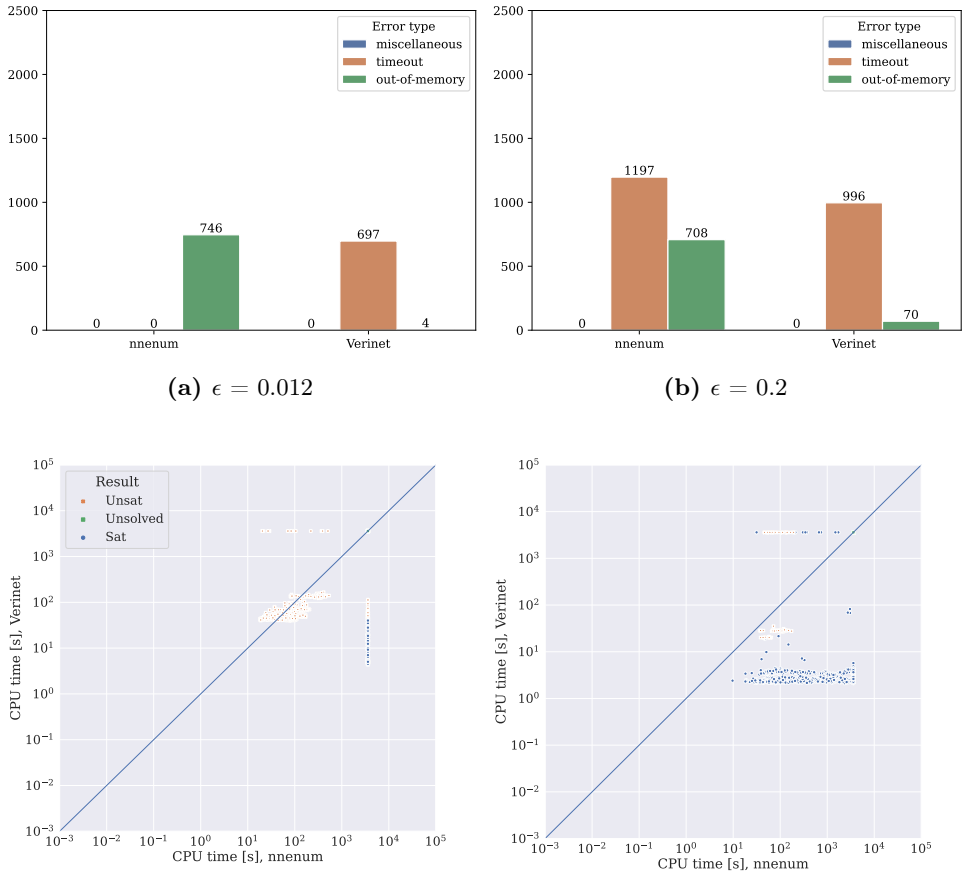
**(c)** GPU - MNIST

**(d)** GPU - CIFAR

**Figure 3.11:** Performance comparison of the two top-performing verification methods (in terms of relative Shapley value) in the ReLU category for CPU-based methods on (a) MNIST and (b) CIFAR networks as well GPU-based methods on (c) MNIST and (d) CIFAR networks, using multiple values of the perturbation radius $\epsilon$.

### 3.4.6 Analysis of *unsat* Instances

To gain further insights, we performed an analysis of *unsat* (*i.e.*, robust) instances; see Table 3.2 for the number of *unsat* instances that were found in each network category. More concretely, we considered only *unsat* instances as solved, since several verification methods considered in this study use counter-example generation mostly




**(a)** $\epsilon = 0.012$ **(b)** $\epsilon = 0.2$

**(c)** $\epsilon = 0.012$ **(d)** $\epsilon = 0.2$

**Figure 3.12: Top row**: Frequency of error types returned by the two top-performing verification methods (in terms of the number of solved instances) on instances in the ReLU category, when (a) $\epsilon = 0.012$ or (b) $\epsilon = 0.02$. **Bottom row**: Performance comparison of the two top-performing verification methods (in terms of relative Shapley value) in the ReLU category for CPU-based methods, when (c) $\epsilon = 0.012$ or (d) $\epsilon = 0.02$.

as an early stopping opportunity. Thus, *unsat* instances pose an interesting subset of the benchmark, as it measures the ability of a method to determine robustness in cases where no such counter-example exist. Furthermore, commonly used robustness metrics, such as adversarial accuracy, are computed by means of the fraction of *unsat* instances in a given instance set. Therefore, verification methods that can efficiently solve those instances enable a more accurate calculation of these metrics.

Table 3.15 shows result from this analysis for $\epsilon = 0.012$ while Table 3.16 shows

**Table 3.13:** Performance comparison of CPU- and GPU-based verification algorithms in terms of the number of solved instances, relative marginal contribution (*RMC*), relative Shapley value ($\phi$), computed for each category and $\epsilon = 0.012$.

| **ReLU** | | | | | | |
|---|---|---|---|---|---|---|
| Verifier | MNIST | | | CIFAR | | |
| | Solved | *RMC* | $\phi$ | Solved | *RMC* | $\phi$ |
| BaDNB | 1 171 | 0.12 | 0.25 | 2 217 | 0.46 | 0.43 |
| BaBSB | 358 | 0.00 | 0.00 | 307 | 0.00 | 0.00 |
| $\beta$-CROWN | 2 245 | 0.00 | 0.23 | 1 819 | 0.27 | 0.34 |
| Marabou | 1 001 | 0.06 | 0.03 | 400 | 0.00 | 0.00 |
| MN-BaB | 2 083 | 0.71 | 0.38 | 1 622 | 0.28 | 0.19 |
| Neurify | 871 | 0.06 | 0.03 | 915 | 0.00 | 0.03 |
| nnenum | 1 754 | 0.00 | 0.03 | 76 | 0.00 | 0.00 |
| VeriNet | 1 799 | 0.06 | 0.03 | 841 | 0.00 | 0.01 |
| **ReLU+MaxPool** | | | | | | |
| Verifier | MNIST | | | CIFAR | | |
| | Solved | *RMC* | $\phi$ | Solved | *RMC* | $\phi$ |
| BaDNB | 69 | 0.00 | 0.05 | 0 | 0.00 | 0.00 |
| $\beta$-CROWN | 128 | 1.00 | 0.67 | 0 | 0.00 | 0.00 |
| Marabou | 5 | 0.00 | 0.00 | 0 | 0.00 | 0.00 |
| MN-BaB | 115 | 0.00 | 0.27 | 64 | 1.00 | 1.00 |
| **Tanh** | | | | | | |
| Verifier | MNIST | | | CIFAR | | |
| | Solved | *RMC* | $\phi$ | Solved | *RMC* | $\phi$ |
| $\beta$-CROWN | 319 | 0.09 | 0.19 | 198 | 1.00 | 1.00 |
| MN-BaB | 0 | 0.00 | 0.00 | 0 | 0.00 | 0.00 |
| VeriNet | 556 | 0.91 | 0.81 | 0 | 0.00 | 0.00 |
| **Sigmoid** | | | | | | |
| Verifier | MNIST | | | CIFAR | | |
| | Solved | *RMC* | $\phi$ | Solved | *RMC* | $\phi$ |
| $\beta$-CROWN | 306 | 0.00 | 0.03 | 538 | 0.96 | 0.82 |
| Marabou | 0 | 0.00 | 0.00 | 0 | 0.00 | 0.00 |
| MN-BaB | 305 | 0.00 | 0.03 | 338 | 0.04 | 0.18 |
| VeriNet | 581 | 1.00 | 0.93 | 0 | 0.00 | 0.00 |

results aggregated over the full range of $\epsilon$ values we considered. First of all, we found that the total number of solved instances decreases when only *unsat* instances are considered. This is particularly noticeable for CIFAR, where the majority of instances

**Table 3.14:** Performance comparison of CPU- and GPU-based verification algorithms in terms of the number of solved instances, relative marginal contribution (*RMC*), relative Shapley value ($\phi$), computed for each category and aggregated $\epsilon \in \{0.004, 0.005, 0.008, 0.01, 0.012, 0.02, 0.025, 0.03, 0.04\}$.

| **ReLU** | | | | | | |
|---|---|---|---|---|---|---|
| Verifier | MNIST | | | CIFAR | | |
| | Solved | *RMC* | $\phi$ | Solved | *RMC* | $\phi$ |
| BaDNB | 9 455 | 0.10 | 0.20 | 20 408 | 0.48 | 0.43 |
| BaBSB | 3 716 | 0.00 | 0.00 | 2 690 | 0.00 | 0.00 |
| $\beta$-CROWN | 18 907 | 0.03 | 0.18 | 16 997 | 0.31 | 0.36 |
| Marabou | 9 457 | 0.44 | 0.20 | 3 651 | 0.00 | 0.00 |
| MN-BaB | 17 601 | 0.14 | 0.24 | 14 581 | 0.20 | 0.16 |
| Neurify | 8 206 | 0.04 | 0.02 | 8 173 | 0.00 | 0.03 |
| nnenum | 15 144 | 0.00 | 0.03 | 744 | 0.00 | 0.00 |
| VeriNet | 15 800 | 0.24 | 0.12 | 7 674 | 0.00 | 0.01 |
| **ReLU+MaxPool** | | | | | | |
| Verifier | MNIST | | | CIFAR | | |
| | Solved | *RMC* | $\phi$ | Solved | *RMC* | $\phi$ |
| BaDNB | 580 | 0.00 | 0.00 | 0 | 0.00 | 0.00 |
| $\beta$-CROWN | 1 127 | 0.99 | 0.74 | 0 | 0.00 | 0.00 |
| Marabou | 316 | 0.00 | 0.00 | 0 | 0.00 | 0.00 |
| MN-BaB | 966 | 0.00 | 0.22 | 576 | 1.00 | 1.00 |
| **Tanh** | | | | | | |
| Verifier | MNIST | | | CIFAR | | |
| | Solved | *RMC* | $\phi$ | Solved | *RMC* | $\phi$ |
| $\beta$-CROWN | 2 576 | 0.17 | 0.24 | 4 535 | 1.00 | 1.00 |
| MN-BaB | 0 | 0.00 | 0.00 | 0 | 0.00 | 0.00 |
| VeriNet | 4 307 | 0.83 | 0.76 | 0 | 0.00 | 0.00 |
| **Sigmoid** | | | | | | |
| Verifier | MNIST | | | CIFAR | | |
| | Solved | *RMC* | $\phi$ | Solved | *RMC* | $\phi$ |
| $\beta$-CROWN | 2 617 | 0.00 | 0.05 | 4 961 | 0.97 | 0.83 |
| Marabou | 0 | 0.00 | 0.00 | 0 | 0.00 | 0.00 |
| MN-BaB | 2 601 | 0.00 | 0.05 | 3 042 | 0.03 | 0.17 |
| VeriNet | 4 728 | 1.00 | 0.90 | 0 | 0.00 | 0.00 |

are non-robust or, in other words, *sat*. Furthermore, we observed only minor changes in the relative performance and complementarity of the given verifiers on MNIST

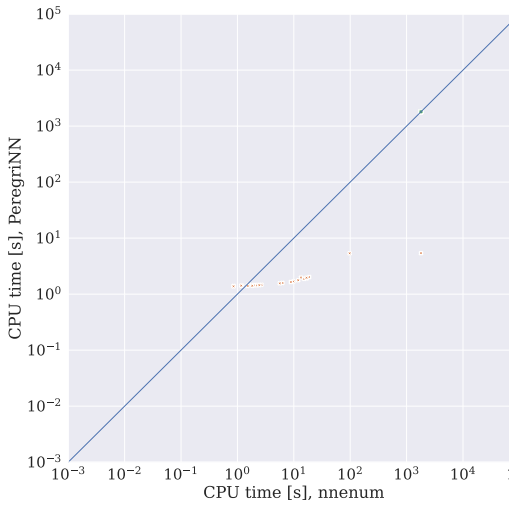
instances across all categories. Specifically, we found that for the broader set of $\epsilon$ values, the *RMC* and Shapley value of Marabou improve substantially, while those for VeriNet strongly deteriorate. This indicates that on *unsat* instances, Marabou can solve a large fraction of instances unsolved by other methods, while VeriNet mainly contributes when *sat* instances are also considered. For CIFAR, we also noticed that the relative performance of the given verifiers changed. Specifically, MN-BaB, which previously achieved competitive relative performance does not seem to complement other methods on *unsat* instances; instead, most instances are solved by BaDNB and $\alpha\beta$-CROWN, which also show strong complementarity in the ReLU category.

### 3.4.7 Analysis of the 2022 VNN Competition Results

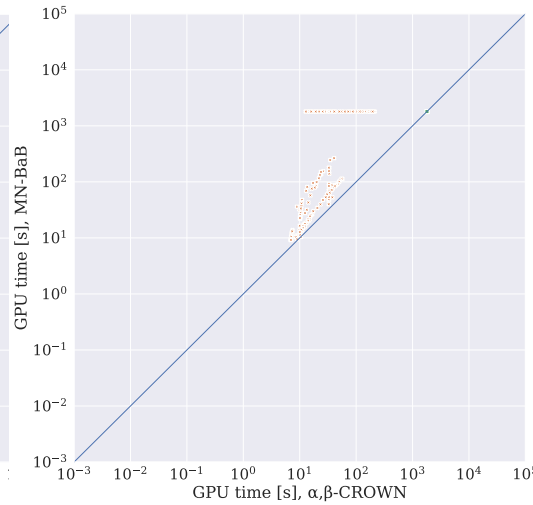
To see if and to what extent our observations hold for a larger set of verifiers as well as different benchmarks, we analysed the results of the 2022 edition of the VNN competition. We refer to the accompanying report [87] for more information about the participating tools, benchmarks and further technical details. Again, we present a joint as well as a separate analysis of CPU- and GPU-based verification algorithms. We excluded CGDTest from the set of methods considered in our analysis, as it represents the only incomplete verification approach participating in the competition, while our work focuses on complete verification. In addition, CGDTest produced a substantial number of incorrect results in the competition, casting doubts on the soundness of the method.

Table 3.17 shows the results from the VNN competition for CPU-based verification algorithms. It reports the number of problem instances solved by each verifier per network category, marginal contribution as well as Shapley values, both in absolute and relative terms. Most notably, we observe strong complementarity between the verifiers considered in two of the three benchmark categories. Concretely, in the CNN + ResNet category, Marabou and VeraPak achieved relative Shapley values of 0.44 and 0.24, respectively. Indeed, as depicted in Figure 3.13c, there are several instances solved by one of the verifiers but unsolved by the other.

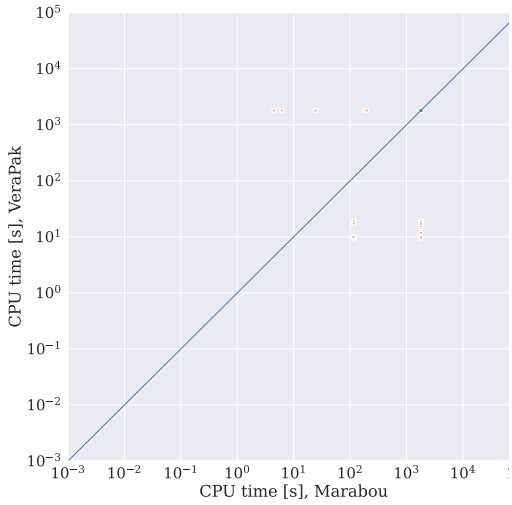
In the FC category, Marabou, nnenum and PerigiNN achieved a similar relative Shapley value of 0.24, again highlighting the complementarity between these algorithms. Given the similar relative Shapley values, we resort to the relative marginal contribution to determine the two best-performing methods in this context; *i.e.*, among these three methods, nnenum and PerigiNN achieved the largest relative marginal contributions and are, thus, considered the two best-performing methods. Again, we compare their



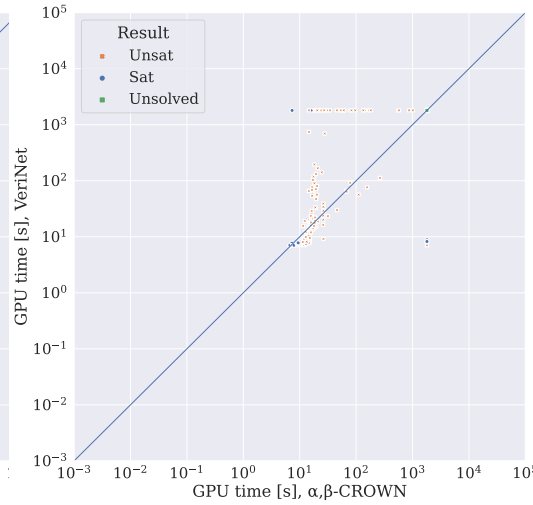

**(a)** CPU - Complex




**(b)** GPU - Complex



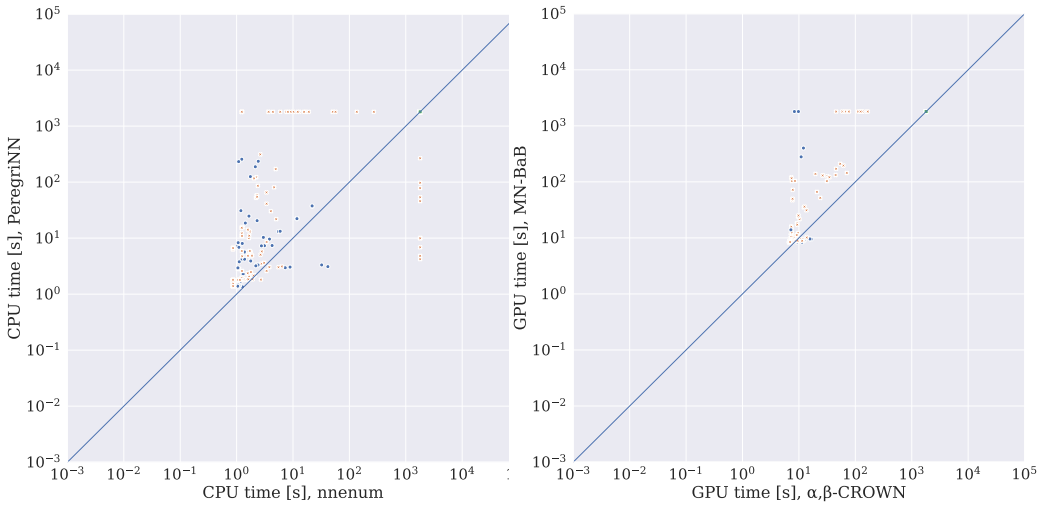

**(c)** CPU - CNN+ResNet




**(d)** GPU - CNN+ResNet

**Figure 3.13:** Performance comparison of the two top-performing verification methods (in terms of relative Shapley value) in each category from the 2022 VNN Competition. Instances that were not solved within their respective time limit are displayed with the maximum running time attributed to any instance in the benchmark set (*i.e.*, 1 800 seconds). (Part 1 of 2)

performance on an instance level, as shown in Figure 3.14a. As can be observed, instances spread out widely around the equal performance line of the plot, with many




**(a)** CPU - FC **(b)** GPU - FC

**Figure 3.14:** Performance comparison of the two top-performing verification methods (in terms of relative Shapley value) in each category from the 2022 VNN Competition. Instances that were not solved within their respective time limit are displayed with the maximum running time attributed to any instance in the benchmark set (*i.e.*, 1 800 seconds). (Part 2 of 2)

instances solved by nnenum but unsolved by PerigiNN, and vice versa.

In the Complex category, nnenum and PeregriNN achieved Shapley values of 0.46 and 0.50, respectively. However, Figure 3.13a reveals that nnenum dominates in performance over PeregriNN on most instances. We note that the Shapley value represents the average contribution made by a given verifier over all possible sets of algorithms in a portfolio. Hence, it indicates that nnenum could solve many instances unsolved by other methods from the full set of algorithms under consideration; however, nnenum does not complement PeregriNN in terms of solved instances.

Next, we discuss the results from the 2022 VNN Competition for GPU-based verification algorithms; these are presented in Table 3.18. Surprisingly, for GPU-based methods, our findings from analysing the competition results differ from those made in our previous assessment, as they do not reveal strong complementarity between the algorithms. Specifically, $\beta$-CROWN dominates in performance on every instance in each category, although relative Shapley values indicate complementary (for similar reasons as those outlined above).

This reflected in Figure 3.13b, Figure 3.13d and Figure 3.14b. Concretely, these

plots show the performance on an instance level for the two top-performing methods in each category (in terms of relative Shapley values). In the Complex and FC category, these are $\beta$-CROWN and MN-BaB, while in the CNN + ResNet category, these are $\beta$-CROWN and VeriNet. The latter category represents the only category in which a small degree of complementarity can be observed, as both verifiers solved some instances unsolved by the other. However, the fraction solved by VeriNet remains comparably small.

Finally, Table 3.19 presents the joint analysis of CPU- and GPU-based methods based on the competition results. Notice that we did not perform a cost calibration in this case, as verifiers were employed on hardware with about equal costs. Most interestingly, we observed performance complementary between these methods in the CNN+ResNet category. More specifically, the CPU-based Marabou solver could solve several instances unsolved by GPU-based $\beta$-CROWN verifier, although the latter solved the most instances overall, as reflected in the relative Shapley values (0.53 *vs* 0.32). Again, this shows that there exist scenarios in which CPU-based methods complement GPU-based methods in their performance.

Overall, we find that the biggest difference between the results of the VNN competition and the results obtained in this study is the degree of complementarity between the GPU-based verification algorithms, as reflected by the marginal contribution and Shapley values. While the results from the VNN competition suggest that there is a single best GPU-based verifier that broadly dominates all other methods, the results presented in our study reveal a more nuanced story. This difference can most likely be attributed to the size and the diversity of the proposed benchmark: while the 2022 VNN Competition considered 17 neural networks as test cases for local robustness verification, our benchmark consists of 79 networks. At the same time, the competition provides valuable insights into how the considered verifiers perform when carefully adapted to a specific benchmark. Moreover, while both analyses have clear contributions, our results highlight the importance of introducing a larger and more diverse benchmark set.

## 3.5 Conclusions and Future Work

In this chapter, we sought to answer the question of what constitutes the state of the art in neural network verification and, thus, address RQ1 of this thesis. To this end, we assessed the performance of a collection of well-known, complete local robustness verification algorithms, *i.e.*, algorithms used to verify the robustness of an image classification network against small input perturbations. We found that

all of these methods support ReLU-based networks, while other network types are strongly under-supported. While this has been suspected in the community, it has, to our knowledge, not yet been subject to formal study. Generally, we observed that all considered verification algorithms show severe limitations with regard to the network structures they can process – in many cases due to unsupported layer operations and in others due to undefined errors.

Furthermore, and more importantly, we presented evidence for strong performance complementarity: even within the same benchmark category (as defined based on verifier compatibility), any two verification systems outperform each other on distinct subsets of instances. Thereby, the state of the art in neural network verification cannot be described by a single algorithm but rather several algorithms that contribute to varying degrees with their own strengths. As we have demonstrated, this complementarity can be exploited by combining individual verifiers into parallel portfolios. At the same time, automated portfolio construction comes with its own challenges, leaving room for further research into the development and evaluation of appropriate frameworks.

Lastly, we showed that, in general, the performance of verifiers strongly differs between image datasets, with some methods achieving the best performance on MNIST (in terms of the number of solved instances and average running time) while falling behind on CIFAR and vice versa. In addition, even for the same dataset, we found that the performance of a given verifier can change drastically depending on the perturbation radius; *i.e.*, an algorithm that performs well for a small value of $\epsilon$ might degrade in performance as the value of $\epsilon$ increases.

In future work, it would be interesting to analyse in more detail how the relative performance of verifiers depends on the given perturbation radius and other performance-relevant characteristics of the given networks and image classification tasks. We suspect this to be an interesting yet challenging research direction, as it requires a novel definition of features specific to neural network verification problem. To the best of our knowledge, no research on the development of such meta-features has been conducted yet. Due to the specifics of both the verification problem instances as well as the verification algorithms that should be systematically explored, we consider this a non-trivial but important challenge to be solved in future work. This line of research would also enable empirical performance modelling. An empirical performance model is a model that predicts the performance, *e.g.*, the running time, of algorithms on previously unseen input, including previously unseen problem instances. Finally, it would be interesting to expand this analysis to other datasets and machine learning tasks beyond supervised image classification.

**Table 3.15:** Performance comparison of CPU- and GPU-based verification algorithms in terms of the number of *unsat* instances, relative marginal contribution (*RMC*), relative Shapley value ($\phi$), computed for each category and $\epsilon = 0.012$.

| **ReLU** Verifier | MNIST | | | CIFAR | | |
|---|---|---|---|---|---|---|
| | Solved | *RMC* | $\phi$ | Solved | *RMC* | $\phi$ |
| BaDNB | 1 072 | 0.13 | 0.25 | 86 | 0.65 | 0.63 |
| BaBSB | 161 | 0.00 | 0.00 | 0 | 0.00 | 0.00 |
| $\beta$-CROWN | 2 143 | 0.00 | 0.23 | 61 | 0.35 | 0.36 |
| Marabou | 995 | 0.07 | 0.03 | 6 | 0.00 | 0.00 |
| MN-BaB | 2 025 | 0.80 | 0.40 | 16 | 0.00 | 0.00 |
| Neurify | 748 | 0.00 | 0.00 | 20 | 0.00 | 0.00 |
| nnenum | 1 686 | 0.00 | 0.03 | 26 | 0.00 | 0.00 |
| VeriNet | 1 675 | 0.00 | 0.03 | 20 | 0.00 | 0.00 |
| **ReLU+MaxPool** Verifier | MNIST | | | CIFAR | | |
| | Solved | *RMC* | $\phi$ | Solved | *RMC* | $\phi$ |
| BaDNB | 59 | 0.00 | 0.10 | 0 | 0.00 | 0.00 |
| $\beta$-CROWN | 88 | 1.00 | 0.52 | 0 | 0.00 | 0.00 |
| Marabou | 5 | 0.00 | 0.00 | 0 | 0.00 | 0.00 |
| MN-BaB | 86 | 0.00 | 0.38 | 0 | 0.00 | 0.00 |
| **Tanh** Verifier | MNIST | | | CIFAR | | |
| | Solved | *RMC* | $\phi$ | Solved | *RMC* | $\phi$ |
| $\beta$-CROWN | 291 | 0.09 | 0.18 | 3 | 1.00 | 1.00 |
| MN-BaB | 0 | 0.00 | 0.00 | 0 | 0.00 | 0.00 |
| VeriNet | 527 | 0.91 | 0.82 | 0 | 0.00 | 0.00 |
| **Sigmoid** Verifier | MNIST | | | CIFAR | | |
| | Solved | *RMC* | $\phi$ | Solved | *RMC* | $\phi$ |
| $\beta$-CROWN | 272 | 0.00 | 0.03 | 66 | 1.00 | 0.00 |
| Marabou | 0 | 0.00 | 0.00 | 0 | 0.00 | 0.00 |
| MN-BaB | 272 | 0.00 | 0.03 | 0 | 0.00 | 0.00 |
| VeriNet | 544 | 1.00 | 0.94 | 0 | 0.00 | 0.00 |

**Table 3.16:** Performance comparison of CPU- and GPU-based verification algorithms in terms of the number of *unsat* instances, relative marginal contribution (*RMC*), relative Shapley value ($\phi$), computed for each category and aggregated $\epsilon \in \{0.004, 0.005, 0.008, 0.01, 0.012, 0.02, 0.025, 0.03, 0.04\}$.

| **ReLU** | | | | | | |
|---|---|---|---|---|---|---|
| Verifier | MNIST | | | CIFAR | | |
| | Solved | *RMC* | $\phi$ | Solved | *RMC* | $\phi$ |
| BaDNB | 8 059 | 0.15 | 0.21 | 1 069 | 0.63 | 0.59 |
| BaBSB | 2 303 | 0.00 | 0.00 | 0 | 0.00 | 0.00 |
| $\beta$-CROWN | 17 433 | 0.04 | 0.19 | 866 | 0.36 | 0.39 |
| Marabou | 9 290 | 0.61 | 0.25 | 60 | 0.00 | 0.00 |
| MN-BaB | 16 588 | 0.20 | 0.28 | 144 | 0.00 | 0.00 |
| Neurify | 6 992 | 0.00 | 0.00 | 168 | 0.00 | 0.00 |
| nnenum | 14 601 | 0.00 | 0.03 | 223 | 0.00 | 0.01 |
| VeriNet | 14 317 | 0.00 | 0.02 | 177 | 0.00 | 0.00 |
| **ReLU+MaxPool** | | | | | | |
| Verifier | MNIST | | | CIFAR | | |
| | Solved | *RMC* | $\phi$ | Solved | *RMC* | $\phi$ |
| BaDNB | 418 | 0.00 | 0.08 | 0 | 0.00 | 0.00 |
| $\beta$-CROWN | 573 | 1.00 | 0.50 | 0 | 0.00 | 0.00 |
| Marabou | 274 | 0.00 | 0.02 | 0 | 0.00 | 0.00 |
| MN-BaB | 566 | 0.00 | 0.40 | 0 | 0.00 | 0.00 |
| **Tanh** | | | | | | |
| Verifier | MNIST | | | CIFAR | | |
| | Solved | *RMC* | $\phi$ | Solved | *RMC* | $\phi$ |
| $\beta$-CROWN | 2 248 | 0.15 | 0.22 | 151 | 1.00 | 1.00 |
| MN-BaB | 0 | 0.00 | 0.00 | 0 | 0.00 | 0.00 |
| VeriNet | 3 993 | 0.85 | 0.78 | 0 | 0.00 | 0.00 |
| **Sigmoid** | | | | | | |
| Verifier | MNIST | | | CIFAR | | |
| | Solved | *RMC* | $\phi$ | Solved | *RMC* | $\phi$ |
| $\beta$-CROWN | 2 290 | 0.00 | 0.04 | 575 | 1.00 | 1.00 |
| Marabou | 0 | 0.00 | 0.00 | 0 | 0.00 | 0.00 |
| MN-BaB | 2 302 | 0.00 | 0.04 | 0 | 0.00 | 0.00 |
| VeriNet | 4 448 | 1.00 | 0.92 | 0 | 0.00 | 0.00 |

**Table 3.17:** Performance comparison of CPU-based verification algorithms in terms of the number of solved instances, absolute and relative marginal contribution (*MC*, *RMC*), absolute and relative Shapley value ($\phi_{abs}$, $\phi$) as well as average running time, computed for each category from the 2022 VNN Competition.

| **Complex**<br>Verifier | Solved | *MC* | *RMC* | $\phi_{abs}$ | $\phi$ | Avg. Time |
|---|---|---|---|---|---|---|
| AveriNN | 0 | 0 | 0.00 | 0 | 0.00 | 192 |
| Debona | 2 | 0 | 0.00 | 1 | 0.04 | 192 |
| FastBATLLNN | 0 | 0 | 0.00 | 0 | 0.00 | 192 |
| Marabou | 0 | 0 | 0.00 | 0 | 0.00 | 192 |
| nnenum | 23 | 0 | 0.00 | 11 | 0.46 | 190 |
| PeregriNN | 24 | 1 | 1.00 | 12 | 0.50 | 189 |
| VeraPak | 0 | 0 | 0.00 | 0 | 0.00 | 192 |
| **CNN + ResNet**<br>Verifier | Solved | *MC* | *RMC* | $\phi_{abs}$ | $\phi$ | Avg. Time |
| AveriNN | 0 | 0 | 0.00 | 0 | 0.00 | 357 |
| Debona | 0 | 0 | 0.00 | 0 | 0.00 | 357 |
| FastBATLLNN | 0 | 0 | 0.00 | 0 | 0.00 | 357 |
| Marabou | 122 | 91 | 0.61 | 106 | 0.44 | 264 |
| nnenum | 81 | 17 | 0.11 | 48 | 0.20 | 273 |
| PeregriNN | 57 | 0 | 0.00 | 28 | 0.12 | 325 |
| VeraPak | 72 | 42 | 0.28 | 57 | 0.24 | 254 |
| **FC**<br>Verifier | Solved | *MC* | *RMC* | $\phi_{abs}$ | $\phi$ | Avg. Time |
| AveriNN | 100 | 0 | 0.00 | 20 | 0.05 | 166 |
| Debona | 339 | 3 | 0.30 | 82 | 0.19 | 91 |
| FastBATLLNN | 32 | 1 | 0.10 | 10 | 0.0 | 0.5 |
| Marabou | 404 | 0 | 0.00 | 102 | 0.24 | 53 |
| nnenum | 411 | 1 | 0.10 | 105 | 0.24 | 37 |
| PeregriNN | 397 | 2 | 0.20 | 102 | 0.24 | 48 |
| VeraPak | 50 | 3 | 0.30 | 13 | 0.03 | 66 |

**Table 3.18:** Performance comparison of GPU-based verification algorithms in terms of the number of solved instances, absolute and relative marginal contribution (*MC*, *RMC*), absolute and relative Shapley value ($\phi_{abs}$, $\phi$) as well as average running time, computed for each category from the 2022 VNN Competition.

| **Complex**<br>Verifier | | | | | | |
|---|---|---|---|---|---|---|
| | Solved | *MC* | *RMC* | $\phi_{abs}$ | $\phi$ | Avg. Time |
| $\beta$-CROWN | 191 | 66 | 1.00 | 0 | 0.62 | 72 |
| MN-BaB | 125 | 0 | 0.00 | 0 | 0.28 | 164 |
| VeriNet | 60 | 0 | 0.00 | 0 | 0.10 | 187 |
| **CNN + ResNet**<br>Verifier | | | | | | |
| | Solved | *MC* | *RMC* | $\phi_{abs}$ | $\phi$ | Avg. Time |
| $\beta$-CROWN | 312 | 28 | 1.00 | 0 | 0.42 | 107 |
| MN-BaB | 254 | 0 | 0.00 | 0 | 0.28 | 179 |
| VeriNet | 259 | 0 | 0.00 | 0 | 0.30 | 171 |
| **FC**<br>Verifier | | | | | | |
| | Solved | *MC* | *RMC* | $\phi_{abs}$ | $\phi$ | Avg. Time |
| $\beta$-CROWN | 448 | 11 | 1.00 | 0 | 0.35 | 15 |
| MN-BaB | 433 | 0 | 0.00 | 0 | 0.33 | 30 |
| VeriNet | 435 | 0 | 0.00 | 0 | 0.32 | 21 |

**Table 3.19:** Performance comparison of GPU- and CPU-based verification algorithms in terms of the number of solved instances, absolute and relative marginal contribution (*MC*, *RMC*) as well as absolute and relative Shapley value ($\phi_{abs}$, $\phi$), computed for each category from the 2022 VNN Competition.

| **Complex** Verifier | Solved | *MC* | *RMC* | $\phi_{abs}$ | $\phi$ |
|---|---|---|---|---|---|
| AveriNN | 0 | 0 | 0.00 | 0 | 0.00 |
| $\beta$-CROWN | 191 | 66 | 0.99 | 20 | 0.91 |
| Debona | 2 | 0 | 0.00 | 0 | 0.04 |
| FastBATLLNN | 0 | 0 | 0.00 | 0 | 0.00 |
| Marabou | 0 | 0 | 0.00 | 0 | 0.00 |
| MN-BaB | 125 | 0 | 0.00 | 2 | 0.09 |
| nnenum | 23 | 0 | 0.00 | 0 | 0.46 |
| PeregriNN | 24 | 1 | 0.01 | 0 | 0.50 |
| VeraPak | 0 | 0 | 0.00 | 0 | 0.00 |
| VeriNet | 60 | 0 | 0.00 | 0 | 0.10 |
| **CNN + ResNet** Verifier | Solved | *MC* | *RMC* | $\phi_{abs}$ | $\phi$ |
| AveriNN | 0 | 0 | 0.00 | 0 | 0.00 |
| $\beta$-CROWN | 312 | 15 | 0.28 | 6 | 0.32 |
| Debona | 0 | 0 | 0.00 | 0 | 0.00 |
| FastBATLLNN | 0 | 0 | 0.00 | 0 | 0.00 |
| Marabou | 122 | 36 | 0.68 | 10 | 0.53 |
| MN-BaB | 254 | 0 | 0.00 | 1 | 0.05 |
| nnenum | 81 | 0 | 0.00 | 0 | 0.00 |
| PeregriNN | 57 | 0 | 0.00 | 0 | 0.00 |
| VeraPak | 72 | 2 | 0.04 | 1 | 0.05 |
| VeriNet | 259 | 0 | 0.00 | 1 | 0.05 |
| **FC** Verifier | Solved | *MC* | *RMC* | $\phi_{abs}$ | $\phi$ |
| AveriNN | 100 | 0 | 0.00 | 0 | 0.00 |
| $\beta$-CROWN | 448 | 9 | 1.00 | 3 | 1.00 |
| Debona | 339 | 0 | 0.00 | 0 | 0.00 |
| FastBATLLNN | 32 | 0 | 0.00 | 0 | 0.00 |
| Marabou | 404 | 0 | 0.00 | 0 | 0.00 |
| MN-BaB | 433 | 0 | 0.00 | 0 | 0.00 |
| nnenum | 411 | 0 | 0.00 | 0 | 0.00 |
| PeregriNN | 397 | 0 | 0.00 | 0 | 0.00 |
| VeraPak | 50 | 0 | 0.00 | 0 | 0.00 |
| VeriNet | 435 | 0 | 0.00 | 0 | 0.00 |

# Chapter 4

# Speeding Up MIP-Based Neural Network Verification via Automated Algorithm Configuration

As outlined in the previous chapter, formal network verification methods tends to be computationally expensive, making it difficult to verify networks with a large number of units and/or on a large number of inputs. At the same time, we have shown that there exist performance complementarity among different verification algorithms. This can be exploited by constructing algorithm portfolios in a principled manner; *i.e.*, constructing them in such a way that they contain a set of solvers that complement each other in the most effective way possible.

As mentioned in Chapter 2.2.3, it is possible to formulate the verification task as a constraint optimisation problem using *mixed integer programming (MIP)*. In light of this, recent work by Tjend et al. [104] presented a verification tool, called MIPVerify, which formulates the verification task as a minimisation problem, which is then solved using a commercial MIP solver. More specifically, the optimisation task is to apply a perturbation to the original sample that maximises model error, while staying close to the initial example, *i.e.*, keeping the distance at a minimum. In other words, the verifier takes an image and a trained neural network as inputs and produces either

an adversarial example or, if the optimisation problem cannot be solved, a certificate of local robustness. While MIPVerify can verify a larger number of instances than previous methods, such as those from the works of Wong et al. [113], Dvijotham et al. [26] or Raghunathan et al. [93], it is computationally costly (in terms of CPU time required per verification query). Specifically, depending on the classifier to be verified, we found that some instances required several thousand CPU seconds of running time of the MIP solver, while a sizeable fraction of instances could not be solved at all, even within a rather generous time limit of 38 400 CPU seconds per sample.

The same holds for other MIP-based verification systems, such as Venus [8]. Here, our experiments showed that, depending on the classifier to be verified, the computational cost per query remains subject to great variance as outlined above, with many instances resulting in timeouts.

We note that, to date, the performance of MIPVerify and Venus has not been compared directly, which motivates our decision to consider both as contributors to the state of the art in MIP-based neural network verification.

Previous work has demonstrated that automated configuration of MIP solvers can yield substantial improvements [46, 44, 45, 76]. Building on these findings, we seek to improve the performance of MIP-based neural network verification tools by leveraging *automated algorithm configuration* techniques to optimise the hyperparameters of the solver at the heart of these verifiers. As such, the proposed method can be used regardless of the underlying MIP problem formulation, and its improvements are orthogonal to any advances made with regard to the formulation. Put differently, we argue that automated algorithm configuration can benefit any verification approach relying on MIP solving or similar techniques.

Automated algorithm configuration of neural network verification engines is a non-trivial task and comes with several challenges. Most prominently, the high running times and heterogeneity/diversity of instances pose problems that are not easily solved by standard configuration approaches, such as SMAC [45]. More precisely, we consistently found in our experiments that a single configuration could not significantly improve mean CPU time over the default. In fact, we observed that a single configuration could achieve a 500-fold speedup on a given instance over the default, but then time out on another, which the default, in turn, could solve. Therefore, we decided to adapt Hydra [116], an advanced approach that combines algorithm configuration and per-instance algorithm selection, to automatically construct a *parallel portfolio* of MIP solver configurations optimised for solving neural network verification problems.

We demonstrate the effectiveness of our approach for both aforementioned verifica-

tion tools. These systems both rely on MIP solving, yet they are conceptually different enough to show the generalisability of our method. To the best of our knowledge, ours is the first study to pursue this direction. In brief, the main contributions of this chapter are as follows:

- A framework for automatically constructing a parallel portfolio of MIP solver configurations optimised for neural network verification, which can be applied to any MIP-based verification method;

- an extensive evaluation of this framework on two well known verification engines, namely Venus [8] and MIPVerify, improving their performance on (i) $\mathrm{SDP_dMLP_A}$ - an MNIST classifier designed for robustness [93], (ii) mnistnet - an MNIST classifier from the neural network verification literature [8] and (iii) the ACAS Xu benchmark [51, 55].

On the $\mathrm{SDP_dMLP_A}$ benchmark, we achieved substantial improvements in CPU time by average factors of 4.7 and 10.3 for MIPVerify and Venus, respectively, on a *solvable* subset of instances from the MNIST dataset. This subset excludes all instances that cannot be solved by any of the baseline approaches we consider. Beyond that, the number of timeouts was reduced by a factor of 1.42 and 1.6, respectively.

On the mnistnet benchmark, we again achieved substantial improvements in CPU time, this time by average factors of 1.61 and 7.26 for MIPVerify and Venus, respectively, on solvable instances. We furthermore reduced timeouts on this benchmark by average factors of 1.14 and 2.81, respectively.

Finally, we strongly improved the performance of the Venus verifier on the ACAS Xu benchmark, attaining a 2.97-fold reduction in average CPU time. We note that on this benchmark, we found MIPVerify to be unable to solve most of the instances within the kinds of computational budgets considered in our experiments.

## 4.1 Background

The following section provides details of MIP-based neural network verification algorithms. It further puts focus on the limitations of current approaches and introduces the concepts behind automated algorithm configuration and portfolio construction.

### 4.1.1 MIP-Based Neural Network Verification

MIPVerify combines and extends existing approaches to MIP-based robustness verification [17, 75, 25, 32] and presents a verifier that encodes the network as a set of mixed-integer linear constraints. Following [104], a valid adversarial example $x'$ for input $x$ with true class label $\lambda(x)$ (encoded as integer) corresponds to the solution to the problem where we minimise:

$$d(x', x) \tag{4.1}$$

subject to

$$\arg\max_i(f_i(x')) \neq \lambda(x) \tag{4.2}$$

$$x' \in (G(x) \cap X_{valid}), \tag{4.3}$$

where $d(\cdot, \cdot)$ denotes a distance metric (e.g., the $l_\infty$-norm), $f_i(\cdot)$ is the $i$-th network output (*i.e.*, indicating whether it predicts the input to belong to the $i$-th class) and $G(x) = \{x' \mid \forall i : -\varepsilon \leq (x - x')_i \leq \varepsilon\}$. Intuitively, $G(x)$ denotes the region around an input $x$ corresponding to all allowable perturbations within a pre-defined radius $\varepsilon$. $X_{valid}$ represents the domain of valid inputs (*e.g.*, the pixel value range of a normalised image, in case of image classification). Note that this formulation assumes that the network predicts a single class label for each observation (*i.e.*, the arg max operator in Eq. 4.2 returns a single element); other behaviour is undefined.

MIPVerify achieves speed-ups through optimised MIP formulations or, more specifically, tight formulations for non-linearities and a pre-solving algorithm that reduces the number of binary variables, *i.e.*, the number of unstable ReLU nodes. More specifically, the information provided by $G(x)$ is used to reduce the interval of the input domain propagated through the network during the calculation of the pre-activation bounds. This is combined with *progressive bounds tightening*, which represents a method for choosing procedures to determine pre-activation bounds, *i.e.*, interval arithmetic or linear programming, based on the potential improvement to the problem formulation.

The MIP-based verifier Venus [8] achieves performance gains over previous methods, such as NSVerify [1], through dependency-based pruning to reduce the search space during branch-and-bound and combines this *dependency analysis* approach with symbolic interval arithmetic and domain splitting techniques.

Moreover, both [104] and [8] report state-of-the-art performance on various network architectures and datasets but their tools consume very substantial amounts of CPU time. Depending on the classifier to be verified, we observed that finding a solution

can easily take up to several hours of computation time for a single instance. Network verification can therefore turn into an extremely time-consuming endeavour, even for a relatively small dataset, such as MNIST. At the same time, a verifier fails to maintain the premise of completeness, meaning that it can certify every input example it is presented with if many instances are subject to timeouts, which we also found to be the case for the verification methods considered in this study.

### 4.1.2 Automated Configuration of MIP Solvers

Commercial tools for combinatorial problem solving usually come with many hyper-parameters, whose settings may have strong effects on the running time required for solving given problem instances. Deviating from the default and manually setting these performance parameters is a complex task that requires extensive domain knowledge and experimentation, and can be automated using algorithm configuration techniques, which are outlined in Section 2.3.

In this study, we use SMAC [45], a widely known, freely available, state-of-the-art configurator based on sequential model-based optimisation (also known as Bayesian optimisation). The main idea of SMAC is to construct and iteratively update a statistical model of target algorithm performance (specifically: a random forest regressor; [9]) to guide the search for good configurations. The random forest regressor allows SMAC to handle categorical parameters and therefore makes it suitable for MIP solvers, which have many configurable categorical parameters; SMAC has been shown to improve the performance of the commercial CPLEX solver over previous configuration approaches on several widely studied benchmarks [45].

### 4.1.3 Automatic Portfolio Construction

As mentioned in Section 2.4, for the configuration procedure to work effectively, the problem instances of interest have to be sufficiently similar, such that a configuration that performs well on a subset of them also performs well on others. In other words, the instance set should be homogeneous. If a given instance set does not satisfy this homogeneity assumption, automated configuration likely results in performance improvements on some instances, while performance on others might suffer, making it difficult to achieve overall performance improvements.

This problem can be addressed through *automatic portfolio construction* [116, 52, 78, 72]. The general concept behind automatic portfolio construction techniques is to create a set of algorithm configurations that are chosen such that they complement

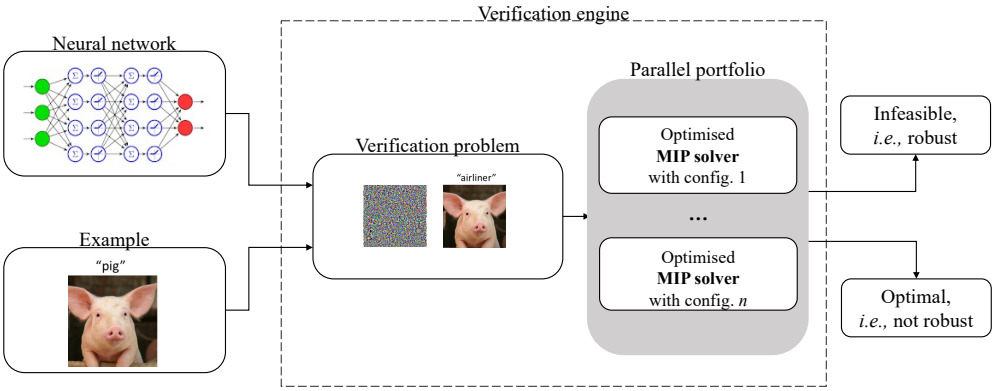
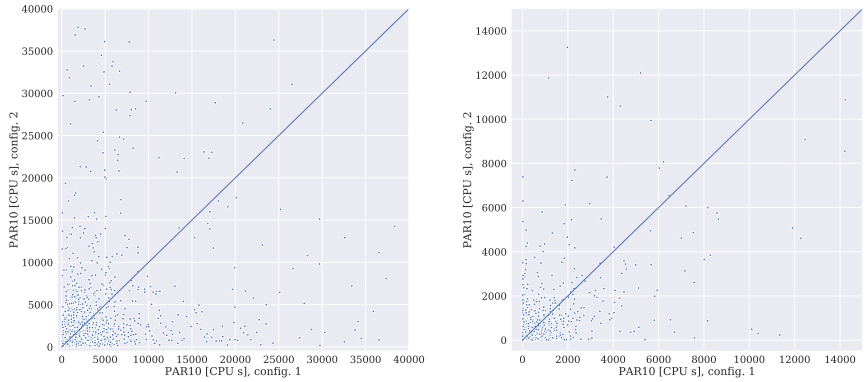



**Figure 4.1:** Schematic diagram of the proposed framework.

each other's strengths and weaknesses. This portfolio should then be able to exploit per-instance variation much more effectively than a single algorithm configuration, which is designed to achieve high overall performance but may perform badly on certain types or subsets of instances.

More specifically, Hydra [116] automatically constructs portfolios containing multiple instances of the target algorithm with different configurations. The key idea behind Hydra is that a new candidate configuration is scored with its actual performance only in cases where it works better than any of the configurations in the existing portfolio, but with the portfolio's performance in cases where it performs worse. Thereby, a configuration is only rewarded to the extent that it improves overall portfolio performance and is not penalised for performing poorly on instances for which it should not be run anyway. More details can be found in Chapter 2.4.

Once a portfolio has been constructed, there are essentially two ways to leverage the performance complementarity of the configurations contained in the portfolio. The first option is to extract instance-specific features and use those to train a statistical model that predicts the performance of each configuration in the portfolio individually. These predictions can then be used to select the configuration with the best-predicted performance (see, *e.g.*, Xu et al. [118]). Alternatively, all configurations can be run in parallel on a given problem instance, which implicitly ensures that we always benefit from the best-performing configuration in the portfolio, at the cost of increased use of parallel resources. An empirical comparison between both approaches has been presented by Kashgarani et al. [54].




**(a)** MIPVerify | mnistnet **(b)** Venus | mnistnet

**Figure 4.2:** Performance comparison of the configurations in the portfolios constructed for (a) MIPVerify and (b) Venus on the mnistnet benchmark. The plots show, that each configuration outperforms the other on some instances, while none of the configurations is dominating in performance across the entire benchmark set. This illustrates the complementary strengths of the configurations, which are exploited through portfolio construction. Note that there are also several instances on which one of the configurations reaches the time limit, but which are solved by the other. These are not shown in the figure due to the scaling of the axes. The diagonal line indicates equal performance of the two configurations.

## 4.2 Network Verification with Parallel MIP Solver Portfolios

In order to reduce complexity, [104] mainly focused on reducing the number of variables in the verification problem. On the other hand, [8] rely on pruning the search space during the branch-and-bound procedure. However, the embedded MIP solver and its numerous parameters were left untouched in both cases. More specifically, both methods employed a commercial MIP solver with default settings. This decision, along with their problem formulation, forms the starting point for our work.

More concretely, we seek to improve the performance of MIP-based neural network verification through configuring the MIP solver embedded in these systems, and constructing a portfolio of solver configurations optimised for the benchmark set at hand; Figure 4.1 provides an overview of the framework we propose. In brief, for a given network-example pair, we employ the verifier with several, differently configured instances of the embedded MIP solver. This portfolio of solvers is run in parallel and

finishes once one solver has returned a solution or a global time limit has been reached.

In the following sections, we describe details of the configuration procedure as well as the MIP solver we configured.

### 4.2.1 Configuration Procedure

In this study, we configure the commercial MIP solver Gurobi; see Section 4.2.2 for further details. Though it should be noted that, in principle, our approach works for any MIP solver.

The configuration procedure employs running Hydra over a predefined set of iterations to construct a portfolio of solver configurations with complementary strengths. The number of iterations is a hyper-parameter of the Hydra algorithm and has to be specified by the user. Since we cannot know the optimal portfolio size for a given benchmark in advance, we run Hydra over a reasonably larger number of iterations and, once the procedure has finished, discard configurations that did not improve portfolio performance on the validation set, *i.e.* that led to stagnation or reduction in total CPU time compared to the previous iteration. Note that the portfolio can contain the default configuration of the MIP solver.

Interestingly enough, we consistently observed strong heterogeneity among the instances in our benchmarks sets, making the use of a single configuration, *i.e.*, a portfolio of size 1, ineffective. This is illustrated in Figure 4.2: Employing two different configurations individually on the same benchmark set shows that none of them outperforms the other, *i.e.*, consistently achieves better performance across the entire set of instances. Combining both configurations into a portfolio, however, makes use of the complementary strengths of the configurations, and thereby achieves the highest overall performance, which motivates our choice of the portfolio approach.

Leveraging standard multi-core CPU architectures, we run the configurations in the portfolio in parallel until one of them returned a solution or until an overall limit on CPU time was exceeded. We note that, in principle, automated algorithm selection (see, *e.g.*, [66]) could be used to determine from this portfolio the configuration likely to solve any given instance most efficiently, though this requires substantial amounts of training data and creates uncertainty from sub-optimal choices made by the machine learning technique at the heart of such selection approaches.

### 4.2.2 MIP Solver

Following Tjeng et al. [104] and Botoeva et al. [8], we used the Gurobi MIP solver with a free academic license. Using the online documentation on Gurobi's parameters, we selected 62 performance-relevant parameters for configuration. These parameters can be categorical, *e.g.*, the simplex variable pricing strategy parameter can take the values {Automatic (-1), Partial Pricing (0), Steepest Edge (1), Devex (2), and Quick-Start Steepest Edge (3)}, or continuous, *e.g.*, the parameter controlling the magnitude of the simplex perturbation can take any value in the range $\{0, \infty\}$.

To control and limit the computational resources given to the solver, we fixed the number of CPU cores, *i.e.*, the parameter *Threads*, to the value of 1. Thereby, we also ensure that the solver is optimised in such a way that it uses minimal computational resources, which, in turn, allows for more efficient parallelisation. In contrast, the default value of this parameter is an automatic setting, which means that the solver will generally use all available cores in a machine. There are further parameters that have an automatic setting as one of their values. In those cases, we allowed for the "automatic" value to be selected, but also other values.

While configuring the MIP solver embedded in MIPVerify is a rather straightforward task, additional considerations arise when configuring the solver embedded in Venus. Essentially, Venus can run two modes, which lead to changes in the configuration space of the MIP solver: (*i*) Venus with *ideal cuts* and *dependency cuts* activated (default mode), in which case several cutting parameters are deactivated in Gurobi and therefore should be left untouched during the configuration procedure; (*ii*) Venus with its cutting mechanism deactivated, which allows for Gurobi's full parameter space to be optimised upon. Along with other, previously mentioned challenges, these considerations illustrate the complexity of adapting automated algorithm configuration techniques to the domain of neural network verification.

In order to maximally exploit the potential of automated hyperparameter optimisation, we decided to provide the configurator with full access to the configuration space and, thus, employ Venus with *ideal cuts* and *dependency cuts* deactivated and Gurobi's cutting parameters activated during portfolio construction.

## 4.3 Setup of Experiments

We test our method on several benchmarks, which will be introduced in the following, along with the objective of our configuration approach and the computational

environment in which experiments were carried out.

### 4.3.1 Configuration Objective

The objective of our configuration experiments is to minimise mean CPU time over all instances from the benchmark set. This choice deviates from the commonly used performance metric in the neural network verification literature, where evaluation is typically performed by operating on a fixed number of CPU cores while measuring wall-clock time. However, we do not consider wall-clock time a sensible performance measure when the evaluated methods use different numbers of cores. Instead, we decide to capture performance by means of CPU time, as it compensates for the possible difference in utilised cores. In other words, by choosing CPU time over wall-clock time, we ensure a more rigorous performance evaluation of our method as well as the baseline approaches, as one could easily gain performance in terms of wall-clock time through parallelisation, while heavily compromising in CPU time. Furthermore, we consider CPU time to be the more sensible performance measure, due to the cost associated with computational efforts. In fact, the rates for cloud services increase with the number of cores in a machine.

Generally, if the cost metric is running time, configurators typically optimise penalised average running time (PAR), notably PAR$k$, as the metric of interest, which penalises unsuccessful runs by counting runs exceeding the cutoff time $t_c$ as $t_c \times k$. In line with common practice in the algorithm configuration literature, we use $k = 10$ and refer to the cost metric as PAR10.

### 4.3.2 Details of the Configuration Procedure

The parameters for the configuration procedure were set as follows. Hydra ran over a predefined set of four iterations, during which it performed two independent runs of SMAC with a time budget of 24 hours each. Thus, running Hydra took $4 \times 2 \times 24 = 192$ hours for training, in addition to a variable amount of time spent on validation. In theory, the number of iterations could be set to a larger value; however, we refrained from this to keep our experiments within reasonable time frames. Lastly, we set $k = 1$, which means that after every run, Hydra added one configuration to the portfolio, i.e., the configuration that yielded the largest gain in overall training performance. The final output, therefore, is a portfolio containing a minimum number of 1 and a maximum number of 4 solver configurations.

### 4.3.3 Data

Our benchmark sets were comprised of randomly chosen verification problem instances created by MIPVerify and Venus, respectively, using the network weights of two MNIST classifiers as well as the property-network pairs from the ACAS Xu repository [51, 55]. ACAS Xu contains an array of neural networks trained for horizontal manoeuvre advisory in unmanned aircraft. The MNIST classifiers were taken from the works of [104] and [8], respectively, and used to cross-test each verifier on both networks. The ACAS Xu benchmark was chosen to find out whether a high diversity in networks (the ACAS Xu repository contains 45 different neural networks) poses any challenges to the configuration procedure.

***MNIST.*** Firstly, we created problem instances using the network weights of the robust classifier $\mathrm{SDP_dMLP_A}$ from [93]. Among the networks considered in the work of [104], we regard this one as the most difficult to verify, since it shows the largest average solving times and optimality gaps for many examples, even compared to classifiers trained on the typically more challenging CIFAR-10 benchmark. Secondly, we used the weights of the network mnistnet from the Venus repository [8], which is the only MNIST classifier considered in their study. In both cases, we created 184 instances, which were split 50-50 into disjoint training and validation sets. The training and validation sets were used during the configuration procedure, whereas the remaining 9 816 instances form the test set and were used to evaluate the final portfolio.

***ACAS Xu.*** For this benchmark, we only considered verification problem instances created by Venus, as MIPVerify at default reached the time limit of 38 400 CPU seconds for more than 80% of the instances. This makes automated configuration infeasible, as these instances do not only cause the default solver to time out but also any solver configuration tried by SMAC. Thereby, the configurator can hardly identify promising regions of the hyperparameter space and, consequently, not exploit them. Using Venus, we created 20 instances for different property-network pairs and, again, split them into disjoint training and validation sets. The remaining 152 instances are used for testing the final portfolio. Note that ACAS-Xu shows the highest average solving time among all benchmarks considered in the work of [8].

### 4.3.4 Execution Environment and Software Used

Our experiments were carried out on Intel Xeon E5-2683 CPUs with 32 cores, 40 MB cache size and 94 GB RAM, running CentOS Linux 7. We used MIPVerify version 0.2.3, Venus version 1.01, SMAC version 2.10.03, Hydra version 1.1 and the Gurobi

**Table 4.1:** Timeouts, adversarial error and PAR10 scores for different solver configurations of the MIP solver embedded in the MIPVerify engine on the MNIST dataset. Note that all approaches were given the same budget in terms of CPU time (the number of cores times the cutoff time). Using our portfolio, we achieved better performance than method of [104] as well as the default configuration of Gurobi using different numbers of cores. Boldfaced values indicate statistically significant improvements according to a binomial test with $\alpha = 0.05$ for timeouts and error bounds, and a permutation test with the number of permutations set at 10 000 and significance threshold of 0.05 for PAR10 scores.

| **Configuration** | **Cores** | **Cutoff** [Seconds] | **Timeouts** | **Adversarial Error** Lower Bound | Upper Bound | **PAR10** [CPU s] |
|---|---|---|---|---|---|---|
| $SDP_dMLP_A$ | | | | | | |
| Default | 32 | 1 200 | 21.29% | 14.37% | 30.67% | 39 772 |
| Default | 4 | 9 600 | 17.74% | 14.40% | 27.49% | 22 065 |
| Default | 1 | 38 400 | 17.66% | 14.36% | 27.58% | 20 117 |
| Portfolio | 4 | 9 600 | **14.96%** | 14.43% | **23.86%** | **8 478** |
| mnistnet | | | | | | |
| Default | 1 | 38 400 | 1.57% | 69.96% | 70.16% | 2 969 |
| Portfolio | 2 | 19 200 | **1.38%** | 70.13% | 70.14% | **1 844** |

solver version 9.0.1.

## 4.4 Results

We report empirical results for our new approach and each baseline in the form of (*i*) the fraction of timeouts; and (*ii*) bounds on adversarial error (the fraction of the dataset for which a valid adversarial example can be found), complement to adversarial accuracy (the fraction of the dataset known to be robust); (*iii*) CPU time (*i.e.*, PAR10 scores) on solvable instances, i.e., instances that were solved by our portfolio or any of the baselines within the given cutoff time. Aggregated performance numbers are presented in Table 4.1 for MIPVerify and Table 4.2 for Venus, whereas Figure 4.3 and Figure 4.4 visualise penalised running time of our portfolio approach against the baselines on an instance level. Generally, we determined statistical significance using a binomial test with $\alpha = 0.05$ for timeouts and error bounds, and a permutation test with the number of permutations set at 10 000 and significance threshold of 0.05 for PAR10 scores.

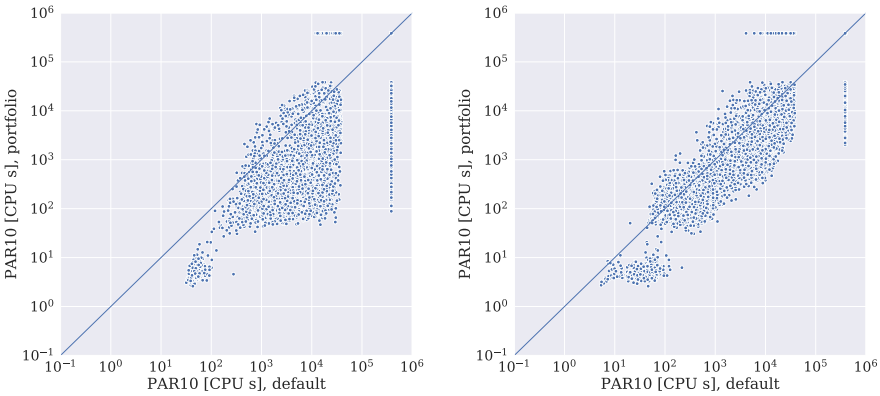
### 4.4.1 MIPVerify

The results from our configuration experiments on the $\text{SDP}_\text{d}\text{MLP}_\text{A}$ classifier are compared against multiple baselines. Firstly, we evaluated our portfolio approach against Gurobi, as used by Tjeng et al. [104], using all 32 cores per CPU available on our compute cluster, with the cutoff time set to $1\,200 \times 32 = 38\,400$ CPU seconds (*i.e.*, 1 200 seconds wall-clock time on a CPU without any additional load). In addition, since our parallel portfolio used 1 core for each of its 4 component configurations, we gathered additional baseline results from running the default configuration of Gurobi on the same number of cores and with the same cutoff as our portfolio, *i.e.*, $9\,600 \times 4 = 38\,400$ CPU seconds. Lastly, to maximise the number of instances processed in parallel, we considered Gurobi in its default configuration limited to a single CPU core, with cutoff time of 38 400 seconds. In short, we compared our approach against baselines with a variable number of cores and a constant budget in terms of CPU time. From these approaches, we considered only the best-performing one as the baseline for our configuration experiments on the mnistnet classifier.

As seen in Table 4.1, our portfolio was able to certify a statistically significantly larger fraction of instances, while reducing CPU time by an average factor of 4.7 on the solvable instances (8 478 *vs* 39 772 CPU seconds). Furthermore, the portfolio strongly outperformed this baseline in terms of timeouts (14.96% *vs* 21.29%). More concretely, 694 instances solved by the portfolio timed out in the default setup with 32 cores; see Fig 4.3a for more details. 1 435 instances were neither solved by the default nor the portfolio within the given time limit. 61 instances on which the portfolio timed out were solved by the default solver.

The default configuration of Gurobi running on 4 cores was also clearly outperformed by our portfolio in terms of CPU time (8 478 *vs* 22 065 CPU seconds). Furthermore, the portfolio was able to reduce the number of timeouts (14.96% *vs* 17.74%), while improving on the upper bound (23.86% *vs* 27.49%). In other words, the portfolio certified more instances using fewer computational resources, although it was provided with the same number of cores and overall time budget. Fig 4.3b shows per-instance results for this set of experiments. Here, the default solver timed out on 378 instances, which were solved by the portfolio. On 109 instances, only the portfolio timed out. On 1 374 instances, both setups resulted in timeouts.

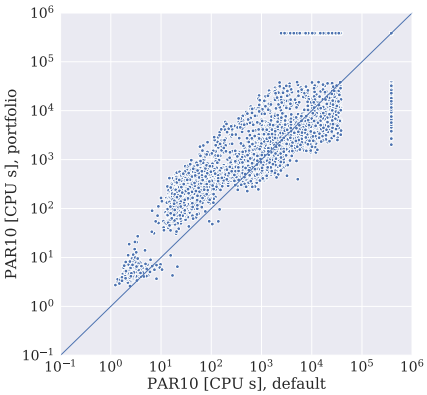
Lastly, we compared the portfolio against the default configuration of Gurobi running on a single-core. Here, our portfolio showed improved performance in terms of PAR10 (8 478 *vs* 20 117 CPU seconds) as well as the fraction of timeouts (14.96% *vs*



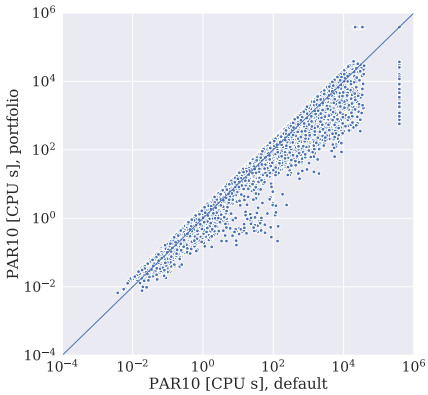

**(a)**
$\mathrm{SDP_dMLP_A}$
Cores = 32




**(b)**
$\mathrm{SDP_dMLP_A}$
Cores = 4




**(c)**
$\mathrm{SDP_dMLP_A}$
Cores = 1




**(d)**
mnistnet
Cores = 1

**Figure 4.3:** Evaluation of our parallel portfolio approach for MIPVerify on the MNIST dataset ($n$=10 000) using weights from the $\mathrm{SDP_dMLP_a}$ and mnistnet classifiers, respectively. Each dot represents a problem instance and the penalised running time for that instance achieved by the baseline approach (x-axis) *vs* our portfolio (y-axis). For $\mathrm{SDP_dMLP_a}$, the baselines we considered are (a) the default solver running on all available, *i.e.*, 32 cores, as in the work of [104], (b) the default solver running on 4 cores and (c) the default solver running on 1 core. Our parallel portfolio, using 4 cores, achieved substantially fewer timeouts than any of the baselines and lower CPU times (in terms of PAR10 scores). Points grouped at the top and right border represent instances for which the solver reached the time limit, and are measured according to their penalised running time values.

17.66%) and the upper bound (23.86% *vs* 27.58%). More specifically, the single-core default timed out on 378 instances that could be solved by the portfolio. On 108 instances, only the portfolio timed out. On 1 388 instances, both setups resulted in timeouts; see Fig 4.3c for more details.

On the mnistnet classifier, our portfolio also outperformed the single-core baseline in terms of PAR10 (1 844 *vs* 2 969 CPU seconds) as well as the fraction of timeouts (1.38% *vs* 1.57%), although to a smaller extent. To be precise, the default baseline timed out on 44 instances that the portfolio was able to solve (Fig 4.3d). On 25 instances, only the portfolio reached the time limit. 113 instances were neither solved by the default nor the portfolio. The default baseline timed out on 44 instances that the portfolio was able to solve. On 25 instances, only the portfolio reached the time limit. 113 instances were neither solved by the default nor the portfolio. These results could be explained by the mnistnet network being comparatively smaller and, thus, easier to verify than the $\mathrm{SDP_dMLP_A}$ classifier, as the latter results in a much larger number of timeouts when verified with equal settings.

### 4.4.2 Venus

The results from our configuration experiments are compared against two baseline approaches. Firstly, we evaluated our portfolio against Venus as employed by Botoeva et al. [8], *i.e.*, using the same hyperparameter settings for the verifier. We refer to this setup as *default**, as the MIP solver is left in its default configuration, while the verification engine is deployed with optimised hyperparameter settings. We note that the number of cores is equivalent to the number of parallel workers, which is set as a hyperparameter of the verifier. More precisely, we were running Venus using 2 workers, *i.e.*, 2 cores per CPU available on our compute cluster, with the cutoff time set to $7\,200 \times 2 = 14\,400$ CPU seconds. In this setup, Venus employs 2 instances of the MIP solver in parallel, while we ensured that each solver is using exactly 1 CPU core. This way, we are giving the same amount of resources to the verifier and the portfolio. It should be noted that for the ACAS Xu benchmark, we also ran Venus with the hyperparameter settings reported by Botoeva et al. [8], however with different numbers of workers. That is, we ran the verifier using 4 workers, 2 workers, and 1 worker, *i.e.*, CPU core(s), to assess the effects of parallelism, and found CPU time to be constant with regards to the number of workers running in parallel. We, therefore, consider each of these baselines to be equally competitive and only report results for Venus running with 2 active workers, *i.e.*, on 2 CPU cores and, thus, similar to the number of cores

**Table 4.2:** Timeouts, adversarial error and PAR10 scores for different configurations of the MIP solver embedded in the Venus engine on the MNIST and ACAS Xu datasets. Note that all approaches were given the same budget in terms of CPU time (the number of cores times the cutoff time). Using our portfolio, we achieved better performance than the method of [8]. Boldfaced values indicate statistically significant improvements according to a binomial test with $\alpha = 0.05$ for timeouts and error bounds, and a permutation test with the number of permutations set at 10 000 and significance threshold of 0.05 for PAR10 scores. The asterisk marks Venus runs using the hyperparameter settings suggested by Botoeva et al. [8], yet with Gurobi at default.

| **Configuration** | **Cores** | **Cutoff** [Seconds] | **Timeouts** | **Adversarial Error** Lower Bound | Upper Bound | **PAR10** [CPU s] |
|---|---|---|---|---|---|---|
| mnistnet | | | | | | |
| Default* | 2 | 7 200 | 1.63% | 70.33% | 71.96% | 1 975 |
| Portfolio | 2 | 7 200 | **0.58%** | 70.61% | 71.19% | **272** |
| $SDP_dMLP_A$ | | | | | | |
| Default | 1 | 14 400 | 9.76% | 14.36% | 24.12% | 6 534 |
| Portfolio | 2 | 7 200 | **6.10%** | 14.31% | **20.41%** | **636** |
| ACAS Xu | | | | | | |
| Default* | 2 | 7 200 | 1.75% | 20.34% | 22.09% | 1 314 |
| Portfolio | 2 | 7 200 | 1.17% | 20.34% | 21.21% | **443** |

utilised by the portfolio.

As there is no optimal setting of Venus hyperparameters provided for the $SDP_dMLP_A$ classifier, we used Venus with default settings as the baseline for our configuration experiments on this benchmark. In this setup, Venus is running with 1 active worker, which uses the same overall time budget of 14 400 CPU seconds.

As Table 4.2 shows, the portfolio strongly outperformed Venus with default* settings. On the mnistnet benchmark, it was able to certify a statistically significantly larger fraction of instances, while reducing CPU time by an average factor of 7.26 on the solvable instances (272 *vs* 1 975 CPU seconds). Furthermore, the portfolio strongly reduced the number of timeouts (1.63% *vs* 0.58%) on this benchmark. More specifically, the verifier timed out for 115 instances that were solved by the portfolio. On the other hand, the portfolio reached the time limit on 10 instances, which could be solved by the default. On 48 instances, both approaches resulted in timeouts; see Figure 4.4a for more details.

This baseline was also used to evaluate our portfolio approach on the ACAS Xu benchmark and, as previously mentioned, employed the verifier using the same hyperparameter settings as reported by Botoeva et al. [8], although with the number of workers or CPU cores fixed at 2. Essentially, the portfolio was able to slightly improve the number of timeouts and statistically significantly reduce CPU time by an average factor of 2.97 on the solvable instances (443 *vs* 1 314 CPU seconds). In concrete terms, the portfolio could solve 1 instance on which the default solver reached the time limit; see Figure 4.4c. For clarification, we achieved comparable performance gains over Venus running with 4 workers in parallel (443 *vs* 1 337 CPU seconds) as well as Venus running with 1 worker (443 *vs* 1 306 CPU seconds).

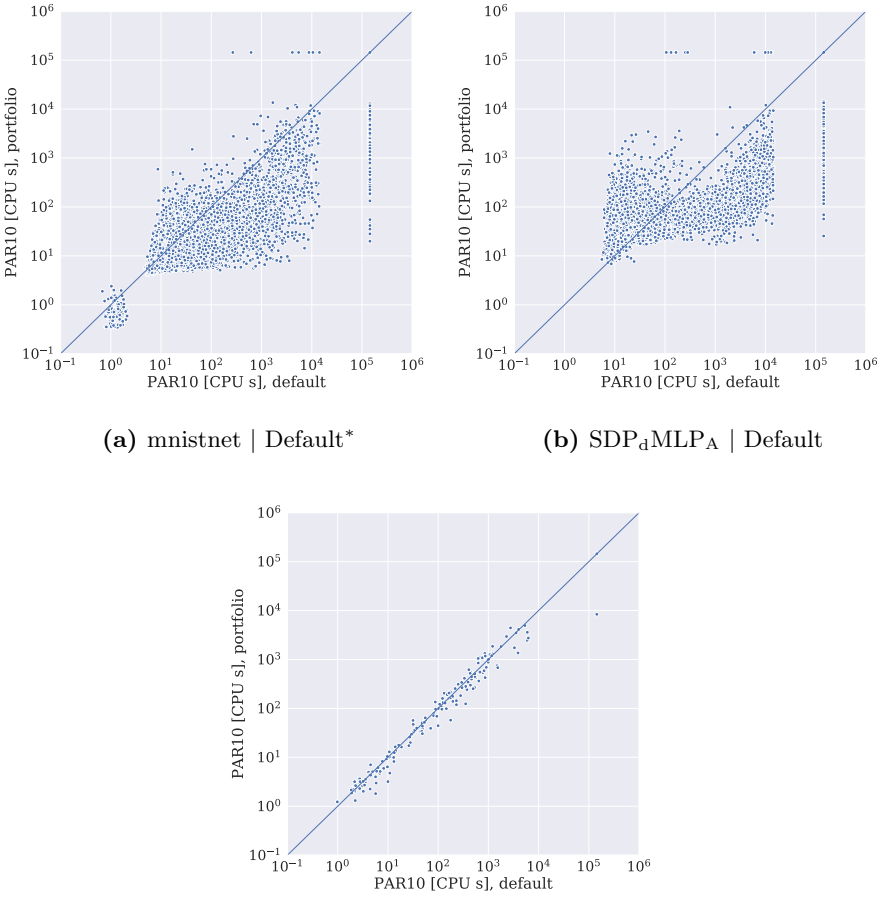
On the $\mathrm{SDP_dMLP_A}$ benchmark, the default baseline, *i.e.*, Venus with default settings, was outperformed by the portfolio in terms of PAR10 (636 *vs* 6 534 CPU seconds) as well as the fraction of timeouts (6.10% *vs* 9.76%). In this setup, the default timed out on 379 instances solved by the portfolio (Figure 4.4b). On 15 instances, only the portfolio reached the time limit. 597 instances were neither solved by the default nor the portfolio. Lastly, the portfolio strongly improved on the upper bound (20.41% *vs* 24.12%), which overall clearly demonstrates the strength of the portfolio approach.

## 4.5 Conclusions and Future Work

In this study, we have demonstrated the effectiveness of automated algorithm configuration and portfolio construction in the context of neural network verification, thereby providing an answer to the second research question (RQ2) of whether we can improve the performance of a MIP-based verification system by leveraging automated algorithm configuration techniques.

Applying these techniques to neural network verification is by no means a trivial extension, due to the high running times and heterogeneity of the problem instances to be solved. In order to address this heterogeneity, we constructed a parallel portfolio of optimised MIP solver configurations with complementary strengths. The potential of this method is supported by the notion of complementarity as explained in 3. Our method advises on the ideal number of configurations in the portfolio and can be used in combination with any MIP-based neural network verification system. We empirically evaluated our method on two recent, well known MIP-based verification systems, MIPVerify and Venus.

Our results show that the portfolio approach can significantly reduce the CPU time required by these systems on various verification benchmarks, while reducing the




**(a)** mnistnet | Default*

**(b)** $\text{SDP}_\text{d}\text{MLP}_\text{A}$ | Default

**(c)** ACAS Xu | Default*

**Figure 4.4:** Evaluation of our parallel portfolio approach for Venus on the MNIST dataset ($n$=10 000) using weights from the $\text{SDP}_\text{d}\text{MLP}_\text{a}$ and mnistnet classifiers, respectively, and on the 172 property-network pairs from the ACAS Xu benchmark. Each dot represents a problem instance and the penalised running time for that instance achieved by the verifier with the embedded MIP solver at default (x-axis) *vs* our portfolio (y-axis). Overall, our parallel portfolio achieved fewer timeouts than the baseline and lower CPU times (in terms of PAR10 scores).

number of timeouts and, thus, certifying a larger fraction of instances.

In more concrete terms, we strongly improved the performance of MIPVerify via speed-ups in CPU time by an average factor of 4.7 on the MNIST classifier $\mathrm{SDP_dMLP_A}$ from [93] and 1.61 on the MNIST classifier mnistnet from [8]. At the same time, we were able to lower the number of timeouts for both benchmarks and tighten previously reported bounds on adversarial error. For the Venus verifier, we achieved even larger improvements, *i.e.*, 10.3- and 7.26-fold reductions in average CPU time on the $\mathrm{SDP_dMLP_A}$ and mnistnet networks, respectively. Beyond that, we strengthened the performance of Venus on the ACAS Xu benchmark, attaining a 2.97-fold speedup in average CPU time. Overall, our results highlight the potential of employing MIP-based neural network verification systems with optimised solver configurations and demonstrate how our method can consistently improve neural network verifiers that make use of MIP solvers. At the same time, we note that our method is inherently dependent on the default performance of the verifier at hand. In other words, we acknowledge that this approach alone cannot scale existing methods to network sizes that are completely beyond the capabilities of these methods. However, our approach can significantly improve the running time of the verifier on the benchmarks it is able to certify, and thus moves the boundary of network/input combinations accessible to the verifier.

We see several fruitful directions for future work. Firstly, we plan to explore the use of per-instance algorithm configuration techniques to further reduce the computational cost of our approach. While our parallel portfolio approach is robust and makes good use of parallel computing resources, judicious use of per-instance algorithm selection techniques could potentially save some computational costs. We note that this will require the development of grounded descriptive attributes (so-called meta-features) for neural network verification, which we consider an interesting research project in its own right.

The neural network verification systems we considered in this study have additional hyperparameters. While our current approach focuses on the hyperparameters of the internal MIP solver, in future work, we will also configure the hyperparameters at the verification level. Due to the potential impact that this has on the MIP formulation and therefore on the running time of a given instance, this poses specific challenges for the algorithm configuration methods we use.

Finally, the portfolios we construct consist of multiple configurations of the same verification engine. In light of the results presented in Chapter 3, we could also consider heterogeneous portfolios that contain configurations of different verification engines,

which could lead to further improvements in the state of the art in neural network verification, and ultimately make it possible to verify networks far beyond the sizes that can be handled by the methods we have introduced here.

# Chapter 5

# Dynamic Algorithm Termination for Branch-and-Bound-based Neural Network Verification

As mentioned in previous chapters, much recent work has been concerned with the development of more efficient verification algorithms, *e.g.*, by employing the Branch and Bound (BaB) method for solving the verification problem [22, 11, 109, 12, 27] or by tightening bounds in the problem formulation using symbolic interval propagation [8, 40, 110, 111] and abstraction [4, 99, 119, 35, 98] techniques. However, even in light of recent developments, neural network verification remains a challenging and expensive computational task, especially as network complexity and dataset size increase.

Recall that neural network verification can be divided into *local* and *global* verification [100]. As in previous chapters, we focus on local robustness verification in this study. Local robustness verification typically considers a trained neural network, along with a set of inputs and a verification property specification. Considering the computational complexity of the verification problem, the required computational resources can grow substantially with the size of the network to be verified and the set of inputs.

We propose a novel approach enabling the efficient allocation of compute resources during the verification procedure. Specifically, we introduce a method to classify verification instances as solvable or unsolvable within a predefined time budget based on cheaply computable features. Furthermore, we operationalise these predictions to

terminate the verification procedure early for instances where a solution can not be obtained within the given time budget. Thereby, we avoid spending compute resources on attempting to verify instances that ultimately do not inform us about the robustness of the network. We evaluated our approach on a broad set of state-of-the-art verification algorithms and benchmarks, including benchmarks from recent VNN competitions [10, 87, 3], and show that we can reliably terminate verification runs for instances that are unsolvable within a given cutoff time without solving considerably fewer instances overall. In summary, the contributions of this chapter are as follows:

- We present features of branch-and-bound-based neural network verification instances that enable predictions about their solubility within a given time budget;
- we introduce a novel method based on those features that reliably identifies instances that cannot be solved within a given time budget;
- we evaluate our method on a broad set of benchmarks and across multiple verification tools;
- we show how this approach can be leveraged to terminate unsolvable instances early in the verification process, leading to savings of 64% in terms of running time on average with a comparable number of solved instances relative to the current state-of-the-art approach.

## 5.1 Method

As previously explained, solving the neural network verification task is computationally challenging. In addition, it is not known what makes certain instances harder to solve than others. Therefore, it cannot be decided *a priori* whether an instance could be solved successfully within a given time budget, potentially leading to ineffective resource allocation; *i.e.*, allocating compute time to unsolvable instances. In light of this, we leverage running time prediction techniques to classify instances as solvable or unsolvable within a given time budget. This allows us to greatly accelerate the verification procedure, by ensuring that resources are only spent on solvable instances.

### 5.1.1 Problem Formulation

As we are interested in the task of terminating instances that cannot be solved within a given time budget, we consider a binary classification problem. Hence, the vector $\mathbf{y}$

introduced in Chapter 2 contains as performance measures binary variables describing whether an instance has been solved or not.

Our goal is to reduce the computational burden demanded by the verification procedure, ensuring the most effective use of resources by not spending the full budget on unsolvable instances. In addition, we need to avoid classifying solvable instances as unsolvable; otherwise, the number of certified instances would be reduced, which might lead to inaccurate conclusions about the robustness of a given neural network.

### 5.1.2 Dynamic Algorithm Termination for BaB-based Neural Network Verification

As explained in Chapter 2, to perform running time prediction, we need to define a feature space $\mathcal{F}$ from which we can obtain a list of features $\mathbf{z}$ characterising a problem instance Specifically, we utilise cheaply computable features that, in part, relate directly to the internal operations of the given verifier.

We distinguish between *static* and *dynamic* features, where the former are computed only once and do not change during the solving process. Examples of static features include the lower bound obtained by an incomplete verification method at the beginning of the verification process. Conversely, dynamic features aim to capture the dynamically changing state of the verification algorithm at any point in time; examples include the current number of nodes in the BaB tree and the current global bounds. Generally, static features reflect the inherent complexity of the verification instance, while dynamic features capture the progress made thus far in solving the query. A detailed discussion of the features we have developed is provided later in this section.

To best leverage the evolving nature of our dynamic features, we propose a novel method that dynamically terminates verification queries when a classification model determines that the given instance will not be solved in the remaining time budget. We give a schematic overview of our method in Algorithm 1. The procedure is parameterised by the frequency $t_{\text{freq}}$ at which the current progress of the verification process is assessed to predict whether the given instance will be solved in the remaining time, and by the maximum allotted running time per instance, $t_{\text{cutoff}}$. We refer to the points in time at which the verification query is examined as a *checkpoint*. For each checkpoint, we train a classifier $\mathcal{C}_t$ with $t$ denoting the time of the checkpoint. $\mathcal{C}_t$ is trained on the feature values of the verification instances in the training set at time $t$ along with their corresponding label indicating whether the instances were solved within $t_{\text{cutoff}}$ seconds. In addition, we trained the classifier on verification instances from our training set that

were successfully solved before the current checkpoint with their feature values when the verification process was completed; thereby, the classifier can learn, which feature values define a completed instance.

Furthermore, our proposed method is configurable via a confidence parameter $\theta$, which defines the threshold that the prediction value for the positive class must exceed such that an instance is labelled accordingly. The incorporation of this parameter ensures that a user can choose whether the algorithm should stop potentially unsolvable instances as soon as possible ($\theta = 0.5$) or whether, in case of doubt, more information should be collected. The verification algorithm is then terminated only in case of a highly confident classifier prediction ($\theta = 0.99$). We note that $\theta$ can also be understood as a tuning parameter between *exploitation* and *exploration*. Therefore, $\theta$ should be chosen according to the user's needs, prioritising either a substantial reduction of the computational burden or a higher number of certified instances.

In summary, our method operates as follows. Given $t_{\text{freq}}$, $t_{\text{cutoff}}$, $\theta$, a verification algorithm, a neural network and a training set of verification instances, we initially collect feature values for each training instance at every checkpoint $t$ by executing the verification algorithm on each query for $t_{\text{cutoff}}$ seconds. In addition, we record whether the instance was solved or not. Subsequently, we train a classification model $C_t$ for every checkpoint $t$ on the collected data. During classification, given a verification query, we start by executing the verification algorithm for $t_{\text{freq}}$ seconds to collect an initial set of features for the given instance. Thereafter, we employ the classifier for the first checkpoint to predict whether the instance will be solved in the remaining time budget. If the confidence of this prediction exceeds $\theta$, we terminate the verification run for the given instance and record its result as *unknown*; otherwise, we continue the verification process for $t_{\text{freq}}$ seconds and update the dynamic instance features accordingly. Next, we query the classification model for the following checkpoint and decide whether to terminate the run. We repeat this process until the verification algorithm solves the instance under consideration or reaches the given cutoff time.

### 5.1.3 Static Instance Features

To perform running time prediction, we need to define instance features that allow us to make performance predictions for a given algorithm. We begin by introducing our proposed static features.

**Prediction margin ($\Delta$).** This feature is defined as the difference between the two highest class scores. *i.e.*, given a neural network $f$ with input $x_0 \in \mathcal{X}$ and corresponding

**Algorithm 1** Dynamic termination for BaB-based neural network verification

**Input:** Verification instance $(x_0, \epsilon)$; maximum per-instance running time $t_{\text{cutoff}}$; dynamic termination frequency $t_{\text{freq}}$; set of classifiers $\mathcal{C} = \{\mathcal{C}_{t_{\text{freq}}}, \mathcal{C}_{2t_{\text{freq}}}, \ldots, \mathcal{C}_{t_{\text{cutoff}}}\}$; confidence parameter $\theta$; verification algorithm VERIFY$((x, \epsilon), t_{\text{freq}})$ that pauses after $t_{\text{freq}}$ seconds to return the features and result of the instance.
**Output:** result or *unknown*
solved, features $\leftarrow$ VERIFY$((x_0, \epsilon), t_{\text{freq}})$
$t_{\text{elapsed}} \leftarrow t_{\text{freq}}$
**while** $\neg$ solved **and** $t_{\text{elapsed}} < t_{\text{cutoff}}$ **do**
    **if** $\mathcal{C}_{t_{\text{elapsed}}}$(features) $> \theta$ **then**
        **return** *unknown*
    **else**
        solved, features $\leftarrow$ VERIFY$((x_0, \epsilon), t_{\text{freq}})$
        $t_{\text{elapsed}} \leftarrow t_{\text{elapsed}} + t_{\text{freq}}$
    **end if**
**end while**
**return** solved

correct label $y_0 \in \mathcal{Y}$, we have $\Delta := f_{y_0}(x_0) - \max_{y \in \mathcal{Y} \setminus y_0} f_y(x_0)$, where $f_y$ refers to the output for class $y$. The prediction margin can be seen as a proxy for the closeness of the input image to the decision boundary. It heuristically captures how much change in the input space is required to change the neural network's prediction and, thus, the likelihood of an adversarial attack succeeding. This feature has recently been used in the context of adversarially robust model selection [64].

**Initial Incomplete Bound.** Each verifier we consider first attempts to solve the verification instance using an incomplete method. We utilised the resulting global upper and lower bounds as features.

**Improved Incomplete Bound.** If the initial problem bounds do not suffice to solve the problem, Oval and $\alpha\beta$-CROWN follow up with a tighter bounding method to further optimise last layer bounds. The initial and improved bounds give an estimate of how much improvement on the lower bound is realisable through (incomplete) bound optimisation methods. Furthermore, these bounds are the starting point for BaB and, thus, indicate the improvements required during BaB for solving the problem.

**Initial Percentage of Safe Constraints.** While VeriNet does not employ bound optimisation, the first call to the LP solver with the initial SIP bounds can already determine that some (or all) linear equations are unsatisfiable; these output constraints do then not have to be examined further during BaB. Thus, the percentage of initial safe constraints also provides an indication of the additional computation VeriNet will require subsequently.

**Adversarial Attack Margin.** Each of the considered verifiers initially carries out an adversarial attack that seeks to minimise the margin between the correct and incorrect classes. If the attack remains unsuccessful, its output can still be utilised to estimate the upper bound of the verification problem. Therefore, we included the adversarial attack margin, *i.e.*, the difference between the two highest scoring classes on the adversarial candidate, as an estimation of the upper bound of the given verification instance.

**Number of Unstable Neurons.** Lastly, we also included the absolute number of unstable neurons in our feature set. This number does not only indicate how many non-linearities have to be approximated but also bounds the maximum depth of the BaB tree.

### 5.1.4 Dynamic Instance Features

The *dynamic* features of BaB-based verification instances are subject to change during the BaB process, as they capture the progress made while solving the given problem instance.

**Branch Characteristics.** We included the number of *visited* branches that are already bounded as well as the total number of branches, also including those that have been created through branch splits but still need to be bounded. We further included the fraction of verified branches; these correspond to the leaves of the BaB tree and do not need to be split further. Once this number reaches a value of 1, the verification system has proven that the property holds.

**Current Global Bounds.** Furthermore, we included the current global bounds of the BaB tree. When the MIP formulation of the problem was solved by $\alpha\beta$-CROWN, we also recorded the resulting global bounds. This constitutes another way of capturing the progress of the given query, as once any global bound changes its sign, the verification process has been completed.

**Depth of the BaB Tree.** One important characteristic of the BaB tree that indicates instance complexity is its current depth, as it indicates how many neuron splits are present in the leaf nodes.

**Number of GPU Batches.** For Oval and $\alpha\beta$-CROWN, which perform the BaB algorithm in batches on a GPU, we included the number of batches that have been already computed. This feature enables a running time predictor to relate the BaB features to the internal operations of the verifiers.

**Batch Computation Time.** In addition, we computed the time used for the

computation of the last completed batch; this number indicates the computational hardness of the problem instance at hand, also in relation to the execution environment used for running the verifier. If feature collection occurs while a batch is still being processed, we additionally considered the computation time already spent on that batch.

### 5.1.5 Classification Model

For each checkpoint, we trained a random forest classifier with 200 decision trees and otherwise default hyperparameter settings. It has been shown in the past that random forests perform very well in the context of running time prediction tasks [48]. We also experimented with automatic hyperparameter configuration using *auto-sklearn* [30], but did not observe substantial improvements. Before training and classification, all features were standardised, *i.e.*, we removed the mean of each feature and scaled it to have unit variance, using the mean and standard deviation of each feature over the training set.

## 5.2 Experiments

We evaluated our approach on several benchmarks, which we will introduce in the following, along with details on the performance data collection and feature computation process.

Each benchmark was run on a compute cluster node equipped with two Intel Xeon Platinum 8480+ processors with 56 cores and a cache size of 105MB, 2TB of RAM and four NVIDIA H100 GPUs with 80GB of video memory, running Rocky Linux 9.4. Each run utilised 28 CPU cores, one GPU and 448GB of RAM.

### 5.2.1 Benchmarks

We considered a wide and diverse set of benchmarks taken from the ERAN repository [86, 99] and the VNN Competition [10, 87, 3], which have been commonly used by the neural network verification community [60, 10, 87, 3, 111, 99].

For our evaluation, we included two usage scenarios. First, we considered an approach aligned with an end-user's needs in assessing the robustness of a neural network. Here, we verified the correctly classified images from the first 1000 test set instances. We also included a competition scenario, where we generated problem instances according to the VNN Competition instance generation protocols.

In the first scenario, we included two convolutional (*Conv Big* and *Conv Small*) and two fully connected networks (*5 100* and *8 100*) trained on the MNIST dataset that were taken from the ERAN repository. For the CIFAR-10 dataset, we considered a small ResNet proposed by Wang et al. [111] (*ResNet 2B*). We verified the first 1000 test images against $l_\infty$ perturbations with $\epsilon$-values chosen in line with those used in previous studies [60, 111, 96].

For the second scenario, we included benchmarks directly taken from different editions of the VNN competition [10, 87, 3]. We employed the instance generation scripts provided in the competitions to generate 500 instances per benchmark that follow specific selection criteria such as correct classification or robustness against adversarial attacks. Concretely, we included the *Marabou*, *Oval21*, *SRI ResNet* and *ViT* benchmarks that consist of networks trained on the CIFAR-10 dataset. If the benchmarks included multiple networks or $\epsilon$ value specifications, we chose the configurations that yielded the most timeouts in the VNN competition, *i.e.*, the presumably most challenging problem instances.

Lastly, we included two benchmarks from the VNN Competition that consider the more complex CIFAR-100 and Tiny ImageNet datasets [87]. For both datasets, we chose the medium-size models for our evaluation. With this collection of networks and benchmarks, we ensured to include instances that have been studied extensively in the literature and that are challenging to solve by state-of-the-art verification tools.

### 5.2.2 Evaluation Setup

We first collected all performance data and feature values by running the verification tools and saving the result of the verification query, the consumed running time and the values of the considered instance features during the verification procedure. In Table 5.1, we report the number of solved instances and the running time for each verification tool and benchmark Missing values indicate that the benchmarks could not be used with the respective verification tools, due to unsupported network architectures.

We then evaluated our method by simulating it on the collected data. Generally, we followed a 5-fold cross validation protocol. To ensure that our training and testing sets were representative, we included in each fold the same proportion of verification instances solved before the first checkpoint, after the first checkpoint and unsolved instances; however, we only report metrics on instances that ran beyond the first checkpoint, as otherwise, we would predict timeouts after the instance has already been solved.

| Benchmark | # Inst. | $\alpha\beta$-CROWN | | VeriNet | | Oval | |
|---|---|---|---|---|---|---|---|
| | | # Solved | Time [GPU h] | # Solved | Time [GPU h] | # Solved | Time [GPU h] |
| 5 100 | 960 | 868 | 31.44 | 580 | 66.65 | 430 | 90.77 |
| 8 100 | 947 | 767 | 41.32 | 501 | 76.64 | 387 | 94.69 |
| Conv Big | 929 | 918 | 1.50 | 868 | 11.29 | 842 | 15.34 |
| Conv Small | 980 | 979 | 1.26 | 931 | 11.96 | 958 | 6.06 |
| ResNet 2B | 703 | 619 | 15.16 | 576 | 22.72 | - | - |
| Marabou | 500 | 193 | 51.73 | 176 | 54.28 | 187 | 53.32 |
| Oval21 | 500 | 210 | 50.47 | 158 | 58.45 | 201 | 52.50 |
| ViT | 500 | 251 | 41.86 | - | - | - | - |
| SRI ResNet A | 500 | 198 | 51.74 | 133 | 62.01 | - | - |
| CIFAR-100 | 500 | 361 | 24.73 | 279 | 40.25 | - | - |
| Tiny ImageNet | 500 | 421 | 14.63 | 356 | 29.47 | - | - |

**Table 5.1:** Overview of benchmarks used in our evaluation, including the number of certified instances and running time for each verification tool. Instances are from the first 1000 test set images for the first 5 benchmarks and otherwise from the VNN Competition [10, 87, 3] instance selection procedure. All experiments used a per-instance timeout of 600 seconds and GPU acceleration.

We evaluated the performance of our method in terms of *accuracy*, *true positive rate* (TPR) and *false positive rate* (FPR). The TPR reflects the fraction of correctly classified timeouts out of all unsolved instances while the FPR indicates the fraction of solved instances wrongly classified as timeouts out of all solved instances. On the convolutional networks for $\alpha\beta$-CROWN, some folds did not include true negatives or true positives. If these folds were used as the holdout set, we excluded them when computing the average TPR and FPR. In addition, we also compared our method to the standard verification procedure in terms of the overall number of solved instances (including those completed before the first checkpoint) and the required running time.

To run the verification algorithms, we used the configurations provided by the respective authors for their entries in the VNN Competitions. We chose a maximum running time of 600 seconds in wall-clock time per instance ($t_{\text{cutoff}} = 600s$) and predicted whether the instance will be solved within the remaining time budget every 10 seconds ($t_{\text{freq}} = 10s$). Lastly, we set the decision threshold $\theta$ to 0.99 to ensure that our method solves as many instances as possible.

| | $\alpha\beta$-CROWN | | | VeriNet | | | Oval | | |
|---|---|---|---|---|---|---|---|---|---|
| Benchmark | Acc. | TPR | FPR | Acc. | TPR | FPR | Acc. | TPR | FPR |
| 5 100 | 0.99 | 0.95 | 0.00 | 0.89 | 0.87 | 0.04 | 0.97 | 0.96 | 0.00 |
| 8 100 | 0.99 | 0.99 | 0.00 | 0.92 | 0.91 | 0.02 | 0.99 | 0.99 | 0.07 |
| Conv Big | 0.47 | 0.43 | 0.00 | 0.88 | 0.74 | 0.00 | 0.78 | 0.75 | 0.05 |
| Conv Small | 0.82 | 1.00 | 0.20 | 0.81 | 0.39 | 0.00 | 0.79 | 0.09 | 0.00 |
| ResNet 2B | 0.98 | 0.98 | 0.00 | 0.77 | 0.71 | 0.00 | - | - | - |
| Marabou | 0.99 | 0.99 | 0.10 | 0.93 | 0.95 | 0.53 | 0.96 | 0.96 | 0.13 |
| Oval21 | 0.97 | 0.98 | 0.05 | 0.89 | 0.88 | 0.07 | 0.96 | 0.95 | 0.03 |
| ViT | 1.00 | 1.00 | 0.00 | - | - | - | - | - | - |
| SRI ResNet A | 0.99 | 1.00 | 0.02 | 0.91 | 0.90 | 0.00 | - | - | - |
| CIFAR-100 | 0.99 | 1.00 | 0.03 | 0.85 | 0.79 | 0.00 | - | - | - |
| Tiny ImageNet | 0.98 | 0.99 | 0.03 | 0.88 | 0.67 | 0.00 | - | - | - |

**Table 5.2:** Results for timeout prediction with continuous feature collection in terms of accuracy, true positive and false positive rate as averages over five folds. We display results for $\theta = 0.99$, *i.e.*, the confidence threshold that must be reached before an instance is terminated.

## 5.3 Results and Discussion

In the following, we present results from our experimental evaluation of our dynamic algorithm termination method for the various verification algorithms we considered, and we show how our approach can be leveraged to allocate available resources more efficiently by terminating instances that will result in timeouts earlier in the verification process.

### 5.3.1 Classification Metrics

We report the classification metrics of our proposed method in Table 5.2 as averages over all five folds. We obtained very high TPR scores while maintaining a FPR close to 0 for most verifiers and benchmarks. Concretely, on average, our classifier correctly identified 85% of timeouts while incorrectly classifying 5% of solvable instances. Noticeably, across all verifiers, there were several benchmarks with TPRs above 90% and FPRs of almost zero. Lower TPR scores on the *Conv Big* and *Conv Small* benchmarks were due to the relatively small number of timeouts occurring in these benchmarks, leading to a lack of training examples for this class. Similarly, we observed higher FPR scores for the *Marabou* benchmark. This is likely due to the small number of queries solved after the first checkpoint, again resulting in less diverse training data.

| | $\alpha\beta$-CROWN | | | | VeriNet | | | | Oval | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Benchmark | Time [GPU h] | | # Solved | | Time [GPU h] | | # Solved | | Time [GPU h] | | # Solved | |
| 5 100 | 21.97 | (70%) | 868 | (±0) | 18.81 | (28%) | 576 | (-4) | 7.88 | (9%) | 430 | (±0) |
| 8 100 | 17.86 | (43%) | 766 | (-1) | 16.75 | (22%) | 500 | (-1) | 3.57 | (4%) | 386 | (-1) |
| Conv Big | 1.01 | (68%) | 918 | (±0) | 5.48 | (49%) | 868 | (±0) | 6.36 | (41%) | 841 | (-1) |
| Conv Small | 1.00 | (80%) | 969 | (-10) | 11.30 | (94%) | 931 | (±0) | 6.08 | (100%) | 958 | (±0) |
| ResNet 2B | 4.30 | (28%) | 619 | (±0) | 10.45 | (46%) | 576 | (±0) | - | - | - | - |
| Marabou | 2.47 | (5%) | 192 | (-1) | 6.36 | (12%) | 168 | (-8) | 4.97 | (9%) | 185 | (-2) |
| Oval21 | 7.57 | (15%) | 207 | (-3) | 15.04 | (26%) | 155 | (-3) | 10.02 | (19%) | 199 | (-2) |
| ViT | 2.00 | (5%) | 251 | (±0) | - | - | - | - | - | - | - | - |
| SRI ResNet A | 3.86 | (7%) | 197 | (-1) | 8.71 | (14%) | 133 | (±0) | - | - | - | - |
| CIFAR-100 | 5.10 | (21%) | 360 | (-1) | 19.37 | (48%) | 279 | (±0) | - | - | - | - |
| Tiny ImageNet | 4.58 | (31%) | 420 | (-1) | 19.92 | (68%) | 355 | (-1) | - | - | - | - |

**Table 5.3:** Results for dynamic termination of verification queries with $\theta = 0.99$. We display the running time and the number of solved instances accumulated over five folds. In parentheses, we provide the fraction of running time used and the difference in the number of solved instances compared to the standard verification procedure.

### 5.3.2 Dynamic Algorithm Termination

We display the results of our method in terms of total cumulative running time and number of solved instances aggregated over all folds for each benchmark and verifier in Table 5.3.

Most importantly, we obtained substantial speed-ups, while only a small amount of solvable instances was terminated prematurely. On average, our approach solved comparably many instances in 36% of the original running time. Notably, the largest acceleration occurred on the Marabou benchmark, where up to 95% of the standard running time could be saved. However, we also observed moderate penalties in terms of the absolute difference of solved instances for the *Marabou* benchmark on VeriNet and the *Conv Small* benchmark on $\alpha\beta$-CROWN, due to the reasons stated earlier. Moreover, on several benchmarks, all solvable instances were certified using substantially reduced running time; *e.g.*, the *5 100* benchmark for Oval and $\alpha\beta$-CROWN or the *ResNet A* benchmark for VeriNet.

Overall, we found that our approach substantially improves neural network verification in terms of running time of several verification algorithms on a broad range of benchmarks.

## 5.4 Conclusions and Future Work

In this study, we have shown that the computational resources demanded by neural network robustness verification can be greatly reduced by identifying and terminating runs on verification instances that will not be solved within their remaining time budget.

This answers the third research question (RQ3) of whether and to which extent we can predict the running time of a given verification algorithm for a specific problem instance: although it seems impossible to precisely extrapolate the running time to previously unseen instances, we show that a timeout can be reliably predicted for a given instance and time budget. Concretely, we showed that our method accelerates the verification procedure by 64% on average compared to the current state-of-the-art approach across a diverse set of benchmarks from the verification literature, while certifying a comparable number of instances. To predict whether an instance will be solved, we leveraged running time prediction techniques that employ novel static and dynamic features capturing both characteristics of the verification instance as well as features related to the internal operations of the given verifier.

The success of the proposed method was enabled by several design decisions. First, we leveraged the evolving nature of our dynamic features by regularly predicting timeouts throughout the verification procedure. Moreover, we included a confidence parameter $\theta$ that controls the threshold the prediction value of the timeout class must exceed before an instance is terminated. Using this parameter, a user can adjust the method to either prioritise savings in compute resources or a higher number of solved instances. We show that for a high value of $\theta$ our method substantially accelerates the verification procedure while solving comparably many instances as the standard verification.

In future work, we seek to extend our approach to further BaB-based verification approaches (*e.g.*, MN-BaB). Furthermore, we plan to investigate if our proposed features could be applied in other contexts, such as algorithm selection or satisfiability prediction. Lastly, we are interested in further studying the running time prediction capabilities of our features, possibly enabling empirical scaling models of BaB-based verification.

# Chapter 6

# Adversarially Robust Model Selection via Racing

In the previous chapters, we have introduced several meta-algorithmic approaches to formally verify the robustness of neural network models against perturbations in their inputs, such as the ones that occur in adversarial attacks. Nonetheless, this particular verification task remains computationally challenging.

In addition to other performance metrics of a neural network model, such as accuracy, one can compute *robust accuracy* by counting the fraction of inputs that are provably robust with regard to the given property. However, this adds significant overhead to the evaluation procedure, due to the high computational demands incurred by most formal verification algorithms, as explained earlier (see, *e.g.*, Chapter 2. This overhead grows substantially if multiple models are considered and compared against each other in (robust) performance; this challenge is not only faced by practitioners (*e.g.*, during model evaluation) but also encountered during Neural Architecture Search (NAS), where the goal is to select a suitable model from a large search space (see, *e.g.*, Elsken et al. [28]). In this context, adding robust accuracy as a selection criterion would hardly be feasible due to the large computational costs.

In this chapter, we seek to improve the efficiency of local robustness verification from a previously unexplored, meta-algorithmic point of view. Specifically, we propose a method to efficiently evaluate and compare the robustness of different neural network models (or variations of the same model) against adversarial attacks. Moreover, we consider the problem of selecting the most robust model, *i.e.*, the model with the

highest certified robust accuracy, from a given set of trained neural networks, whilst making the most efficient use of the computational budget.

In a nutshell, our proposed method employs a racing algorithm, in which a given set of neural network models are subjected to local robustness verification with respect to adversarial attacks. After each input iteration, the performance of each network (in terms of robust accuracy) is measured, and the verification procedure stops for a given network as soon as its robust accuracy is lower than the robust accuracy obtained by its competitors. Racing approaches are well studied and have already been successfully employed in other, resource-intense domains, such as hyperparameter optimisation [45, 7].

Complementary to the racing approach, we propose a novel sampling strategy based on the likelihood of a given input instance being adversarially robust. Essentially, this strategy prioritises input instances during the verification procedure that are most likely to expose vulnerabilities of the neural network model and, therefore, provide valuable insights into its robustness after fewer input iterations of the verification procedure and, hence, at a lower computational cost. At the same time, it reduces the risk of selecting sub-optimal models, which might show higher robust accuracy than other candidate models after verifying some randomly sampled input instances but might perform worse overall. In fact, when using random sampling, the only way to mitigate this risk would be to increase the number of input iterations – with the associated costs involved.

To enable the proposed sampling strategy, we must define or estimate the likelihood of a network input being adversarially robust. In our case, this involves estimating the proximity of a network input to the decision boundaries of the model, captured by means of $\Delta$-values, which will be explained in the following. Using this strategy, we can bias the sampling towards inputs for which adversarial attacks are most likely to occur. Although the relation between adversarial examples and the decision boundary of a neural network has been extensively studied [23, 120, 20, 38, 74], we are not aware of any existing work leveraging these insights in the context of local robustness verification procedures. In summary, the main contributions of this chapter are as follows:

- We propose an efficient model selection method based on a novel heuristic that quantifies the likelihood of a network being adversarially robust with respect to a given input;

- we introduce $\Delta$-values, which serve as a proxy for the distance of an input instance to the decision boundaries of a neural network model;

- we provide statistical evidence demonstrating significant differences in the empirical cumulative distribution of $\Delta$-values between robust and non-robust instances;
- we evaluate our method on two diverse sets of neural networks trained on the MNIST and CIFAR-10 datasets, achieving a 108-fold reduction in cumulative running time for MNIST networks and a 42-fold reduction for CIFAR-10 networks.

## 6.1 Adversarially Robust Model Selection

Given a set of neural network models $\mathcal{N} = \{N_1, N_2, \ldots, N_m\}$ and a set of input instances $\mathcal{X}$, the objective is to identify the model $N^* \in \mathcal{N}$ that maximises robust accuracy with respect to adversarial attacks. Notice that robust accuracy is computed by verifying all instances $x \in \mathcal{X}$, measuring the fraction of cases in which the model correctly classifies an instance and adversarial input perturbations do not change the original output produced by the model.

To address this problem, we propose a method with two main components: a racing approach and a sampling strategy based on a sorting mechanism for the input instances on which the network is verified. We considered two variants of the racing approach. The first one is a naïve racing approach in which the best-performing candidate models are selected at every input iteration, whereas the second one represents an adaptation of the *F-Race* algorithm [7], which gathers statistical evidence against some candidate models before they are discarded. Both variants of the racing approach as well as our proposed sorting mechanism will be further explained in the following.

### 6.1.1 Naïve Racing Approach

Generally, the idea of a racing approach is to evaluate a finite set of candidate models while allocating the computational resources among them in a systematic way (see, *e.g.*, [79]). To do so, the racing approach verifies step-by-step each candidate model in the given set, where in this context, a step corresponds to an input instance on which the neural network models are verified. At each step, all the remaining candidate models are verified, possibly in parallel, and candidate models are discarded once they are outperformed by others, *i.e.*, once one or more networks have obtained a higher robust accuracy.

An overview of this approach, which we refer to as the naïve racing approach for the remainder of this chapter, can be found in Algorithm 2. After each iteration over the input instances, it identifies the model with the highest robust accuracy (determined

**Algorithm 2** Racing approach for robust model selection

**Require:** Trained neural network models $\mathcal{N} = \{N_1, N_2, \ldots, N_m\}$; Network input instances $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$; Verification algorithm $\textsc{Verify}(N_i, x_j)$ that returns *sat*, *unsat* or *unsolved*;

**Ensure:** Model with highest robust accuracy: $N_{\text{selected}}$

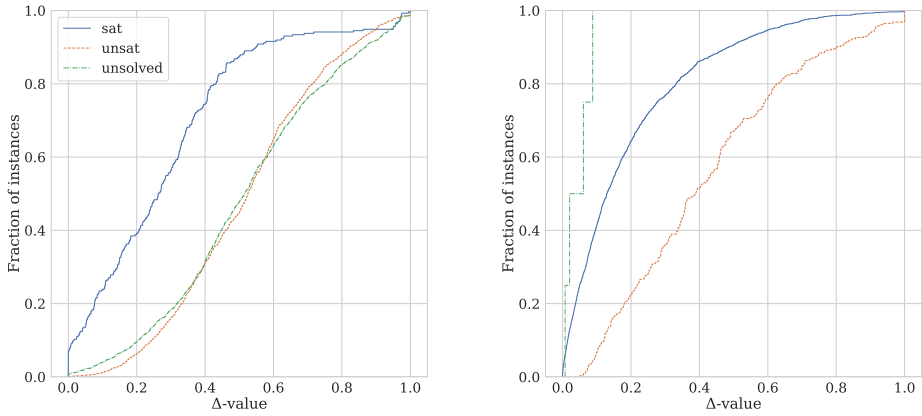
1: $\mathcal{C} \leftarrow \mathcal{N}$
2: $u_i \leftarrow 0$ with $i = 1, 2, \ldots, |N|$
3: **for all** $x_j \in \mathcal{X}$ **do**
4:  **for all** $N_i \in \mathcal{C}$ **do**
5:  **if** $\textsc{Verify}(N_i, x_j)$ is *unsat* **then**
6:  $u_i \leftarrow u_i + 1$
7:  **end if**
8:  **end for**
9:  $\mathcal{C} \leftarrow \{ N_i \mid i \in \arg\max_i \{u_i\} \}$
10: **end for**
11: Randomly select one element $N_{\text{selected}}$ from set $\mathcal{C}$
12: **Return** $N_{\text{selected}}$

based on $u_i$ which represents the number of unsat instances for network $N_i$) and updates the set of candidate models $\mathcal{C}$ accordingly. Notice that the selection criterion on line 9 can by virtue of the arg max operator return a set of multiple networks. Moreover, $u_i$ increases whenever a network $N_i$ is found to be robust, *i.e.*, *unsat*, w.r.t. to a given input. On the other hand, an instance that is misclassified by the model would be considered as *sat*. The algorithm stops once all input instances have been processed, and the final output is the model with the highest determined robust accuracy.

### 6.1.2 F-Race

An important aspect of the model selection problem outlined above is that it can be viewed as a stochastic problem. In fact, although the process of formally verifying the behaviour of a neural network model with respect to certain input instances is deterministic (*i.e.*, multiple runs on the same input instance will always lead to the same result), its outcome depends on the particular instance to which it is applied. Concurrently, the specific instance being verified can be regarded as having been sampled from an underlying probability distribution, which may be unknown. For the naïve racing approach, this could lead to models being prematurely discarded after a few input iterations, even if that model would achieve the highest robust accuracy overall, *i.e.*, if it was verified with respect to all available input instances.

To address this stochasticity, the authors of [7] proposed *F-Race*, a widely known,




**(a)** MNIST (VeriNet, $\epsilon = 0.04$) **(b)** CIFAR ($\alpha\beta$-CROWN, $\epsilon = 0.008$)

**Figure 6.1:** Empirical cumulative probability distribution of normalised $\Delta$-values for *sat*, *unsat* and unsolved instances for the considered MNIST and CIFAR networks, respectively. Notably, the plot shows a statistically significant difference between the empirical distribution functions of $\Delta$-values for *sat* and *unsat* instances. Specifically, for both MNIST and CIFAR networks, *sat* instances generally have smaller $\Delta$-values than *unsat* instances. Statistical significance is determined by means of a Kolmogorov–Smirnov test with a significance threshold of 0.05.

state-of-the-art racing algorithm. F-Race can be considered an extension of the naïve racing approach, where the naïve selection criterion (line 9 in Algorithm 2) is replaced with a statistical test. Concretely, after each iteration over the input instances, F-Race performs a statistical test, typically, the non-parametric Friedman test, to determine if there are significant differences in the number of *unsat* instances per neural network model. If the null hypothesis is rejected or, in other words, significant differences exist, F-Race applies post-tests to identify the models which are performing statistically significantly worse than the best, and updates $\mathcal{C}$ accordingly. The algorithm stops when all input instances have been processed, and the final output is the model with the highest robust accuracy.

Since we are interested in the fraction of instances that are *unsat*, we used Cochran's Q test to determine if there are significant differences among the *unsat* counts for each of the networks. Notice that the Cochran's Q test is identical to the Friedman test but applicable when the responses are binary. When only two candidate networks remain, we used the McNemar test (without continuity correction), which can be seen as a

special case of Cochran's Q test [102]. Any significant Cochran's Q (or McNemar) statistic is followed by Dunn's post-hoc test with a significance threshold of p= 0.05, and networks are selected if they have a significantly higher certified robust accuracy than their competitors.

### 6.1.3 Sorting Mechanism

In addition, we propose a sampling strategy based on a mechanism that sorts the considered input instances according to their likelihood of being adversarially robust. The key idea behind this mechanism is that by exposing a neural network model to inputs that are least likely to be adversarially robust, we can more quickly gather insights into its vulnerability or, similarly, its robustness. In other words, if we initially verify a neural network model on its most "challenging" input instances, *i.e.*, instances on which it is most likely not robust, but obtain robustness guarantees for these instances, we can at least heuristically assume the model to also be robust with respect to the remaining instances.

To enable this sorting mechanism, we must define or estimate the likelihood of a neural network model input being adversarially robust. In our case, this involves estimating their distance from the decision boundaries of the model, captured by means of network outputs. Intuitively, if an input lies very close to the boundary between two classes, it can be assumed that small perturbations, such as those applied to adversarial examples, have a higher chance to change the prediction made by the model.

In this study, we estimate the distance to an adjacent class boundary as the difference between the neural network output corresponding to the most likely class and that corresponding to the second-most likely class, and we refer to this difference as $\Delta$. Formally, we define $\Delta := \max(\{y_1, y_2, \ldots, y_n\}) - \max(\{y_1, y_2, \ldots, y_n\} \setminus \max(\{y_1, y_2, \ldots, y_n\}))$, where $y_n$ refers to the network output for a given class $n$. Based on the resulting $\Delta$-values, we can, for each neural network model individually, sort the input instances in an non-decreasing order, where the smaller the value of $\Delta$, the closer we assume an instance to lie to an adjacent class boundary.

## 6.2 Setup of Experiments

We compiled two sets of neural network models: one set consisting of 31 neural networks trained on the MNIST dataset and one set containing 27 neural networks trained on the CIFAR-10 dataset. All networks were taken from the repository of the ERAN

verification system [85, 96, 99, 97, 98] and greatly vary in terms of architecture, training method as well as robust accuracy. Details of the considered networks can be found in the supplementary material. We verified each network for local robustness with respect to the first 100 instances in the test set of the MNIST and CIFAR-10 datasets, respectively.

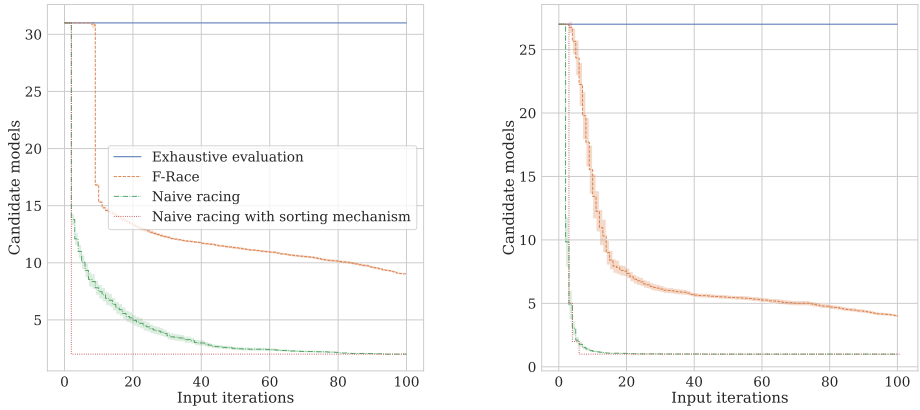
To verify the MNIST networks, we used the state-of-the-art complete CPU-based verification algorithm VeriNet [40] with a perturbation radius of $\epsilon = 0.04$, which lies well within the range of commonly chosen values for $\epsilon$ when verifying networks trained on MNIST [114, 8, 109]. Verification queries ran with a time budget of 3600 seconds on a cluster of machines equipped with Intel Xeon E5-2683 CPUs with 32 cores, 40 MB cache size and 94 GB RAM, running CentOS Linux 7.

To verify the more challenging CIFAR networks, we used $\alpha\beta$-CROWN, a state-of-the-art complete GPU-accelerated verification method [111]. For these networks, we verified local robustness with $\epsilon = 0.008$, a value in line with commonly chosen values of $\epsilon$ for networks trained on CIFAR (see, *e.g.*, [87]). Again, all verification queries ran with a time budget of 3600 seconds on machines equipped with NVIDIA GeForce GTX 1080 Ti GPUs with 11 GB video memory. Overall, the verification of the CIFAR networks used in our study consumed 558 hours in GPU time, whereas the verification of the MNIST networks demanded a total of 1380 CPU hours.

We note that, although the verification algorithms presented above are complete, they were sometimes unable to solve an instance due to time or memory limitations; we report such instances as unsolved.

## 6.3 Empirical Results

In the following, we will compare our proposed selection method against the F-Race approach, the naïve racing approach as well as selection based on exhaustive evaluation. The latter represents the conceptually simplest baseline for selecting the most robust model from a given set of neural network models. Using this approach, each model is verified with respect to all available input instances during the verification procedure. At each input iteration, the candidate model with the highest certified robust accuracy is selected as the incumbent; *i.e.*, the model that would be returned if the process were terminated at the given iteration. Differently from the racing approaches, the exhaustive evaluation approach does not eliminate any candidate models during the selection process; therefore, when run to completion, it will always achieve a regret of zero.

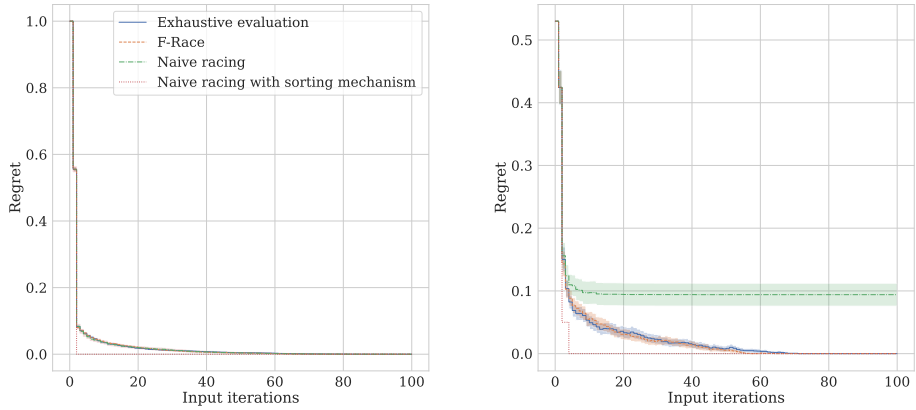



**(a)** MNIST (VeriNet, $\epsilon = 0.04$) **(b)** CIFAR ($\alpha\beta$-CROWN, $\epsilon = 0.008$)

**Figure 6.2:** Number of candidate models as determined by each method after every input iteration. For the three methods that do not use the sorting mechanism, the line represents the average number of candidate models over 200 random input orders, each with different random seeds, along with along with the respective 95% confidence intervals. Clearly, naïve racing, coupled with our proposed sorting mechanism, reduces the number of candidates after substantially fewer input iterations than other methods. Notice that selection based on exhaustive evaluation does not eliminate models from the set of candidates, which, therefore, does not decrease in size.

We evaluated each method in terms of cumulative running time and regret. The former describes the total running time consumed by the verification algorithm until all given input instances have been processed and the most robust model has been determined. Regret, in the context of model selection, describes the difference between the performance of the selected model and the performance of the best model that could have been chosen based on complete and perfect knowledge. In other words, it represents the loss incurred by selecting a sub-optimal model.

Formally, the regret $R$ is defined as follows. Suppose we have a set of candidate models $\mathcal{C} = \{C_1, C_2, \ldots, C_n\}$, and we want to select one model from this set based on certified robust accuracy. Let $C_{\text{best}}$ be the best model in the set, *i.e.*, the model with the highest certified robust accuracy $ra$. Then, $R := ra(C_{\text{best}}) - ra(C_{\text{selected}})$, where $ra(C_{\text{best}})$ represents the robust accuracy of the best model and $ra(C_{\text{selected}})$ the robust accuracy of the selected model.



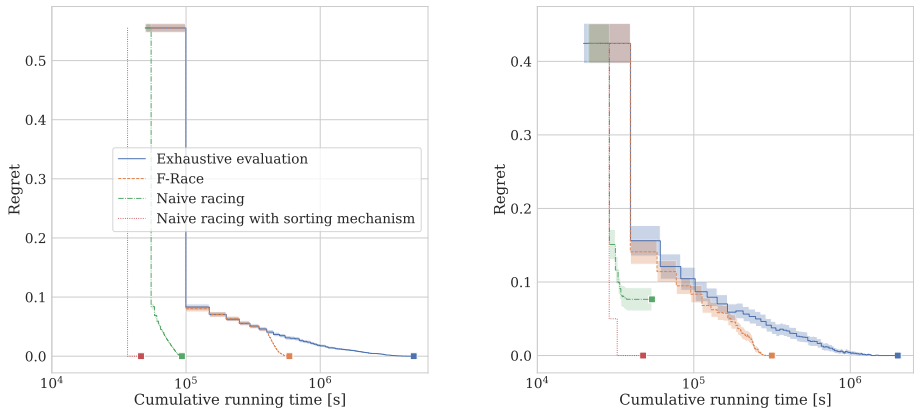

**(a)** MNIST (VeriNet, $\epsilon = 0.04$) **(b)** CIFAR ($\alpha\beta$-CROWN, $\epsilon = 0.008$)

**Figure 6.3:** Regret achieved by the considered methods, where regret describes the difference between the performance of the selected model and the performance of the best model that could have been chosen given all available information. For methods not using the sorting mechanism, the regret was averaged over 200 random input orders, each with different random seeds, and is shown with a 95% confidence interval. The plots show that naïve racing, coupled with our proposed sorting mechanism, achieves optimal regret with fewer input iterations than other methods.

### 6.3.1 Local Robustness at the Decision Boundary

First of all, we investigated the relationship between the local robustness of a neural network model and the estimated distance of an input instance from the decision boundary of the model. More specifically, we examined the empirical cumulative probability distribution of $\Delta$-values across all considered models, giving rise to 3100 individual verification problem instances for MNIST and 2800 for CIFAR. Remember that $\Delta$-values serve as a proxy for the distance of an instance from the closest adjacent class boundary. We normalised these values per network under consideration.

The empirical cumulative distribution of the $\Delta$-values is visualised in Figure 6.1. Notice that some instances could not be verified due to timeouts or memory limitations; we show these instances as unsolved. The plots clearly show that *sat* instances, *i.e.*, instances for which an adversarial example could be found, tend to have a smaller $\Delta$-value than those that are *unsat*, *i.e.*, robust. The difference in distributions is determined as statistically significant by means of a Kolmogorov–Smirnov test with a standard significance threshold of 0.05.




**(a)** MNIST (VeriNet, $\epsilon = 0.04$) **(b)** CIFAR ($\alpha\beta$-CROWN, $\epsilon = 0.008$)

**Figure 6.4:** Regret as a function of cumulative running time for each of the considered methods. Running time represents wall-clock time on the machine on which the experiments were carried out. For methods not using the sorting mechanism, the regret was averaged over 200 random input orders, each with different random seeds, and is shown with a 95% confidence interval. The plots show that naïve racing, coupled with our proposed sorting mechanism, achieves optimal regret while using substantially less compute time than other methods. Each line ends once a specific method has processed all given input instances.

At the same time, Figure 6.1 shows that there exist instances, which are found to be *sat* despite having a relatively large $\Delta$-value, *i.e.*, a $\Delta$-value close to the end of the (normalised) range of values. Upon further investigation, we found that for MNIST, such instances occurred for 12 out of the 31 neural network models we considered and for 15 out of the 27 CIFAR networks. Notice that for these models, no instance was found to be robust, which indicates that large $\Delta$-values can occur also for *sat* instances if a neural network model generally suffers from poor robustness. However, this observation does not affect the performance of our proposed selection method, as models which are non-robust with respect to any input instance would be discarded from the set of candidate models early in the selection process regardless of their $\Delta$-value and, hence, the sorting of input instances. Notice that when removing these neural network models from the set, the difference in $\Delta$-values between *sat* and *unsat* instances grows even larger; more details can be found in the supplementary material.

### 6.3.2 Evaluation of Our Proposed Selection Method

We evaluated our proposed selection method, naïve racing coupled with the sorting mechanism, in terms of cumulative running time and regret, and compared its performance against the following three baselines: (i) F-Race, (ii) naïve racing without a sorting mechanism and (iii) selection based on exhaustive evaluation. For methods that do not employ the sorting mechanism (*i.e.*, all baselines), we repeated the selection process 200 times, where each time the order of the input instances was based on a different random seed. We report the average running time over all runs, along with the respective 95% confidence intervals.

Figure 6.2a displays the size of the set of candidate networks trained on the MNIST dataset throughout the selection process. It shows that our proposed selection method reduces the number of candidate models after fewer iterations compared to each considered baseline. At the same time, for the exhaustive evaluation approach, the number of considered models remains constant, resulting in a larger number of queries that need to be performed at every input iteration.

As the number of candidate models reduces very quickly, it could be assumed that the aggressive nature of our selection method might lead to a sub-optimal outcome of the model selection process. We investigated this potential trade-off and show the results in Figure 6.3a. As can be seen, every method reached an optimal regret, indicating that the significant speed-up does not necessarily compromise on the quality of the selection process. However, we note that some of the MNIST networks were found to be fully robust. These are, consequently, always selected by any of the selection methods, even those that are more aggressive. Lastly, notice that F-Race eliminates candidate models based on statistical evidence, which can lead to models being selected that are less robust than others but where this difference is not found to be statistically significant at the given iteration.

We also tested our method on networks trained on the more challenging CIFAR dataset. Neural networks trained on this dataset are generally more difficult to verify than those trained on the MNIST dataset [71]. Figure 6.2b shows the size of the set of candidate CIFAR networks throughout the selection process. Again, we found that our proposed selection method eliminates candidate models after fewer iterations compared to other methods. Concurrently, the difference between the naïve racing approach with and without the sorting mechanism is much smaller than the difference observed on MNIST networks.

However, Figure 6.3b shows the advantage of the sorting mechanism: the naïve

racing approach using the sorting mechanism very quickly converges towards an optimal regret, while other methods either require substantially more iterations or do not reach the optimum at all. In fact, on this set of models, the naïve racing approach without the sorting mechanism always resulted in a sub-optimal model choice. Overall, these results clearly demonstrate that our new method can effectively select the most robust model, and does so in a more efficient way than F-Race, which discards models only after it obtained statistical significance between the robust accuracy of the candidate models.

Lastly, we studied in more detail the efficiency of our method compared to the baselines we considered, in terms of regret achieved for a specific time budget. This is visualised in Figure 6.4a for MNIST networks and Figure 6.4b for CIFAR networks. Notably, these plots reveal that for both sets of models, our method selects the best-performing, *i.e.*, most robust model while demanding less compute time than any of the considered baselines, especially selection based on exhaustive evaluation. In fact, for MNIST networks, the cumulative running required to complete the selection process is reduced by several orders of magnitude, *i.e.*, a 108-fold speedup factor, when compared to selecting based on exhaustive evaluation (1380.93 *vs* 12.83 hours). Furthermore, for CIFAR networks, our selection method achieved a 41-fold speedup compared to the exhaustive evaluation approach (558.44 *vs* 13.18 hours). Generally, this decrease in cumulative running time occurs because our selection method iteratively eliminates models from the set of candidates, subsequently reducing the number of verification queries in the following iterations, as previously explained. We note that the number of verification queries directly depends on the number of models, which decreases throughout the selection process.

These results highlight that our proposed selection method is well-suited for scenarios in which computing resources are limited, as it is likely to select, within any given amount of running time, models that are more robust than those determined by the baselines considered in our study.

## 6.4 Conclusions and Future Work

In this chapter, we have demonstrated the effectiveness of advanced model selection techniques in the context of neural network verification. Specifically, we studied the problem of selecting the most robust neural network model from a given set of models, whilst reducing the compute time needed to obtain robustness certificates for the given input instances.

To enable our proposed selection method, we introduced a novel sorting mechanism based on the likelihood of an input instance being robust with respect to adversarial input perturbations. This likelihood is captured by means of $\Delta$-values, which serve as a proxy for the distance of an input instance to the model decision boundaries, and we present statistical evidence indicating significant differences in the empirical cumulative distribution of these values for robust and non-robust instances. Overall, our method guides the allocation of computing resources required to perform local robustness verification towards adversarially robust models and can, in principle, be used in combination with any verification system.

We empirically evaluated our method on two diverse sets of 31 and 27 neural networks, trained on the MNIST and CIFAR-10 datasets, respectively. Our results clearly show that our proposed model selection method significantly reduces the cumulative running time required to select the most robust neural network model from these sets. Thereby, we provide an answer to the fourth research question (RQ4) of how to efficiently select the neural network model from a given set of models that achieves the highest certified robust accuracy. Specifically, compared to the exhaustive evaluation approach, our method achieved a speedup factor of 108 for the set of MNIST networks and a speedup factor of 42 for the set of CIFAR networks while still selecting the most robust model.

In future work, we plan to apply our method to other verification tasks (*e.g.*, robustness verification under bias field perturbations), network architectures and datasets, and to perform a systematic analysis of the relationship between $\Delta$-values and the robustness of neural network models. In addition, we are interested in the precise relationship between the $\Delta$-value and the distance to the nearest decision boundary.

# Chapter 7

# Conclusions and Outlook

In this chapter, we revisit the research questions presented in Chapter 1 and provide a detailed discussion on how each question has been addressed. Lastly, we discuss directions for future research.

## 7.1 Answers to Research Questions

In Chapter 1, we presented four research questions that guided the exploration and structure of this thesis. In the following, we will revisit each research question, outlining the methods we used to address them and summarising our key findings.

*RQ1: What constitutes the state of the art in neural network verification?*

In Chapter 3, we performed a comprehensive performance analysis of various CPU- and GPU-based neural network verification methods and revealed an algorithmic landscape characterised by significant performance diversity across different types of verification problem instances. From this study, we concluded that no single verifier consistently outperforms other methods in every scenario, challenging the notion of a universally superior algorithm within the neural network verification domain. Instead, we observed high levels of complementarity, *i.e.*, instances solved by one verifier that other verifiers could not solve and vice versa, quantified by means of marginal contribution and Shapley values. Moreover, these findings highlight the complex nature of neural network verification problems and suggest the usage of algorithm portfolios for optimal verification outcomes. Notice that these findings are in line with those for

other NP-hard problems; *e.g.*, the work of Leyton-Brown et al. [70].

*RQ2: How can we leverage automated algorithm configuration techniques to improve the performance of a MIP-based verification system?*

In Chapter 4, we investigated the application of automated algorithm configuration techniques to enhance the operational efficiency of MIP-based verification systems. Notably, we observed strong heterogeneity among different problem instances, which is not handled well by standard configuration approaches. Instead, we employed advanced portfolio construction techniques, which combine different solver configurations with complementary strengths into a parallel portfolio. This portfolio runs the solver configurations in parallel, stopping each configuration as soon as one of them has returned a solution. This implicitly ensures the we always benefit from the best-performing algorithm in the portfolio. Notably, we achieved substantial improvements in terms of running time and an increased number of successfully solved instances, despite the increased overhead demanded by the parallel portfolio. These outcomes highlight the potential of automated configuration and portfolio construction techniques as a fruitful approach for improving the performance of combinatorial solvers employed in the context of neural network verification systems, thereby streamlining the verification procedure overall. This confirms findings from previous work on related problems, notably mixed integer linear programming [46, 44, 45, 76].

*RQ3: To which extent can we predict the running time of a given verification algorithm for a specific problem instance?*

In Chapter 5, we investigated the possibilities of running time prediction for neural network verification algorithms. While we found that precise running time prediction (*i.e.*, regression) remains a challenging task, we developed a timeout prediction model that anticipates the feasibility of completing a verification query within a predefined time budget. This was enabled by newly defined features of the problem instance and the verification algorithm in use. Using this approach, we achieved a more efficient allocation of computational resources, strongly enhancing the overall efficiency of the verification procedure. Furthermore, our timeout prediction method could be leveraged in the context of parallel algorithm portfolios or, more specifically, per-instance algorithm selection; given several algorithms that run in parallel on a specific instance, our method could terminate those that are not able to solve the instance in the given

time budget. However, the verification algorithms we considered in our study did not display sufficient performance complementarity on the benchmarks we used. This could be explained by the fact that the verification algorithms were employed with configurations specifically tailored to the given benchmarks.

*RQ4: How can we efficiently select the neural network model from a given set of models that achieves the highest certified robust accuracy?*

In Chapter 6, we introduced a novel racing algorithm designed to guide the selection process of the most robust neural network model from a set of candidate models. In this context, we proposed a novel heuristic that captures the likelihood of a given instance to be robust or non-robust. Using this heuristic, we can guide the search towards neural network models that are most likely to show a high adversarial robustness. We found that our proposed solution significantly reduces the computational costs typically associated with model selection by iteratively eliminating less promising candidates, thereby facilitating a more efficient selection process of the optimal, *i.e.*, most robust model. This approach presents a practical solution to the challenge of robust model selection, ensuring computational resources are utilised judiciously while selecting the model with the highest robustness.

## 7.2 Directions for Future Research

With the work presented in this thesis, we sought to enable future progress in the field of neural network verification. These methods offer great potential for obtaining safety guarantees for neural-network-based AI systems, which is a crucial requirement for their use in high-risk domains, such as medical diagnosis or advanced driver assistant systems. At the same time, computational complexity remains a major challenge and current methods do not scale to complex architectures, such as large language models.

One general direction involves expanding this work to encompass a broader spectrum of neural network architectures and verification problems beyond local robustness for image classification models. Such an expansion would not only further validate the generalisability of the findings presented in this thesis but also potentially reveal new insights and challenges that could further refine the state of the art in neural network verification. While we considered only local robustness properties in this thesis, mainly due to their prominence in the literature and the availability of suitable benchmarks and solvers, these properties do not capture semantic changes or domain-specific noise

models; these would require different distance metrics that take into account the dependencies among input variables and, possibly, novel approaches for reasoning over the resulting properties.

Furthermore, we focused on the automated configuration of MIP-based verification systems. At the same time, verification algorithms have additional hyperparameters, also unrelated to MIP solvers; *e.g.*, configuring $\alpha\beta$-CROWN [111, 119] gives rise to several choices ranging from the selection of a bounding method to the number of branches for non-linear branching. Automatically configuring these algorithms could lead to substantial performance improvements and enhance usability, given the vast hyperparameter space presented by some of these methods.

Another fruitful direction for future work lies in the enhancement of running time prediction models. Enabling running time regression could provide a more nuanced understanding of the verification process, leading to improved resource allocation and process efficiency. Specifically, it would be desirable to predict the running time needed to solve a specific instance beyond a given cutoff point. This would, firstly, require to study the behaviour of verification algorithms when supplied with large time budgets to see if and when hard instances get solved and, potentially, the definition additional features.

In addition, applying the insights and methodologies developed in this thesis to a variety of real-world scenarios holds considerable promise, such as medical diagnosis. Such applications would not only test the practical implications of the research but also uncover new challenges and opportunities for innovation in the field of neural network verification. For example, assessing the robustness of a neural network model used for the classification of ECG measurements requires the definition of robustness properties with respect to specific noise models, such as baseline wander or power-line interference [81]. Furthermore, the scalability of verification methods to neural network models used in practice would be interesting to study.

Finally, the methods and findings presented in this thesis could be jointly utilised in the context of neural architecture search. Given the task of finding a neural network architecture that achieves a high level of robustness, it becomes necessary to perform verification as efficiently as possible, as multiple network architectures or configurations need to evaluated to guide the search process. This would require well-calibrated verifiers as well as efficient resource allocation, and a joint framework that can handle both the neural architecture search process as well as the verification system.

Overall, these future research directions hold the potential to significantly advance the research are of neural network verification, building on the contributions of this

thesis to explore new frontiers in neural network verification and automated machine learning. This can ultimately foster the employment of neural-network-based AI systems in safety-critical tasks, as neural network verification provides a method to formally prove that the system behaves as intended for a given operational domain. In fact, proving the safety (and, specifically the robustness) of an AI System is demanded by the European AI Act [106], underlining the relevance and importance of the concepts and methods introduced in this thesis.

# Bibliography


[1] Michael Akintunde, Alessio Lomuscio, Lalit Maganti, and Edoardo Pirovano. Reachability Analysis for Neural Agent-Environment Systems. In *Proceedings of the 16th International Conference on Principles of Knowledge Representation and Reasoning (KR2018)*, pages 184–193, 2018.

[2] Syed Muhammad Anwar, Muhammad Majid, Adnan Qayyum, Muhammad Awais, Majdi Alnowami, and Muhammad Khurram Khan. Medical image analysis using convolutional neural networks: a review. *Journal of medical systems*, 42:1–13, 2018.

[3] Stanley Bak, Changliu Liu, and Taylor Johnson. The Second International Verification of Neural Networks Competition (VNN-COMP 2021): Summary and Results. *arXiv preprint arXiv:2109.00498*, 2021.

[4] Stanley Bak, Hoang-Dung Tran, Kerianne Hobbs, and Taylor T. Johnson. Improved Geometric Path Enumeration for Verifying ReLU Neural Networks. In *Proceedings of the 32nd International Conference on Computer Aided Verification (CAV 2020)*, pages 66–96, 2020.

[5] Thomas Bartz-Beielstein, Carola Doerr, Daan van den Berg, Jakob Bossek, Sowmya Chandrasekaran, Tome Eftimov, Andreas Fischbach, Pascal Kerschke, William La Cava, Manuel Lopez-Ibanez, et al. Benchmarking in Optimization: Best Practice and Open Issues. *arXiv preprint arXiv:2007.03488*, 2020.

[6] Osbert Bastani, Yani Ioannou, Leonidas Lampropoulos, Dimitrios Vytiniotis, Aditya Nori, and Antonio Criminisi. Measuring Neural Net Robustness with Constraints. In *Advances in Neural Information Processing Systems 29 (NeurIPS 2016)*, pages 2613–2621, 2016.

[7] Mauro Birattari, Zhi Yuan, Prasanna Balaprakash, and Thomas Stützle. F-Race and Iterated F-Race: An Overview. In Thomas Bartz-Beielstein, Marco Chiarandini, Luís Paquete, and Mike Preuss, editors, *Experimental Methods for the Analysis of Optimization Algorithms*, pages 311–336. Springer, 2010.

[8] Elena Botoeva, Panagiotis Kouvaros, Jan Kronqvist, Alessio Lomuscio, and Ruth Misener. Efficient Verification of ReLU-based Neural Networks via Dependency

Analysis. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI-20)*, pages 3291–3299, 2020.

[9] Leo Breiman. Random Forests. *Machine Learning*, 45(1):5–32, 2001.

[10] Christopher Brix, Stanley Bak, Changliu Liu, and Taylor T Johnson. The Fourth International Verification of Neural Networks Competition (VNN-COMP 2023): Summary and Results. *arXiv preprint arXiv:2312.16760*, 2023.

[11] Rudy Bunel, Jingyue Lu, Ilker Turkaslan, Philip H. S. Torr, Pushmeet Kohli, and M. Pawan Kumar. Branch and Bound for Piecewise Linear Neural Network Verification. *Journal of Machine Learning Research*, 21(42):4795–4804, 2020.

[12] Rudy Bunel, Ilker Turkaslan, Philip Torr, Pushmeet Kohli, and Pawan K Mudigonda. A Unified View of Piecewise Linear Neural Network Verification. In *Advances in Neural Information Processing Systems 31 (NeurIPS 2018)*, pages 1–10, 2018.

[13] Nicholas Carlini, Guy Katz, Clark Barrett, and David L Dill. Provably Minimally-Distorted Adversarial Examples. *arXiv preprint arXiv:1709.10207*, 2017.

[14] Nicholas Carlini and David Wagner. Towards Evaluating the Robustness of Neural Networks. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (IEEE S&P 2017)*, pages 39–57, 2017.

[15] Marco Casadio, Ekaterina Komendantskaya, Matthew L. Daggitt, Wen Kokke, Guy Katz, Guy Amir, and Idan Refaeli. Neural Network Robustness as a Verification Property: A Principled Case Study. In *Proceedings of the 34rd International Conference on Computer Aided Verification (CAV 2022)*, pages 219–231, 2022.

[16] Pin-Yu Chen, Yash Sharma, Huan Zhang, Jinfeng Yi, and Cho-Jui Hsieh. EAD: Elastic-Net Attacks to Deep Neural Networks via Adversarial Examples. In *Proceedings of the 32nd AAAI Conference on Artificial Intelligence (AAAI-18)*, pages 10–17, 2018.

[17] Chih-Hong Cheng, Georg Nührenberg, and Harald Ruess. Maximum Resilience of Artificial Neural Networks. In *Proceedings of the 15th International Symposium on Automated Technology for Verification and Analysis (ATVA2017)*, pages 251–268, 2017.

[18] Marco Chiarandini, Chris Fawcett, and Holger H Hoos. A Modular Multiphase Heuristic Solver for Post Enrolment Course Timetabling. In *Proceedings of the 7th International Conference on the Practice and Theory of Automated Timetabling (PATAT 2008)*, 2008.

[19] Jeremy Cohen, Elan Rosenfeld, and Zico Kolter. Certified Adversarial Robustness via Randomized Smoothing. In *Proceedings of the 36th International Conference on Machine Learning (ICML 2019)*, pages 1310–1320, 2019.

[20] Francesco Croce, Maksym Andriushchenko, and Matthias Hein. Provable Robustness of ReLU networks via Maximization of Linear Regions. In Kamalika Chaudhuri and Masashi Sugiyama, editors, *Proceedings of the 22nd International Conference on Artificial Intelligence and Statistics (AISTATS 2019)*, pages 2057–2066, 2019.

[21] George B. Dantzig. Linear Programming. *Operations Research*, 50(1):42–47, 2002.

[22] Alessandro De Palma, Rudy Bunel, Alban Desmaison, Krishnamurthy Dvijotham, Pushmeet Kohli, Philip H. S. Torr, and M. Pawan Kumar. Improved Branch and Bound for Neural Network Verification via Lagrangian Decomposition. *arXiv preprint arXiv:2104.06718*, 2021.

[23] Gavin Weiguang Ding, Yash Sharma, Kry Yik Chau Lui, and Ruitong Huang. MMA training: Direct input space margin maximization through adversarial training. In *Proceedings of the 8th International Conference on Learning Representations (ICLR 2020)*, pages 2057–2066, 2020.

[24] Yinpeng Dong, Fangzhou Liao, Tianyu Pang, Hang Su, Jun Zhu, Xiaolin Hu, and Jianguo Li. Boosting Adversarial Attacks With Momentum. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2018)*, pages 9185–9193, 2018.

[25] Souradeep Dutta, Susmit Jha, Sriram Sankaranarayanan, and Ashish Tiwari. Output Range Analysis for Deep Neural Networks. In *Proceedings of the Tenth NASA Formal Methods Symposium (NFM 2018)*, pages 121–138, 2018.

[26] Krishnamurthy Dvijotham, Robert Stanforth, Sven Gowal, Timothy A Mann, and Pushmeet Kohli. A Dual Approach to Scalable Verification of Deep Networks. In *Proceedings of the 38th Conference on Uncertainty in Artificial Intelligence (UAI 2018)*, pages 550–559, 2018.

[27] Ruediger Ehlers. Formal Verification of Piece-Wise Linear Feed-Forward Neural Networks. In *Proceedings of the 15th International Symposium on Automated Technology for Verification and Analysis (ATVA 2017)*, pages 269–286, 2017.

[28] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural Architecture Search: A Survey. *The Journal of Machine Learning Research*, 20(1):1997–2017, 2019.

[29] Claudio Ferrari, Mark Niklas Mueller, Nikola Jovanović, and Martin Vechev. Complete Verification via Multi-Neuron Relaxation Guided Branch-and-Bound. In *Proceedings of the 10th International Conference on Learning Representations (ICLR 2022)*, pages 1–15, 2022.

[30] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and Robust Automated Machine

Learning. In *Advances in Neural Information Processing Systems 28 (NeurIPS 2015)*, pages 2962–2970, 2015.

[31] Matthias Feurer, Jost Tobias Springenberg, and Frank Hutter. Initializing Bayesian Hyperparameter Optimization via Meta-Learning. In *Proceedings of the 29th AAAI Conference on Artificial Intelligence (AAAI–15)*, pages 1128–1135, 2015.

[32] Matteo Fischetti and Jason Jo. Deep neural networks and mixed integer linear optimization. *Constraints*, 23(3):296–309, 2018.

[33] Alexandre Fréchette, Lars Kotthoff, Tomasz Michalak, Talal Rahwan, Holger Hoos, and Kevin Leyton-Brown. Using the shapley value to analyze algorithm portfolios. In *Proceedings of the 30th AAAI Conference on Artificial Intelligence (AAAI-16)*, pages 3397–3403, 2016.

[34] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Torsten Schaub, Marius Thomas Schneider, and Stefan Ziller. A Portfolio Solver for Answer Set Programming: Preliminary Report. In *Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR2019)*, pages 1–6, 2011.

[35] Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin Vechev. AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (IEEE S&P 2018)*, pages 3–18, 2018.

[36] Carla P Gomes and Bart Selman. Algorithm portfolios. *Artificial Intelligence*, 126(1–2):43–62, 2001.

[37] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and Harnessing Adversarial Examples. In *Proceedings of the 3rd International Conference on Learning Representations (ICLR 2015)*, pages 1–11, 2015.

[38] Warren He, Bo Li, and Dawn Song. Decision Boundary Analysis of Adversarial Examples. In *Proceedings of the 6th International Conference on Learning Representations (ICLR 2018)*, pages 1–15, 2018.

[39] Patrick Henriksen, Kerstin Hammernik, Daniel Rueckert, and Alessio Lomuscio. Bias Field Robustness Verification of Large Neural Image Classifiers. In *Proceedings of the 32nd British Machine Vision Conference 2021 (BMVC 2021)*, pages 202–2016, 2021.

[40] Patrick Henriksen and Alessio Lomuscio. Efficient Neural Network Verification via Adaptive Refinement and Adversarial Search. In *Proceedings of the 24th European Conference on Artificial Intelligence (ECAI 2020)*, pages 2513–2520, 2020.

[41] Holger H. Hoos and Thomas Stützle. *Stochastic Local Search: Foundations & Applications*. Elsevier / Morgan Kaufmann, 2004.

[42] Bernardo A. Huberman, Rajan M. Lukose, and Tad Hogg. An Economics Approach to Hard Computational Problems. *Science*, 275(5296):51–54, 1997.

[43] Frank Hutter, Domagoj Babic, Holger H Hoos, and Alan J Hu. Boosting Verification by Automatic Tuning of Decision Procedures. In *Proceedings of Formal Methods in Computer Aided Design (FMCAD'07)*, pages 27–34, 2007.

[44] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Automated Configuration of Mixed Integer Programming Solvers. In *Proceedings of the 7th International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming (CPAIOR 2010)*, pages 186–202, 2010.

[45] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential Model-Based Optimization for General Algorithm Configuration. In *Proceedings of the 5th International Conference on Learning and Intelligent Optimization (LION 5)*, pages 507–523, 2011.

[46] Frank Hutter, Holger H Hoos, Kevin Leyton-Brown, and Thomas Stützle. ParamILS: An Automatic Algorithm Configuration Framework. *Journal of Artificial Intelligence Research*, 36:267–306, 2009.

[47] Frank Hutter, Marius Lindauer, Adrian Balint, Sam Bayless, Holger Hoos, and Kevin Leyton-Brown. The Configurable SAT Solver Challenge (CSSC). *Artificial Intelligence*, 243:1–25, 2017.

[48] Frank Hutter, Lin Xu, Holger H Hoos, and Kevin Leyton-Brown. Algorithm Runtime Prediction: Methods & Evaluation. *Artificial Intelligence*, 206:79–111, 2014.

[49] Kai Jia and Martin C. Rinard. Efficient Exact Verification of Binarized Neural Networks. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33 (NeurIPS 2020)*, pages 1782–1795, 2020.

[50] Kyle D Julian, Mykel J Kochenderfer, and Michael P Owen. Deep Neural Network Compression for Aircraft Collision Avoidance Systems. *Journal of Guidance, Control, and Dynamics*, 42(3):598–608, 2019.

[51] Kyle D Julian, Jessica Lopez, Jeffrey S Brush, Michael P Owen, and Mykel J Kochenderfer. Policy compression for aircraft collision avoidance systems. In *Proceedings of the 35th Digital Avionics Systems Conference (DASC2016)*, pages 1–10, 2016.

[52] Serdar Kadioglu, Yuri Malitsky, Ashish Sabharwal, Horst Samulowitz, and Meinolf Sellmann. Algorithm Selection and Scheduling. In *Proceedings of the Seventeenth International Conference on Principles and Practice of Constraint Programming (CP2011)*, pages 454–469, 2011.

[53] Manisha M Kasar, Debnath Bhattacharyya, and TH Kim. Face recognition using neural network: a review. *International Journal of Security and Its Applications*, 10(3):81–100, 2016.

[54] Haniye Kashgarani and Lars Kotthoff. Is Algorithm Selection Worth It? Comparing Selecting Single Algorithms and Parallel Execution. In *AAAI Workshop on Meta-Learning and MetaDL Challenge*, pages 58–64, 2021.

[55] Guy Katz, Clark Barrett, David L Dill, Kyle Julian, and Mykel J Kochenderfer. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In *Proceedings of the 29th International Conference on Computer Aided Verification (CAV 2017)*, pages 97–117, 2017.

[56] 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. The Marabou Framework for Verification and Analysis of Deep Neural Networks. In *Proceedings of the 31st International Conference on Computer Aided Verification (CAV 2019)*, pages 443–452, 2019.

[57] Konstantin Kaulen, Matthias König, and Holger H Hoos. Dynamic algorithm termination for branch-and-bound-based neural network verification. In *To appear in Proceedings of the 39th AAAI Conference on Artificial Intelligence (AAAI-25)*, pages 1–9, 2025.

[58] Pascal Kerschke, Holger H Hoos, Frank Neumann, and Heike Trautmann. Automated Algorithm Selection: Survey and Perspectives. *Evolutionary Computation*, 27(1):3–45, 2019.

[59] Matthias König, Annelot W Bosman, Holger H Hoos, and Jan N van Rijn. Critically Assessing the State of the Art in CPU-based Local Robustness Verification. In *Proceedings of the Workshop on Artificial Intelligence Safety 2023 (SafeAI 2023) co-located with the Thirty-Seventh AAAI Conference on Artificial Intelligence (AAAI2023)*, pages 1–9, 2023.

[60] Matthias König, Annelot W Bosman, Holger H Hoos, and Jan N van Rijn. Critically Assessing the State of the Art in Neural Network Verification. *Journal of Machine Learning Research*, 25(12):1–53, 2024.

[61] Matthias König, Holger H Hoos, and Jan N van Rijn. Speeding up neural network robustness verification via algorithm configuration and an optimised mixed integer linear programming solver portfolio. *Machine Learning*, 111(12):4565–4584, 2022.

[62] Matthias König, Holger H Hoos, and Jan N van Rijn. Towards Algorithm-Agnostic Uncertainty Estimation: Predicting Classification Error in an Automated Machine Learning Setting. In *ICML Workshop on Automated Machine Learning*, pages 1–6, 2020.

[63] Matthias König, Holger H Hoos, and Jan N van Rijn. Speeding Up Neural Network Verification via Automated Algorithm Configuration. In *ICLR Workshop on Security and Safety in Machine Learning Systems*, pages 1–4, 2021.

[64] Matthias König, Holger H Hoos, and Jan N van Rijn. Accelerating Adversarially Robust Model Selection for Deep Neural Networks via Racing. In *Proceedings of the 38th AAAI Conference on Artificial Intelligence (AAAI-24)*, pages 21267–21275, 2024.

[65] Matthias König, Xiyue Zhang, Holger H Hoos, Marta Kwiatkowska, and Jan N van Rijn. Automated Design of Linear Bounding Functions for Sigmoidal Nonlinearities in Neural Networks. In *Proceedings of the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases*, 2024.

[66] Lars Kotthoff. Algorithm Selection for Combinatorial Search Problems: A Survey. In *Data Mining and Constraint Programming*, pages 149–190. Springer, 2016.

[67] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. Adversarial examples in the physical world. *arXiv preprint arXiv:1607.02533*, 2016.

[68] Ailsa H. Land and Alison G. Doig. An Automatic Method for Solving Discrete Programming Problems. In *50 Years of Integer Programming 1958-2008 - From the Early Years to the State-of-the-Art*, pages 105–132. Springer, 2010.

[69] Mathias Lecuyer, Vaggelis Atlidakis, Roxana Geambasu, Daniel Hsu, and Suman Jana. Certified Robustness to Adversarial Examples with Differential Privacy. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (SP2019)*, pages 656–672. IEEE, 2019.

[70] Kevin Leyton-Brown, Eugene Nudelman, Galen Andrew, Jim McFadden, and Yoav Shoham. A portfolio approach to algorithm selection. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI-03)*, pages 1542–1543, 2003.

[71] Linyi Li, Xiangyu Qi, Tao Xie, and Bo Li. Sok: Certified robustness for deep neural networks. *arXiv preprint arXiv:2009.04131*, 2020.

[72] Marius Lindauer, Holger H Hoos, Frank Hutter, and Torsten Schaub. AutoFolio: An Automatically Configured Algorithm Selector. *Journal of Artificial Intelligence Research*, 53:745–778, 2015.

[73] Changliu Liu, Tomer Arnon, Christopher Lazarus, Christopher A. Strong, Clark W. Barrett, and Mykel J. Kochenderfer. Algorithms for Verifying Deep Neural Networks. *Foundations and Trends in Optimization*, 4(3-4):244–404, 2021.

[74] Weiyang Liu, Yandong Wen, Zhiding Yu, and Meng Yang. Large-Margin Softmax Loss for Convolutional Neural Networks. In *Proceedings of the 33nd International Conference on Machine Learning (ICML 2016)*, pages 507–516, 2016.

[75] Alessio Lomuscio and Lalit Maganti. An approach to reachability analysis for feed-forward ReLU neural networks. *arXiv preprint arXiv:1706.07351*, 2017.

[76] Manuel Lopez-Ibanez and Thomas Stützle. Automatically improving the anytime behaviour of optimisation algorithms. *European Journal of Operational Research*, 235(3):569–582, 2014.

[77] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards Deep Learning Models Resistant to Adversarial Attacks. *arXiv preprint arXiv:1706.06083*, 2017.

[78] Yuri Malitsky, Ashish Sabharwal, Horst Samulowitz, and Meinolf Sellmann. Parallel SAT Solver Selection and Scheduling. In *Proceedings of the Eighteenth International Conference on Principles and Practice of Constraint Programming (CP2012)*, pages 1–15, 2012.

[79] Oded Maron and Andrew Moore. Hoeffding Races: Accelerating Model Selection Search for Classification and Function Approximation. In *Advances in Neural Information Processing Systems 6 (NeurIPS 1993)*, pages 59–66, 1993.

[80] Mark Huasong Meng, Guangdong Bai, Sin Gee Teo, Zhe Hou, Yan Xiao, Yun Lin, and Jin Song Dong. Adversarial robustness of deep neural networks: A survey from a formal verification perspective. *IEEE Transactions on Dependable and Secure Computing*, pages 1–19, 2022.

[81] Saeed Mian Qaisar. Baseline wander and power-line interference elimination of ecg signals using efficient signal-piloted filtering. *Healthcare Technology Letters*, 7(4):114–118, 2020.

[82] Jeet Mohapatra, Ching-Yun Ko, Lily Weng, Pin-Yu Chen, Sijia Liu, and Luca Daniel. Hidden Cost of Randomized Smoothing. In *Proceedings of the 24th International Conference on Artificial Intelligence and Statistics (AISTATS 2021)*, pages 4033–4041, 2021.

[83] Jeet Mohapatra, Tsui-Wei Weng, Pin-Yu Chen, Sijia Liu, and Luca Daniel. Towards Verifying Robustness of Neural Networks Against A Family of Semantic Perturbations. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2020)*, pages 241–249, 2020.

[84] Leonardo de Moura and Nikolaj Bjørner. Satisfiability Modulo Theories: An Appetizer. In *Proceedings of the Brazilian Symposium on Formal Methods (SBMF 2018)*, pages 23–36, 2009.

[85] Christoph Müller, François Serre, Gagandeep Singh, Markus Püschel, and Martin Vechev. Scaling Polyhedral Neural Network Verification on gpus. In *Proceedings of Machine Learning and Systems 3 (MLSys 2021)*, pages 1–14, 2021.

[86] Mark Niklas Müller, Gleb Makarchuk, Gagandeep Singh, Markus Püschel, and Martin Vechev. PRIMA: General and Precise Neural Network Certification via Scalable Convex Hull Approximations. In *Proceedings of the 49th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2022)*, pages 1–33, 2022.

[87] Mark Niklas Müller, Christopher Brix, Stanley Bak, Changliu Liu, and Taylor T. Johnson. The Third International Verification of Neural Networks Competition (VNN-COMP 2022): Summary and Results. *arXiv preprint arXiv:2212.10376*, 2023.

[88] Nina Narodytska, Shiva Prasad Kasiviswanathan, Leonid Ryzhyk, Mooly Sagiv, and Toby Walsh. Verifying Properties of Binarized Deep Neural Networks. In *Proceedings of the 32nd AAAI Conference on Artificial Intelligence (AAAI-18)*, pages 6615–6624, 2018.

[89] Nicolas Papernot, Patrick McDaniel, Xi Wu, Somesh Jha, and Ananthram Swami. Distillation as a Defense to Adversarial Perturbations Against Deep Neural Networks. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (IEEE S&P 2016)*, pages 582–597, 2016.

[90] Luca Pulina and A. Tacchella. Checking Safety of Neural Networks with SMT Solvers: A Comparative Evaluation. In *AI*IA*, pages 127–138, 2011.

[91] Luca Pulina and A. Tacchella. NeVer: A Tool for Artificial Neural Networks Verification. *Annals of Mathematics and Artificial Intelligence*, pages 403–425, 2011.

[92] Luca Pulina and Armando Tacchella. Challenging SMT Solvers to Verify Neural Networks. *AI Communications*, pages 117–135, 2012.

[93] Aditi Raghunathan, Jacob Steinhardt, and Percy Liang. Certified Defenses against Adversarial Examples. *arXiv preprint arXiv:1801.09344*, 2018.

[94] Karsten Scheibler, Leonore Winterer, Ralf Wimmer, and Bernd Becker. Towards Verification of Artificial Neural Networks. In *Proceedings of the 18th Workshop on Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV 2015)*, pages 30–40, 2015.

[95] David Shriver, Sebastian Elbaum, and Matthew B. Dwyer. DNNV: A Framework for Deep Neural Network Verification. In *Proceedings of the 33rd International Conference on Computer Aided Verification (CAV 2021)*, pages 137–150, 2021.

[96] Gagandeep Singh, Rupanshu Ganvir, Markus Püschel, and Martin Vechev. Beyond the Single Neuron Convex Barrier for Neural Network Certification. In *Advances in Neural Information Processing Systems 32 (NeurIPS 2019)*, pages 15072–15083, 2019.

[97] Gagandeep Singh and Timon Gehr. Boosting Robustness Certification of Neural networks. In *Proceedings of the 7th International Conference on Learning Representations (ICLR 2019)*, pages 1–12, 2019.

[98] Gagandeep Singh, Timon Gehr, Matthew Mirman, Markus Püschel, and Martin Vechev. Fast and Effective Robustness Certification. In *Advances in Neural Information Processing Systems 31 (NeurIPS 2018)*, pages 10825–10836, 2018.

[99] Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin Vechev. An Abstract Domain for Certifying Neural Networks. In *Proceedings of the 46th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2019)*, pages 1–30, 2019.

[100] Weidi Sun, Yuteng Lu, Xiyue Zhang, and Meng Sun. DeepGlobal: A framework for global robustness verification of feedforward neural networks. *Journal of Systems Architecture*, 128:102582, 2022.

[101] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus. Intriguing properties of neural networks. In *Proceedings of the 2nd International Conference on Learning Representations (ICLR 2014)*, pages 1–10, 2014.

[102] Merle W Tate and Sara M Brown. Note on the Cochran Q test. *Journal of the American Statistical Association*, 65(329):155–160, 1970.

[103] Chris Thornton, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Auto-WEKA: Combined Selection and Hyperparameter Optimization of Classification Algorithms. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD2013)*, pages 847–855, 2013.

[104] Vincent Tjeng, Kai Xiao, and Russ Tedrake. Evaluating Robustness of Neural Networks with Mixed Integer Programming. In *Proceedings of the 7th International Conference on Learning Representations (ICLR 2019)*, pages 1–21, 2019.

[105] Alan Turing. Intelligent machinery (1948). *B. Jack Copeland*, pages 395–432, 2004.

[106] European Union. Regulation (eu) 2024/1689 of the european parliament and of the council of 13 june 2024 laying down harmonised rules on artificial intelligence and amending certain union legislative acts (artificial intelligence act), 2024.

[107] Mauro Vallati, Chris Fawcett, Alfonso Emilio Gerevini, Holger Hoos, and Alessandro Saetti. Automatic Generation of Efficient Domain-Specific Planners from Generic Parametrized Planners. In *Proceedings of the 6th Annual Symposium on Combinatorial Search (SOCS)*, pages 184–192, 2013.

[108] Bruno Veloso, Luciano Caroprese, Matthias König, Sónia Teixeira, Giuseppe Manco, Holger H Hoos, and João Gama. Hyper-Parameter Optimization for Latent Spaces in Dynamic Recommender Systems. In *Proceedings of the European*

*Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases*, pages 249–264, 2021.

[109] Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. Efficient Formal Safety Analysis of Neural Networks. In *Advances in Neural Information Processing Systems 31 (NeurIPS 2018)*, pages 6369–6379, 2018.

[110] Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. Formal Security Analysis of Neural Networks using Symbolic Intervals. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security 18)*, pages 1599–1614, 2018.

[111] Shiqi Wang, Huan Zhang, Kaidi Xu, Xue Lin, Suman Jana, Cho-Jui Hsieh, and Zico Kolter. Beta-CROWN: Efficient Bound Propagation with Per-neuron Split Constraints for Neural Network Robustness Verification. In *Advances in Neural Information Processing Systems 34 (NeurIPS 2021)*, pages 29909–29921, 2021.

[112] Lily Weng, Huan Zhang, Hongge Chen, Zhao Song, Cho-Jui Hsieh, Luca Daniel, Duane Boning, and Inderjit Dhillon. Towards Fast Computation of Certified Robustness for ReLU Networks. In *Proceedings of the 35th International Conference on Machine Learning (ICML 2018)*, pages 5276–5285, 2018.

[113] Eric Wong and Zico Kolter. Provable Defenses against Adversarial Examples via the Convex Outer Adversarial Polytope. In *Proceedings of the 35th International Conference on Machine Learning (ICML 2018)*, pages 5286–5295, 2018.

[114] Haoze Wu, Aleksandar Zeljic, Guy Katz, and Clark W. Barrett. Efficient Neural Network Analysis with Sum-of-Infeasibilities. In Dana Fisman and Grigore Rosu, editors, *Proceedings of the 28th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2022)*, volume 13243, pages 143–163, 2022.

[115] Weiming Xiang, Hoang-Dung Tran, and Taylor T Johnson. Output Reachable Set Estimation and Verification for Multilayer Neural Networks. *IEEE Transactions on Neural Networks and Learning Systems*, 29(11):5777–5783, 2018.

[116] Lin Xu, Holger Hoos, and Kevin Leyton-Brown. Hydra: Automatically Configuring Algorithms for Portfolio–Based Selection. In *Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI–10)*, pages 210–216, 2010.

[117] Lin Xu, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. SATzilla: Portfolio-based Algorithm Selection for SAT. *Journal of Artificial Intelligence Research*, 32:565–606, 2008.

[118] Lin Xu, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Hydra-MIP: Automated Algorithm Configuration and Selection for Mixed Integer Programming. In *RCRA Workshop on Experimental evaluation of Algorithms for Solving Problems with Combinatorial Explosion*, pages 16–30, 2011.

[119] Huan Zhang, Tsui-Wei Weng, Pin-Yu Chen, Cho-Jui Hsieh, and Luca Daniel. Efficient Neural Network Robustness Certification with General Activation Functions. In *Advances in Neural Information Processing Systems 31 (NeurIPS 2018)*, volume 31, pages 4944–4953, 2018.

[120] Jingfeng Zhang, Xilie Xu, Bo Han, Gang Niu, Lizhen Cui, Masashi Sugiyama, and Mohan S. Kankanhalli. Attacks Which Do Not Kill Training Make Adversarial Learning Stronger. In *Proceedings of the 37th International Conference on Machine Learning (ICML 2020)*, pages 11278–11287, 2020.

# Summary

In an era marked by the widespread deployment of neural networks, the criticality of ensuring their reliability and safety cannot be overstated. This thesis delves into the sphere of neural network verification, a crucial yet challenging task aimed at proving the correctness of neural network models. Moreover, this thesis seeks to introduce novel techniques and strategies that could significantly improve the efficiency of neural network verification algorithms, thereby contributing to the development of more secure and reliable deep learning applications.

In Chapter 3, the thesis presents a detailed examination of the current landscape in neural network verification, specifically focusing on local robustness verification. The diversity within the field, both in terms of verification techniques and neural network architectures, presents a complex challenge for practitioners aiming to ensure the safety and reliability of these systems. We conduct a thorough empirical analysis of several prominent verification algorithms, revealing the fragmented nature of the state of the art. The findings suggest that there is no single dominant algorithm that excels across all types of verification instances. Instead, the performance of these algorithms is highly complementary, indicating the potential benefits of employing algorithm portfolios to enhance verification efficiency. This nuanced view highlights the importance of considering a range of methods and techniques to address the varied challenges posed by different neural network configurations and verification properties.

Following this, Chapter 4 delves into the realm of mixed integer programming (MIP)-based neural network verification, with a particular focus on speeding up the verification process through automated algorithm configuration. The chapter introduces novel approaches to harness algorithm portfolios, demonstrating how the performance of MIP-based verification systems can be significantly improved. By adopting automated configuration techniques, we present a systematic approach to tailor verification methods more closely to the specific characteristics of a given neural network, thereby reducing

computational costs and enhancing the efficiency of the verification task.

Chapter 5 introduces a novel perspective on verification by exploring the predictability of the running time of various verification algorithms for specific problem instances. This approach seeks to allocate computational resources more judiciously, focusing efforts on instances with a higher likelihood of being solvable within reasonable time budgets. The integration of running time predictions into the verification process represents a strategic shift towards more resource-aware methodologies, potentially transforming the efficiency and applicability of neural network verification in real-world scenarios.

In the final thematic chapter, Chapter 6, the research focuses on the task of robust model selection within the domain of adversarial robustness. We propose a sophisticated racing algorithm that leverages simple yet novel heuristics to efficiently determine the most robust neural network model from a given set. This method not only streamlines the model selection process but also significantly reduces the computational overhead associated with evaluating multiple candidate models.

Therefore, we have demonstrated how automated machine learning, or meta-algorithmic approaches in general, can improve the performance and practical efficiency of neural network verification systems, thereby contributing to a safer and more reliable usage of deep neural networks in the evolving landscape of artificial intelligence applications.

# Samenvatting

In een tijdperk dat gekenmerkt wordt door de wijdverspreide inzet van neurale netwerken, kan het belang van het waarborgen van hun betrouwbaarheid en veiligheid niet genoeg worden benadrukt. In dit proefschrift staat de verificatie van neurale netwerken centraal, een cruciale maar uitdagende taak gericht op het bewijzen van de robuustheid van neurale netwerkmodellen. Daarnaast introduceert deze dissertatie nieuwe technieken en strategieën die de efficiëntie van verificatiealgoritmen voor neurale netwerken aanzienlijk verbeteren en zo bijdragen aan de ontwikkeling van veiligere en betrouwbaardere toepassingen van deep learning.

Hoofdstuk 3 presenteert een gedetailleerd onderzoek van het huidige landschap in verificatie van neurale netwerken, specifiek gericht op de verificatie van lokale robuustheid. De diversiteit binnen het onderzoeksveld, zowel wat betreft verificatietechnieken als neurale netwerkarchitecturen, vormt een complexe uitdaging voor experts die de veiligheid en betrouwbaarheid van deze systemen willen garanderen. We voeren een grondige empirische analyse uit van een aantal prominente verificatiealgoritmen en brengen daarmee de gefragmenteerde aard van het onderzoeksveld aan het licht. De bevindingen suggereren dat er niet één algoritme domineert in alle verificatieproblemen. In plaats daarvan zijn de prestaties van verschillende algoritmen zeer complementair, wat wijst op de potentiële voordelen van het gebruik van algoritmeportfolio's om de verificatie-efficiëntie te verbeteren. Deze genuanceerde kijk benadrukt het belang van het overwegen van een reeks methoden en technieken om de gevarieerde uitdagingen van verschillende neurale netwerkconfiguraties en verificatie-eigenschappen aan te pakken.

Hoofdstuk 4 gaat vervolgens in op mixed integer programming (MIP)-gebaseerde verificatie van neurale netwerken, met speciale aandacht voor het versnellen van het verificatieproces door middel van geautomatiseerde algoritmeconfiguratie. Het hoofdstuk introduceert nieuwe benaderingen om algoritmeportfolio's in te zetten en laat zien hoe de prestaties van MIP-gebaseerde verificatiesystemen aanzienlijk kunnen

worden verbeterd. Door gebruik te maken van geautomatiseerde configuratietechnieken toont het hoofdstuk de mogelijkheid om verificatiemethoden beter af te stemmen op de specifieke kenmerken van een specifiek neurale netwerk, waardoor de berekenkosten worden verlaagd en de efficiëntie van de verificatietaak wordt verbeterd.

Hoofdstuk 5 introduceert een nieuw perspectief op verificatie door de voorspellende mogelijkheden te onderzoeken met betrekking tot de rekentijd van verschillende verificatiealgoritmen voor specifieke problemen. Deze benadering is erop gericht om rekenkracht adequater onder verschillende problemen toe te wijzen, door de inspanningen te richten op verificatieproblemen waarvan de kans groter is dat ze binnen redelijke tijdbudgetten kunnen worden opgelost. De integratie van voorspelling van de benodigde rekentijd in het verificatieproces vertegenwoordigt een strategische verschuiving naar methodologieën die bewust omgaan met reken middelen, die mogelijk de efficiëntie en toepasbaarheid van verificatie van neurale netwerken in echte scenario's kunnen veranderen.

In het laatste thematische hoofdstuk, hoofdstuk 6, spitst de discussie zich toe op de kritieke taak van robuuste modelselectie binnen het domein van adversariële robuustheid. We stellen een geavanceerd race-algoritme voor dat gebruik maakt van eenvoudige maar nieuwe heuristieken om efficiënt het meest robuuste neurale netwerkmodel te bepalen uit een gegeven set. Deze methode stroomlijnt niet alleen het modelselectieproces, maar vermindert ook aanzienlijk de computationele verspilling die gepaard gaat met het evalueren van meerdere kandidaatmodellen.

Concreet hebben we aangetoond hoe geautomatiseerde machine learning, of meta-algoritmische benaderingen in het algemeen, de prestaties en praktische efficiëntie van verificatiesystemen voor neurale netwerken kunnen verbeteren en daarmee bijdragen aan een veiliger en betrouwbaarder gebruik van diepe neurale netwerken in het ontwikkelende landschap van kunstmatige intelligentietoepassingen.

# Acknowledgements

This work would not exist without the support of many people. First and foremost, I would like to thank my family, especially my mother, who always encouraged me and gave me the confidence to embark on this exiting yet challenging journey. Furthermore, I am deeply grateful to my grandparents, who supported me in any way they could throughout my graduate studies.

I would like to thank my thesis supervisors Prof. Holger Hoos and Dr. Jan van Rijn for their continuous support and advice, and for giving me the opportunity to develop my own research project. Ever since I started working with with them as a master student, I have learned countless important lessons from which I will benefit for the rest of my professional life.

I am extremely thankful to have been a member of the wonderful ADA research group during the last few years. The great discussions as well as the social events, especially the evenings at De Bonte Koe and amazing group outings in Leiden and Aachen will always be remembered. In particular, I would like to thank Annelot Bosman for the joint work we did on research projects and for the fun we had while working together, even during frustrating periods. I also owe deep gratitude to Bram Renting who helped me navigate my way through our compute cluster, and for introducing me to many aspects of Dutch culture that would otherwise have remained undiscovered to me. Furthermore, I gratefully acknowledge the support of my colleagues, collaborators and students, who all had an impact on this thesis in one way or the other: Hermes Spaink, Serban Vadineanu, Charles Moussa, Amine Hadji, Hadar Shavit, Julia Wąsala, Laurens Arp, Mike Huisman, Jasmin Kareem, Corné Spek, Maria Kavvadia, Konstantin Kaulen, and Thijs Snelleman.

In addition, I thank Prof. Marta Kwiatkowska and Dr. Xiyue Zhang and all the members of the QAVAS group for their hospitality and support I received from them during my stay at the University of Oxford. The insightful discussions we had during

my research visit have greatly advanced my understanding of the methods discussed in this thesis. My special thanks goes to Christopher Weinhuber, Tobias Lorenz and Emanuele La Malfa for the fruitful discussions and good times we had at the office or the pub.

I feel deeply indebted towards my housemate Varol and part-time housemate Ceren for helping me push through challenging COVID times and making Amsterdam feel more like a home. Finally, I thank my best friends Daniel, Andi, Freddy, Thomy and Felix – our friendship has lasted over time and distance and I feel blessed to have you in my life.

Clearly, this list of acknowledgments is incomplete, and I would like to thank all the people who supported me and my work in so many different ways.

# Curriculum Vitae

Matthias König was born in Saarbrücken, Germany on the 7th of August, 1993, and grew up in Saarbrücken as well. He studied in Berlin, Leiden and Amsterdam and holds master's degrees in Information Studies from the University of Amsterdam and in Mediatechnology from Leiden University, which he both obtained in 2019. He started his Ph.D. in February 2020 on investigating trustworthiness and safety of AI systems in the distributed ADA (Automated Design of Algorithms) research group at Leiden University and RWTH Aachen University under the supervision of Dr. Jan van Rijn and Prof.dr. Holger Hoos. As part of his Ph.D., he visited the QAVAS research group led by Prof.dr. Marta Kwiatkowska at the University of Oxford for a period of 3 months. His research interests lie in AI safety, formal verification and automated machine learning. During his Ph.D. studies, he supervised several master students and took courses on scientific conduct, among others.