Universiteit Leiden

Master Computer Science

Auto-Verify: A framework for portfolio-based neural network verification

Name:	Corné Spek
Student ID:	s2337258
Date:	December 19, 2023
Specialisation:	Artificial Intelligence
Daily supervisor:	Matthias König
Daily supervisor:	Annelot W. Bosman
1st supervisor:	Jan N. van Rijn
2nd supervisor:	Holger H. Hoos

Master's Thesis in Computer Science

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

Deep neural networks have made great advancements in recent years, leading to them being deployed in a wide range of domains. Despite this, neural networks have been shown to be vulnerable to a class of attacks known as *adversarial attacks*, where an input image is slightly altered to produce arbitrary misclassifications. Such vulnerabilities could prevent neural networks from being used in safety-critical tasks, where these risks are unacceptable. To defend against these categories of attacks, methods that can mathematically guarantee a neural network's robustness against adversarial attacks have been proposed, although providing these guarantees is a computationally expensive task. In this thesis, we propose a novel framework called Auto-Verify that makes experimentation and portfolio construction with neural network verification tools more convenient. Auto-Verify provides interfaces for applying algorithm configuration techniques to four state-of-the-art neural network verification tools, as well as managing the environments and installation of these verification tools. Moreover, we improve the efficiency of neural network verification by leveraging parallel portfolios of different neural network verification tools. Thereby, we overcome several challenges typically associated with formal neural network verification, such as selecting a verification tool given one or more instances, and not having a uniform interface for each verification tool. Our experiments on MNIST, CIFAR and TLL Verify Bench datasets show that using parallel portfolios of neural network verification tools can lead to a speed-up in total time by a factor of up to 1.5, and reduce the number of timeouts by a factor of up to 1.12 compared to using a single verification tool. Lastly, we find that automatically configuring the verification tools in a parallel portfolio to achieve a meaningful improvement in performance remains challenging, due to long running times, heterogeneous instances, and error-prone verification tools.

Table of Contents

Chapter 1

Introduction

Recent advances in the field of deep learning have greatly improved the performance of deep neural networks in many domains such as computer vision, natural language processing, game-playing, and medical diagnoses to name just a few. Consequently, deep neural networks have been deployed on an increasingly wide range of domains, including safety-critical applications like autonomous driving, facial recognition, and cyber security. As neural networks grow in size and complexity, they effectively become "black-box" algorithms, meaning it becomes increasingly difficult to reason about their decisions and behaviours. This lack of "explainability" is troublesome when neural networks are deployed on a safety-critical task, where confident deployment and formal guarantees are a necessity.

By now it is widely known that neural networks are vulnerable to *adversarial attacks* [GSS15] (see Figure 1.1), where an input is altered to have the model make arbitrary (wrong) predictions. In the case of image classification, these changes to the image can be so small (as small as changing a single pixel [SVS19]) that the human eye cannot detect them anymore. If a model is susceptible to small changes in the input producing different classifications, we say that the model is not *locally robust*. Local robustness considers robustness w.r.t. a single input, while global robustness reasons about the input space as a whole [Sun+22]. In this thesis, we focus on local robustness, which aims to prove if a perturbation within a given radius of the original image exists that would lead to a misclassification.

Different methods have been proposed to defend against such adversarial attacks, including *empirical* defences and *formal* verification methods. While empirical defences generally scale to larger neural network sizes than formal verification, they only provide a limited perspective on the robustness of a neural network. Empirical defences can usually be broken again by sophisticated attackers, and can also not provide us with the full picture of robustness given that the input space is virtually infinite in cardinality. Formal verification, however, can give us rigorous mathematical guarantees on certain properties of input-output combinations, at the cost of increased computational complexity and, thus, even worse scalability.

Although significant progress has been made with regard to the scale and complexity of neural networks on which verification queries can be completed within a reasonable amount of resources, state-of-the-art methods scale up to (specifically trained) networks of around 10^5 neurons [Xu+20]. This still leaves many neural networks out of the question, as their architectures and sizes make formal verification infeasible.

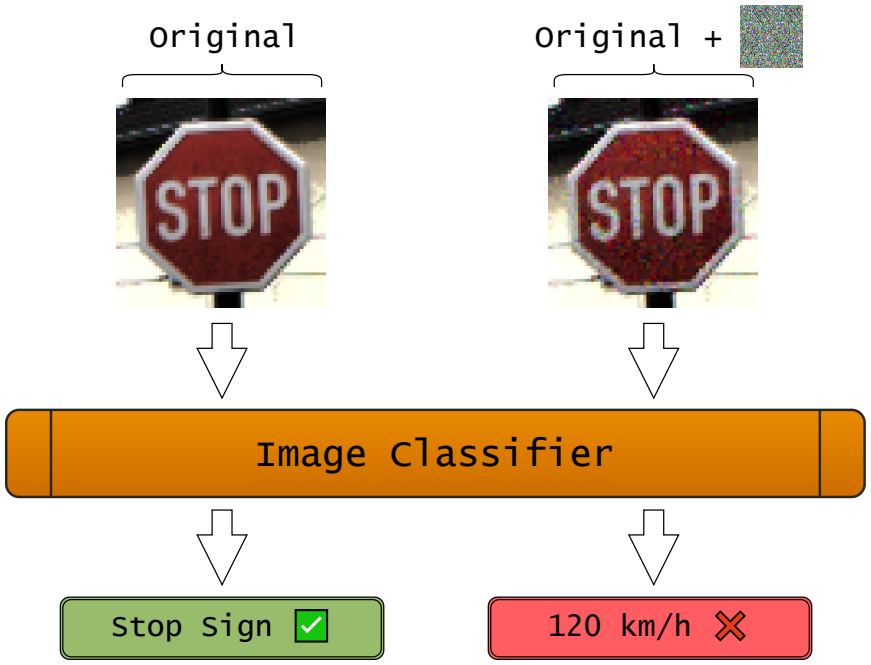

Figure 1.1: An example of an adversarial attack. The original image is correctly classified as a stop sign. By applying a specific perturbation to the original image, it is possible to make the image classification model generate an arbitrary prediction even though the image is still clearly a stop sign. Source: https://kennysong.github.io/adversarial.js/

Prior work by König et al. [KHR22] shows that by using algorithm configuration techniques to construct a portfolio of MIP-based verification systems, a significant decrease in CPU time on verification instances can be achieved. To be more specific, CPU times were reduced by factors ranging from 1.6 up to as high as 10.3. These results form the foundation that we build this work upon, in which we aim to extend and streamline the process of automatically configuring complete neural network verification systems.

Applying algorithm configuration techniques to tools that can prove properties on neural networks is by no means a trivial task, considering the high running times and heterogeneous problem instances. Furthermore, these tools often do not directly expose or document all their hyperparameters, do not perform similarly (or are not compatible) on the same problem instances, and do not have uniform interfaces. Additional work by König et al. [Kön+23] has also shown that there is not one verification tool that consistently dominates all others, making the use of parallel portfolios of different verification tools an interesting potential source for further improvement in performance.

We developed *Auto-Verify*,[1] a user-friendly software package aimed at automatically configuring and using parallel portfolios of neural network verification tools. Furthermore, Auto-Verify provides uniform interfaces to streamline interaction with neural network verification tools, hyperparameter spaces to sample configurations for verification tools, and additional utilities such as automated installation scripts and managing environments to run verification tools.

[1] https://github.com/ADA-research/auto-verify. Licensed under the BSD-3 license.

Our contributions are the following:

- We developed a novel framework for neural network verification with a focus on automated algorithm configuration and portfolio construction;
- we investigate the use of parallel portfolios of different (configured) neural network verification tools on benchmarks from the neural network verification literature based on prior work by König et al. [KHR22]; and
- we discuss our findings and challenges associated with neural network verification and applying algorithm configuration in this field.

Our experiments show that running parallel portfolios of unconfigured neural network verification tools, i.e., with default settings, leads to an improvement in performance up to a factor of 1.5 and a reduction in timeouts by a factor of up to 1.12, while configuring the hyperparameters of neural network verification tools did not yield a meaningful improvement within our budget and resource constraints.

The rest of this thesis is structured as follows: Chapter 2 dives into the required background information. Chapter 3 discusses related work, also highlighting the overlap algorithm configuration in neural network verification has with algorithm configuration in SAT solving. Chapter 4 takes an in-depth look at Auto-Verify, discussing examples, inner workings and design decisions. Chapters 5 through 7 outline the experiments we ran to test the performance of parallel portfolios created by Auto-Verify. Lastly, Chapter 8 summarises the findings and proposes future research directions.

Chapter 2

Background

We follow common equations for neural networks and robustness in this chapter, such as the equations described in "Algorithms for Verifying Deep Neural Networks" by Liu et al. [Liu+21], where the symbols outlined below have the following meaning:

- Scalars and scalar functions: lowercase italic letters (x)
- Vectors and vector functions: lowercase bold letters ($\boldsymbol{x}$)
- Matrices and matrix functions: uppercase bold letters ($\boldsymbol{X}$)
- Sets and set functions: uppercase calligraphic letters ($\mathcal{X}$)

2.1 Neural Networks

Neural networks are a class of machine learning algorithms, inspired by the workings of the human brain [Hin05], that are capable of learning complex patterns and relationships within data. Typically, a neural network consists of an input layer, one or more hidden layers and an output layer. Inside the layers are one or more neurons that perform computations on their inputs and produce outputs that can be passed on to other nodes via weighted edges. An example of a simple neural network with three layers can be seen in Figure 2.1.

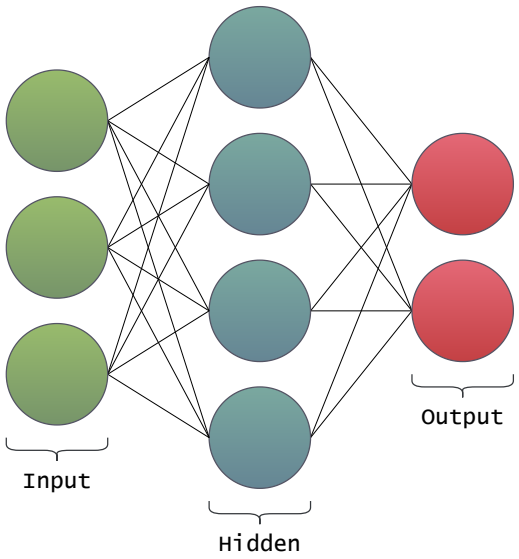

Figure 2.1: A simplified view of a small neural network with an input, hidden and output layer.

The strength of a connection between nodes is determined by weights and biases, which are hyperparameters that are learned during the training of the neural network to optimise the performance of the neural network. Nodes in the network receive input signals via these connections, on which they can then perform computations, which then become the output of that node and can subsequently become the input of connected nodes. Before the output is produced however, nodes will usually apply a non-linear activation function. These activation functions allow the neural network to model more complex patterns and relations by introducing non-linearity. There are many types of activation functions, and the choice of which to use and where in the neural network influences the performance of the neural network.

2.2 Robustness Verification

Because the computation obtained by training a neural network is discovered in an automated fashion [LBH15], it is difficult to interpret said computation. One would expect that a neural network trained for image classification on a large dataset will be robust to small changes made in the input, but that does not seem to always be true (see Figure 1.1). Adversarial examples can be created by applying slight perturbations to the input that maximise the model error while staying in proximity of the original input [Sze+14; Yua+19; Ues+18; ACW18]. This problem has been widely studied, resulting in the creation of different methods and approaches to evaluate the robustness of a neural network and protect against adversarial attacks. [TXT19; Bot+20; Bas+16; Bun+18; Dvi+18; Ehl17; Geh+18; HL20; HL21; Kat+17; Kat+19; Sch+15; Wan+18a; XTJ18; Bak21; Wan+21; Fer+22]

Traditionally, the *validation* of neural networks primarily involved feeding a large number of inputs into the network and checking if these yielded the intended outputs, see e.g. cross-validation [Bro00]. But this does not paint a full picture of the robustness of a given network, since the amount of possible inputs may be *very* large, too large to test all possible inputs. Different *empirical* defences have been proposed that aim to defend against adversarial examples. These defences use a variety of techniques, such as obfuscated gradients [XZZ20; Ma+18; Guo+18], or using ensembles of models to improve robustness [VS19; Pan+19; SRR20]. The disadvantage of these defences is that attacks can be created to bypass even the most sophisticated defences [Tra+20a; Ues+18; ACW18]. This cat-and-mouse game between attackers and defenders, plus the lack of certainty provided by empirical methods motivates the need for *formal* verification.

Formal neural network verification techniques can provide rigorous mathematical proof of whether certain input-output properties of a given network hold. As described by Liu et al. [Liu+21], properties can be formulated as statements, for example: if the inputs belong to some set $\mathcal{X}$, the outputs will belong to some set $\mathcal{Y}$. While formal verification can provide us with strong guarantees whether properties hold, verifying even a simple property has been proven to be an NP-complete problem [Kat+17].

Given a feed-forward neural network with n layers with a k_0-dimensional input, k_m-dimensional output, input $\boldsymbol{x} \in \mathcal{D}_{\boldsymbol{x}} \subseteq \mathbb{R}^{k_0}$ and output $\boldsymbol{y} \in \mathcal{D}_{\boldsymbol{y}} \subseteq \mathbb{R}^{k_m}$, where $\mathcal{D}_{\boldsymbol{x}}$ and $\mathcal{D}_{\boldsymbol{y}}$ are the domains of possible values for $\boldsymbol{x}$ and $\boldsymbol{y}$, respectively. This neural network can be used to represent a function $\boldsymbol{f}$, as $\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{x})$. To solve the verification problem, we have to check whether input-output relationships of $\boldsymbol{f}$ hold. In local robustness verification in image

classification networks, where we want to verify if input instances within a predefined radius to a certain input $\boldsymbol{x_0}$ belong to the same output class as $\boldsymbol{x_0}$, we can formulate the problem as follows:

$$\forall \boldsymbol{x} : \|\boldsymbol{x} - \boldsymbol{x}_0\|_p \leq \epsilon \implies \boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{f}(\boldsymbol{x}_0) \tag{2.1}$$

Where $\boldsymbol{x}$ is a sample in the neighbourhood of $\boldsymbol{x_0}$, denoted by an ℓ_p distance metric, such as ℓ_1 [Car+17; Che+18], ℓ_2 [Sze+14] or ℓ_∞ [GSS15; Pap+16]. The allowed radius of perturbations is denoted by ϵ. For a broader definition that extends to other problems besides classification problems, we refer to the work of Liu et al. [Liu+21].

Approaches to robustness verification can be characterised by soundness and completeness. A formal verification algorithm is one of the following two:

- *Sound*: Will only state if a property holds, if it actually holds.
- *Complete*: Will correctly state that a property holds, whenever it holds.

Sound verification algorithms are commonly referred to as incomplete verification, complete algorithms as complete verification. Verification algorithms can return one of three possible results to a verification query, namely:

- Satisfiable
- Unsatisfiable
- Unknown (sound algorithms only)

If a property is unsatisfiable, the property holds. For example, there does not exist an adversarial perturbation that will cause a misclassification. If it is satisfiable, the property is violated, e.g. there exists an adversarial perturbation that causes a misclassification. The case of "unknown" is only produced by sound algorithms, since complete algorithms are guaranteed to state if a property holds given enough time and resources. While complete verification is more desirable, incomplete methods can scale to larger networks than complete methods. Due to the long running times of verification queries, a fourth possible result to a verification query is usually introduced: a *timeout*, which is a predetermined amount of time for which the query is allowed to run after which the process is forcibly stopped. Note that when a timeout is returned, we also do not know the outcome of the verification query.

2.3 Methods of Robustness Verification

In this work, we limit ourselves to only considering complete verification. We leave similar studies on incomplete methods to future work.

The verification problem can be viewed from different angles and thus solved by different methods and approaches, of which we will discuss the most prominent methods. Most available complete verification methods do not support every type of neural network architecture and activation function, with most only supporting the ReLU activation function [Kön+23]. Note that it is possible to simplify networks to allow more neural network verification methods to be used, by applying transformations that preserve semantics on the operation graph of the network [SED21].

2.3.1 Solver-based Methods

Feed-forward ReLU networks can be encoded as a conjunction of linear inequalities [LXL23], which makes it possible to use SMT solvers to find a solution to the verification query. Similarly, the problem can also be encoded as a mixed-integer program (MIP) — opening the door to using highly optimised MIP solvers. Early work by Pulina et al. [PT10; PT11; PT12] on using SMT solvers for complete verification was able to achieve state-of-the-art results at the time, but verification of networks of realistic size within a reasonable time limit remained an open problem. Katz et al. introduced Reluplex [Kat+17] and Marabou [Kat+19], which both make use of the Simplex algorithm [Dan02] to scale verification to larger networks. The use of SMT solvers is generally not scalable to large networks [PT12].

Building on Equation 2.1, the equation can be formulated as a minimisation problem. Let $\boldsymbol{\lambda}(\boldsymbol{x})$ be the true class label for any $\boldsymbol{x}$, and $\mathcal{X}_{\text{valid}} \subseteq \mathcal{D}_{\boldsymbol{x}}$ be the domain of valid inputs, and p a distance metric such as ℓ_1, ℓ_2, or ℓ_∞. If Equation 2.2 can be solved, it will yield an adversarial example.

$$\begin{aligned} \min_{\boldsymbol{x}} \quad & \|\boldsymbol{x} - \boldsymbol{x_0}\|_p \\ \text{s.t.} \quad & \text{argmax}_i(\boldsymbol{f}_i(\boldsymbol{x})) \neq \boldsymbol{\lambda}(\boldsymbol{x_0}) \\ & \|\boldsymbol{x} - \boldsymbol{x_0}\|_p \leq \epsilon \\ & \boldsymbol{x} \in \mathcal{X}_{\text{valid}} \end{aligned} \tag{2.2}$$

In addition to making use of existing highly optimised commercial MIP-solvers, additional techniques such as tighter formulations and presolving can be applied to further reduce verification time [TXT19].

Cheng et al. [CNR17] and Lomuscio et al. [LM17] were among the first to show the potential of MIP solvers in complete verification. Subsequent work [Dut+18; TXT19; Bot+20; FJ18] successfully used MIP solvers to achieve new state-of-the-art results, being able to verify both larger and more complex networks within reasonable time limits. Networks trained in standard ways or of larger size however still remain problematic, with verification even timing out with generous time limits [KHR22].

2.3.2 Branch-and-Bound Methods

Branch-and-Bound (BaB) [LD60] is a technique used for solving hard optimisation problems where exhausting the entire search space would be intractable. In a nutshell, branch-and-bound works by systematically splitting (branching) the original minimisation problem into smaller subproblems, which form a rooted tree structure. During the procedure, the global upper bound represents the current candidate for the global minimum. When the lower bound of a subproblem becomes greater than or equal to the current global upper bound, this branch is pruned (bounding). These steps of branching and bounding reduce the size of the search space, which leads to efficient minimisation, since regions that cannot lead to better solutions are not explored.

Equation 2.2 is one such minimisation problem that can be solved efficiently by using the branch-and-bound paradigm. In the case of satisfiability problems, the global upper bound can be initialised to 0. In the context of neural network verification, that means that any branch with a lower bound greater than 0 cannot contain a counter-example, and thus be pruned. The nature of the branch-and-bound algorithm leads to various design choices

in the algorithm, such as the search strategy, branching rule and bounding method. The lower bounds of the subproblems created during the branch-and-bound procedure can be obtained using incomplete verification algorithms such as [Xu+21; Pal+21a; Pal+21b; Ehl17; WK18; Sin+19]. Upper bounds are found using falsification algorithms, such as [Don+18; Ehl17; Bun+20; Xu+21]. Different branching strategies include [Zha+18; Bun+18; De +21].

Compared to solver-based verification methods, neural network verification tools based on the branch-and-bound paradigm are able to verify bigger neural networks in reasonable time limits than solver-based approaches. In the 2022 edition of VNNCOMP [Mül+22] (see Section 2.6), an annual neural network verification competition, the top three competitors all used branch-and-bound techniques in their implementations.

2.3.3 Set Representations

The set of valid inputs $\mathcal{X}_{\text{valid}}$ for the network can be represented in different ways, which can lead to more efficient verification. Different representations of these sets have their own characteristics that may be able to solve a problem more efficiently in some cases compared to other set representations. Set representations that have been studied include *star-sets* [Bak21], *zonotopes* [Geh+18; Sin+18; Bak21], *polyhedra* [Sin+19; Zha+18] and *(error-based) symbolic interval propagation* [Wan+18a; Wan+18b; HL20; Bot+20; KBS22].

2.4 Automated Algorithm Configuration

2.4.1 Algorithm Configuration

Achieving peak performance of any given algorithm is important for many different practical or economical reasons. To obtain this peak performance, algorithms often require that their hyperparameters are configured to suit the specific problem scenario at hand. The process of *algorithm configuration* (AC) (or *hyperparameter tuning*) however, can be a very complex and undesirable task to do by hand for a number of reasons:

- Requires extensive domain knowledge;
- potentially large number of hyperparameters to configure;
- irreproducible, tedious and error-prone process.

Because of these problems, many different hands-off, automated approaches for algorithm configuration have been proposed.

In the problem of algorithm configuration, we have four distinct elements: a target algorithm $\mathcal{A}$, its hyperparameter search space Θ, a set of problem instances Π and a cost metric that measures the performance of a hyperparameter configuration on an instance: $c : \Theta \times \Pi \to \mathbb{R}$. We want to find a hyperparameter configuration $\theta^* \in \Theta$ that minimises the cost metric across all instances in Π, see Equation 2.3

$$\theta^* \in \arg\min_{\theta \in \Theta} \sum_{\pi \in \Pi} c(\theta, \pi) \tag{2.3}$$

In general, algorithm configuration methods are first given a set of training instances in the offline phase. These instances are used by the algorithm configuration method to produce a hyperparameter configuration, see Figure 2.2. The new configuration can then be used in

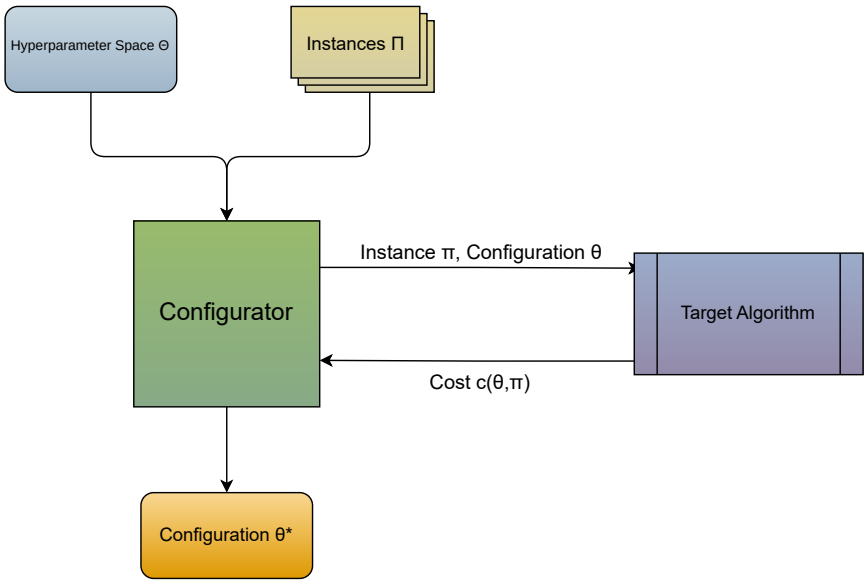

Figure 2.2: Schematic illustration of the Algorithm Configuration procedure. The configurator takes as input a hyperparameter space Θ and a set of instances Π. By observing the cost of the target algorithm using different configurations, the configurator tries to find the configuration θ^* that has the best performance.

the online phase, where the expectation is that the performance during the offline phase will generalise to previously unseen instances.

2.4.2 Methods for Algorithm Configuration

By performing random or grid searches over the hyperparameter search space Θ, the process of finding the best-performing hyperparameter configuration can be naively automated. However, conducting the search for hyperparameter configurations in this manner leads to an issue: a lot of time is spent evaluating hyperparameter configurations that are not promising, leading to long running times and wasted resources. In practice the search space of hyperparameters can have dozens of hyperparameters that impact performance, making it increasingly difficult to justify the use of random and grid search methods for such search spaces [YS20]. Hyperparameters can also affect each other or impose constraints on each other, which also plays a role. For example, hyperparameter x may only be a valid hyperparameter in the search space if Boolean hyperparameter y is set to `true`. Additionally, constraints can be imposed on the search space beforehand if it is known that some hyperparameter configurations are nonsensical or lead to undefined behaviour in the target algorithm.

The complex nature of the problem has led to the development of more sophisticated methods that are able to more efficiently explore the hyperparameter search space. Such methods can be broadly categorised into two categories [Sch+22]: model-free and model-based methods. As the names suggest, the difference between the two is in whether they make use of a learned model to gain insight into the performance of a hyperparameter configuration.

In this work we use SMAC [HHL11; Lin+22], a model-based state-of-the-art framework

for determining well-performing hyperparameter configurations. SMAC is a *sequential model-based optimisation* (SMBO) approach, that defines a surrogate model $\hat{c}(\theta, \pi)$ whose purpose is to approximate the cost metric c (see Eq. 2.3) as close as possible. By making use of an *acquisition function* that picks a set of configurations to be evaluated by considering the current approximation provided by $\hat{c}$, the resulting data from said evaluations can be used to iteratively train the model and improve its approximation of the target function. In the end, the knowledge gained about the performance of different configurations can be used to provide a hyperparameter configuration that yields the best performance on the problem instances. SMAC makes efficient use of random forests [Bre01] as the surrogate model and a core Bayesian optimisation [SLA12] loop combined with aggressive racing [Hut+09]. It has been shown SMAC can achieve state-of-the-art results on large hyperparameter spaces in various domains, such as determining the feasibility of radio spectrum repackings [NFL18].

2.4.3 Portfolio Construction

While automated algorithm configuration methods alone can produce very well-performing configurations, there is a problem: when only one configuration is given by the configurator, this configuration may not perform well on every instance in the instance set. This occurs when the instance set is not *homogeneous*, meaning different instances have different "characteristics". A configuration may perform well on one type of instance, but poorly on another. If the configurator can only produce one single configuration, it may be impossible to capture the heterogeneity of the instance set. To illustrate this, consider the following hypothetical scenario:

- We have distinct instances A and B in our uniformly distributed instance set.
- The hyperparameter space for the target algorithm consists of two Boolean hyperparameters: x and y.
- If x is `true`: A is solved efficiently, B is not
- If y is `true`: B is solved efficiently, A is not
- x and y cannot both be `true` at the same time

An algorithm configuration procedure that returns one configuration could, in this case, not find a configuration that performs well on the entire instance set and is thus forced to choose which instance to optimise over. While this is a trivial example meant to illustrate the problem, much more nuanced cases arise in real problems where similar situations of one configuration not being able to cover all instances can happen.

To solve this issue, we can make use of methods that construct *portfolios* of configurations. A portfolio of configurations is a collection of different configurations for one or more target algorithms. The key idea of portfolios is that the different configurations and algorithms in the portfolio each have distinct strengths that complement each other in order to achieve better overall performance. Looking at the previously given hypothetical example, we could create a portfolio $P := \{\theta_1, \theta_2\}$, where θ_1 is a configuration with x set to `true` and y set to `false` and θ_2 the inverse. Because crafting such portfolios by hand leads to the same problems as manual hyperparameter tuning, approaches for automated portfolio construction have been proposed.

There are basically two ways in which a portfolio can be used: running the algorithms in the portfolio in parallel or selecting an algorithm from the portfolio to run per instance.

This work does not consider the latter, we are mainly interested in running the algorithms in parallel. Parallel portfolios work by distributing the available resources to all algorithms, which are run in parallel until one of the algorithms finds a solution to the verification query. As previously stated, the outcome of a verification query should either be *satisfiable* or *unsatisfiable*. Once one of the algorithms returns one of these answers, all other running algorithms can be stopped as the answer to the verification query has been found. This can lead to both a decrease or an increase in CPU time, as multiple algorithms are being run in parallel.

A well-known approach for portfolio construction is Hydra [XHL10], which has been proven to work in the context of MIP-based neural network verification [KHR22]. Hydra is a greedy algorithm, whose main idea is scoring a configuration with its real score if it performs better than the portfolio on the current instance, but with the cost of the portfolio if it performs worse. Potential configurations are thus scored only by how much they improve the current portfolio leading the configurator to optimise for instances on which the current portfolio does not yet perform well.

Hydra systematically constructs portfolios as follows: Let P_i be the portfolio after iteration i, with $P_0 := \{\}$ being the empty portfolio. At iteration 1, we run a configurator to obtain the first configuration θ_1 which is added the the portfolio, resulting in $P_1 := \{\theta_1\}$. After the first iteration, the performance metric is updated as follows: if the incumbent configuration performs worse than the portfolio on an instance, it is instead scored with the performance of the portfolio. After each iteration, the performance of the new configuration(s) is evaluated and the portfolio is updated according to a portfolio updating strategy, resulting in a new portfolio $P_i = P_{i-1} \cup \{\theta_1, \ldots, \theta_n\}$.

2.5 Neural Network Verification Standards

To verify properties on a neural network we need two things: a network and a property. The lack of standard formats for networks and properties has been an issue in neural network verification in the past, but efforts to establish standard formats for networks and properties have been made recently by initiatives such as VNNCOMP [Mül+22].

2.5.1 ONNX

For networks, the ONNX [BLZ+23] format has been proposed as the default format. ONNX aims to create an open format for machine learning models, such as neural networks, but also more traditional machine learning models. Definitions for creating an extensible computational graph are given, whose operators can be used across different frameworks. This flexibility and active ecosystem that supports conversion for many common formats makes it a suitable option to standardise the network formats in neural network verification.

2.5.2 VNN-LIB

The format proposed for properties is VNN-LIB [Gui+23], which builds upon the ONNX and SMT-LIB [BFT16] formats. VNN-LIB provides support for the most common operations used in ONNX networks and the common networks in neural network verification literature. It is heavily inspired by the SMT-LIB format, an established language for specifying Satisfiability Modulo Theories (SMT) problems. A property in VNN-LIB consists

of a pre-condition that encodes the bounds of an input space and a post-condition that defines the "safe-zone" in which the outputs should land. VNN-LIB is a good candidate to standardise input formats, due to its interoperability with ONNX and making use of the well-known SMT-LIB format.

2.6 VNNCOMP

VNNCOMP [Mül+22] is an annual neural network verification competition which aims to bring together tools and research concerning neural network verification as well as benchmark the existing state-of-the-art. The competition started in 2019 and has successfully managed to bring in competitors, push standard formats, and establish a variety of interesting benchmarks to be studied. Varied benchmarks have been proposed by different authors to be studied in VNNCOMP, ranging from easy to very hard benchmarks that represent different domains in which neural network verification can be applied. This collection of benchmarks, as well as the reproducibility of the results on said benchmarks, makes them an interesting choice to benchmark our own methods on and compare the results.

Chapter 3

Related Work

3.1 Algorithm Configuration in SAT Solving

An area of research that faces a lot of the same challenges as neural network verification is that of satisfiability solving (SAT solving). Both are $\mathcal{NP}$-complete problems that try to state if a formula (or property) is satisfiable, which is why many of the algorithm techniques applied to SAT solving make sense to use for neural network verification.

One of the ways that make solving $\mathcal{NP}$-complete problems such as SAT feasible is the use of heuristics. Over the last decades (the first SAT-solving algorithms date back to the 1960s), many different heuristics have been developed and improved upon, making it feasible to solve larger and larger instances. An important insight here is that some heuristics work well on some types of instances, while other heuristics work better on other instances. This is why finding a good configuration for a SAT solver is important; a heuristic that does not suit the given problem instances may perform very poorly. Methods for algorithm configuration such as Hydra (Section 2.4.3) were initially developed with the intent of improving SAT-solving performance. Previously discussed approaches to tackling neural network verification such as MIP (Section 2.3.1) and BaB (Section 2.3.2) have been extensively researched in the context of SAT solving, yielding many high-performance solvers, insights, and ideas that can be leveraged for improving neural network verification.

3.2 Automated Algorithm Configuration in NNV

Work on automated algorithm configuration in the context of neural network verification is sparse. While most neural network verification tools come with a number of performance-relevant hyperparameters, these hyperparameters are usually not the focus of the study. Consequently, the configurations used in most studies are defaults or produced by hand.

König et al. [KHR22] showed the effectiveness of automated algorithm configuration and portfolio construction for MIP-based neural network verification tools. While their work shows very promising results, there is still room for improvement. The study only considered MIP-based verification tools, which are by now no longer state-of-the-art. Moreover, only the hyperparameters of the embedded MIP solver were configured. Tools themselves can also have hyperparameters that are relevant to performance. The portfolios constructed consisted of one verification tool with different configurations, but portfolios can contain more than one verification tool which could lead to an improvement in performance as verification tools have different characteristics. Further work by König et al. [Kön+23]

on CPU-based neural network verification revealed that verification tools exhibit a strong complementarity, further strengthening the case for the use of mixed algorithm portfolios.

3.3 Neural Network Verification Systems

DNNV [SED21] is a framework aimed at reducing the burden on users, developers and researchers doing neural network verification. By standardising in- and output formats, simplifying neural network operations and providing further utilities for convenience, DNNV is closely related to the tool we propose in this work. However, there are some key differences:

- Hyperparameters cannot be configured through DNNV.
- Recent state-of-the-art tools are not available, such as those using GPU resources.
- At the time of writing, DNNV is infrequently updated and/or maintained.

Goose [Sco+22] is a meta-solver that combines three techniques: algorithm selection, probabilistic satisfiability inference and time iterative deepening. By combining different verification tools with the aforementioned techniques, Goose is able to achieve speedups in verification time. At the time of writing, however, there is no public implementation of Goose available. Furthermore, Goose does not tune the hyperparameters of the verification tools that are used.

Chapter 4

Auto-Verify

4.1 Overview

As neural network verification in isolation is already a very challenging problem, it is our vision that it should not be made even more difficult by having to deal with numerous technical, often time-consuming challenges. Some of the most notable which we aim to address are:

- **No support for algorithm configuration**
  Most verification tools expose a number of hyperparameters that impact performance. However, determining the right hyperparameters to use for a particular set of instances is very hard since in-depth expert knowledge of said hyperparameters is required to make the proper changes. Since verification tools can have many dozens of hyperparameters to configure, exhaustively traversing the search space is infeasible. Selecting the right hyperparameter configuration for a scenario is important, as this can have a big impact on performance.
- **Selecting the right verification tool**
  As shown by König et al. [Kön+23], there is not a single verification tool that performs the best across all problem instances, indicating that selecting the right verification tool is important. Given arbitrary problem instances, it is not clear how to select this optimal verification tool without doing manual experimentation or having expert knowledge. While DNNV [SED21] has addressed this issue, it does not provide the level of control we need for using algorithm configuration methods, and also differs from Auto-Verify in other ways (see Section 3.3).
- **No uniform interface**
  Verification tools each have their own interfaces to run the verification procedure. If one wants to compare performance between one or more verification tools, this forces them to write scripts to conform their experiments to each individual interface per verification tool. Additionally, the verification tools may also not support the same input and output formats.
- **No support for (parallel) portfolios**
  There are no trivial ways to construct and run portfolios of (different) verification tools, which can potentially lead to faster verification of properties.
- **Varying installation procedures**
  Different verification tools have different requirements and installation methods, again making installing verification tools a time-consuming process.

To address these problems, we have created a user-friendly framework for neural network verification, which includes the following main features:

- Python API to interface with various verification tools.
- Support for applying algorithm configuration and portfolio construction methods.
- Command line interface to install verification tools and manage environments for said tools.
- Extensible to include new and/or custom verification tools.
- Open-source, documented, maintained and easily installable.

Auto-Verify is intended as an extensible framework for neural network verification which provides support for algorithm configuration and parallel portfolios, while tackling many of the difficulties associated with neural network verification.

To assure the quality of Auto-Verify, we have adhered to high software engineering standards. This includes strongly typed Python code, docstrings explaining the use of functions and classes, and general use of best practices in software engineering. Furthermore, all relevant parts of the code are extensively tested. Adding to this, we have also created a continuous integration pipeline that executes both unit and integration tests each time code is pushed to an important branch of the repository. The integration tests make sure that Auto-Verify is properly working by creating a new Docker container[1] and installing Auto-Verify and the verification tools supported by Auto-Verify while checking if all procedures finish without any errors.

In this chapter, we will first show and discuss the Auto-Verify Python API for verifying properties on neural networks. Afterwards, we will show its capability to perform algorithm configuration and parallel portfolio methods for neural network verification.

Note: Code examples shown in this chapter have various parts simplified or cut out for the sake of clarity. Working examples can be found in the documentation.[2]

4.2 Verification

After using Auto-Verify to install a verification tool, Listing 1 shows how the Auto-Verify API can be used to verify a local robustness property on a network using a specific verification tool.
After supplying the network, property and verifier, we can get one of 4 possible outputs to the verification process: *SAT*, *UNSAT*, *TIMEOUT* or *ERR*. While we stated in Section 2.2 that there are 3 possible outcomes in complete verification, in practice, programs can crash during the verification of a property, which is given its own error result state. Additional information that is reported are counter-examples (in VNNCOMP format[3]) in case of a violated property and if supported by the verification tool, time used to verify the property, and the output of the verification tool.

[1] https://www.docker.com/resources/what-container/

[2] https://ada-research.github.io/auto-verify/

[3] https://docs.google.com/document/d/1wdzF_WVME4XFqlg_ReCLavxwT5eX7h1OGPV16O21BGc/edit

```
$ pip install auto-verify
$ auto-verify install nnenum
```

```
from autoverify.verifier import Nnenum

network = "my_network.onnx"
prop = "my_propery.vnnlib"
verifier = Nnenum()

result = verifier.verify_property(network, prop)
print(result)  # => SAT | UNSAT | TIMEOUT | ERR
```

Listing 1: Example of installing Auto-Verify, installing a verification tool (nnenum [Bak21] in this case), and using nnenum to verify a property on a network.

4.2.1 Interface

All the available verification tools in Auto-Verify are classes that inherit from the same abstract class which provides a public interface, see Figure 4.1. In general, it is assumed that any verification tool implemented works via a command line interface (CLI). Since a verification tool may be implemented in any arbitrary way, a CLI is a flexible and generalising option. Internally, the flow of the verification process then goes as follows:

1. A verification problem instance, consisting of a network, property and timeout, is given by the user.
2. If a configuration was also provided, this configuration is initialised by the child class. This is necessary because verification tools often use their own mechanism for managing configurations, so conversion between formats might need to happen.
3. Based on the verification instance and configuration, the CLI command is constructed by the child class.
4. The base class enters the environment and contexts required to run the verification tool and executes the CLI command, recording and returning the observed results.

The data returned after verification includes the verification result, seconds taken, counter example (if *SAT*) and the tool output (stdout and stderr). To support using portfolios of verification tools, it is possible to specify the number of resources a verifier is allowed to use, specifically the number of CPU cores and the number of GPUs. This is required to be able to distribute the resources of a machine among verification tools and allow them to be used in parallel.

4.3 Algorithm Configuration

To apply algorithm configuration techniques and configure the hyperparameters of verification tools, we need a verification tool with its space of valid configurations, a set of instances to train on and a configurator, see Listing 2.

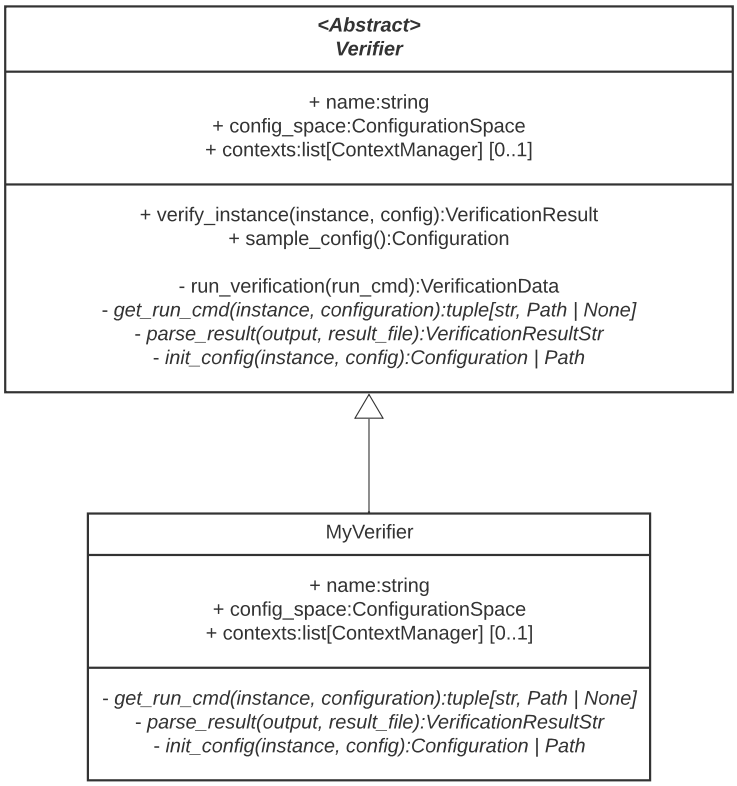

Figure 4.1: UML diagram of the abstract "Verifier" class and a child verifier class that inherits from it. Public methods are denoted with a "+", and private methods with a "-". All of the available verification tools in Auto-Verify inherit from this class, which allows for the creation of new verifier classes that are compatible with Auto-Verify.

```python
from autoverify.verifier import AbCrown
from autoverify.tune import smac_tune_verifier
from autoverify.instances import read_vnncomp_instances

verifier = AbCrown()
instances = read_vnncomp_instances("mnist_fc")
time_limit = 60 * 60 * 8  # 8 hours

config = smac_tune_verifier(verifier, instances, time_limit)
print(config)
```

Listing 2: Code example of configuring the hyperparameters of a verifier (α,β-CROWN in this case) on a set of instances for 8 hours, using SMAC as the configurator. The process returns the best configuration SMAC was able to find, which can then be used (passed as an argument) to verify properties as shown in Listing 1.

Table 4.1: Number of hyperparameters, conditions and forbiddens used in Auto-Verify for each of the 4 included neural network verification tools.

	α,β-**CROWN**	**nnenum**	**VeriNet**	**Oval-BaB**
Hyperparameters	14	24	25	40
Conditions	6	2	0	0
Forbiddens	2	1	0	0

Since our work builds upon the study by König et al. [KHR22] (which uses SMAC as the configurator), we have created hyperparameter spaces that are compatible with SMAC [HHL11; Lin+22], which uses the ConfigurationSpace [Lin+19] package for representing hyperparameter spaces. For each of the verifiers that are included by default in Auto-Verify we made hyperparameter spaces to the best of our abilities, see Tables 4.1, A.1, A.2, A.3 and A.4. As visible from Table 4.1, ConfigurationSpaces consist of three main elements:

- Hyperparameters
- Conditions
- Forbiddens

hyperparameters are, as the name implies, named values that can have an impact on performance. Their values can either be categorical (finite number of choices) or be sampled from a numerical interval of valid values. An example of a categorical hyperparameter could be Nnenum's branching strategy, which has four valid values: *Split Largest* (0), *Split One Norm* (1), *Split Smallest* (2) and *Split Inorder* (3). An example of a numerical hyperparameter range is α,β-CROWN's *BaB-Cut-Learning Rate*, which can take any value inside the interval $[0, 1]$.

Conditions and *forbiddens* can impose additional constraints on the hyperparameter space. A *condition* determines whether a hyperparameter is active or not, for example: α,β-CROWN's *PGD-Attack Mode* hyperparameter is only valid if *PGD-Attack Order* is not set to `Skip`. *Forbiddens* are used to make combinations of hyperparameters invalid. For example, Nnenum's *Split Order* hyperparameter cannot be in $\{0, 1, 2\}$ if the *Eager Bounds* hyperparameter is set to `False`.

While the hyperparameters, conditions and forbiddens make it possible to express a large part of the hyperparameter space, one of the design choices behind ConfigurationSpace is that each hyperparameter space must be serialisable. This means that it becomes impossible to express certain constraints within our hyperparameter space, leading to the space becoming larger than it should be. Another limitation of the hyperparameter spaces we made is that they are not as accurate as they could be due to the valid space of hyperparameters not being strongly defined or documented. It is infeasible to analyse the entire codebase of a verification tool to find all possible conflicts, ranges, values and quirks of hyperparameters. This can lead to the space of possible hyperparameter configurations containing invalid configurations that crash the verification tool. These challenges further highlight the difficulties that come with adopting algorithm configuration techniques in the context of neural network verification.

Although the representation of the hyperparameter spaces might not be perfect, SMAC can negate this issue to some extent. If the process is started with an illegal configuration it should crash relatively quickly. SMAC treats crashed runs differently by setting their cost to infinity, which should steer the configuration process away from further exploring that part of the hyperparameter search space.

4.4 Portfolios

Portfolios in Auto-Verify are separated into two parts: automatically creating portfolios and running portfolios in parallel.

4.4.1 Automatic Portfolio Construction

Auto-verify provides built-in support for constructing portfolios of verification tools using the Hydra algorithm (see Section 2.4.3). Hydra should take as input a number of available verification tools, the instances the portfolio should be constructed for and output a portfolio of one or more (configured) verification tools. An example of how this can be done in Auto-Verify is shown in Listing 3. The `PortfolioScenario` object takes more arguments that further specify how Hydra should work, for a full list of these options we refer to the Auto-Verify documentation.
In each iteration of the Hydra algorithm, two important things need to happen. First, a verification tool must be selected and secondly, this verifier should be configured further on the instances. There are two approaches to this:

- Make one big ConfigurationSpace that includes the verification tool and its hyperparameters. Meaning there is a parent hyperparameter that determines the verifier that will be used.
- Split the Hydra iteration into two isolated parts: selecting a verification tool and configuring its hyperparameters.

Because the individual range of valid configurations for the verification tools is already quite large, we opted to pick the second approach: splitting each Hydra iteration into a selection and configuration part. A hyperparameter that determines the time spent on each of these two parts, $\alpha \in [0, 1]$, is exposed. The fraction of time spent on selecting a verifier is given by $1 - \alpha$, and the fraction of time spent configuring is given by α. During the selection

```python
from autoverify.portfolio import Portfolio, PortfolioScenario, Hydra
from autoverify.instances import read_vnncomp_instances

instances = read_vnncomp_instances("mnist_fc")
pf_scenario = PortfolioScenario(
    ["nnenum", "abcrown", "ovalbab", "verinet"],
    instances,
)

hydra = Hydra(pf_scenario)
portfolio = hydra.tune_portfolio()

portfolio.to_json("example_pf.json")
```

Listing 3: Example of how to automatically construct a portfolio of verification tools in Auto-Verify, using the Hydra algorithm, on the `mnist_fc` dataset with four possible verification tools. The resulting portfolio is serialisable and can be exported to different formats.

stage, SMAC is given a ConfigurationSpace containing only the verification tools passed as input in the PortfolioScenario (see Listing 3). SMAC is then used to configure this single hyperparameter, making it select one of them. After one has been selected, the hyperparameters of the verification tool are then configured using SMAC. We assume here that the performance of the default configuration of a verification tool provides an indication of its performance after configuring the hyperparameters. If the best-performing configuration can only be found after first selecting a verification tool whose default configuration does not perform well, this approach will not be able to find that configuration.

Another consideration when setting up the scenario for portfolio construction is the allocation of available resources. The current default strategy used by Auto-Verify is to divide the number of CPU cores equally across all verification tools in the portfolio and give each GPU-based verifier 1 GPU. It should be specified in the scenario if a verification tool needs a GPU. This way all the CPU cores and GPUs in the machine can be fully utilised.

The portfolios used in Auto-Verify are serialisable, meaning they can be exported to and read from different formats. This allows portfolios to be shared and stored for further usage.

4.4.2 Parallel Portfolio Execution

After the portfolio has been constructed, Auto-Verify also provides interfaces for executing the portfolio in a fully parallel fashion, see Listing 4.
As discussed in Section 2.4.3, the procedure for running a parallel portfolio of verification tools works as follows:

1. Each verification tool in the portfolio is launched in parallel on the same verification instance. All verification tools are assigned unique CPU cores and GPUs to run their verification procedures on.

```python
from autoverify.portfolio import Portfolio, PortfolioRunner
from autoverify.instances import read_vnncomp_instances

instances = read_vnncomp_instances("mnist_fc")
portfolio = Portfolio.from_json("example_pf.json")

pf_runner = PortfolioRunner(portfolio)
pf_runner.verify_instances(instances, out_csv="results.csv")
```

Listing 4: Example of how to read and run a parallel portfolio of verification tools using Auto-Verify. Every verification tool inside the portfolio is launched in parallel for each instance, once one of the verification tools solves the instance (or a timeout is reached) the entire portfolio moves on to the next instance. Results are accumulated into a CSV file.

2. If one of the verification tools finds that the property is (un)satisfiable the instance is considered solved and the procedure moves on the the next instance.
3. If every verification tool times out, the instance is treated as a timeout and the procedure moves on to the next instance.

By running all the verification tools in parallel, we ensure that the strength of each (configured) verification tool is leveraged. Note that is also possible to select a verification tool for each instance, which will free up additional resources for this verification tool since it does not have to share with the verification tools in the portfolio. This approach has been shown to work well in MIP contexts [Xu+11], but is outside the scope of this thesis.

4.5 Available Verification Tools

Auto-Verify currently supports the four verification tools listed below out of the box. These were selected based on their support for the ONNX and VNNLIB formats, their vast configuration space, being open-source, and their performance in VNNCOMP 2022.

α,β-CROWN [Zha+18; Xu+21; Wan+21; Zha+22] α,β-CROWN is a neural network verification tool that is based on the linear bound propagation framework. It combines different methods into one: CROWN [Zha+18] and α-CROWN [Xu+20] for optimising intermediate bounds, β-crown [Wan+21] for branching and bounding, GCP-CROWN [Zha+22] as a cutting-plane method, and MIP formulations for smaller networks. α,β-CROWN supports a large range of network architectures, is GPU optimised and won the 2021 and 2022 editions of VNNCOMP.

nnenum [Bak21] Nnenum is a CPU-based verification tool that implements efficient path enumeration by making use of star set overapproximations [Tra+19], the ImageStar [Tra+20b] method, and parallelised RELU-case splitting [Bak+20].

VeriNet [HL20; HL21] VeriNet is a symbolic interval propagation-based neural network verification tool. By using symbolic interval propagation to create linear abstractions of networks, LP solvers can be used to find solutions. VeriNet further uses a branch-and-bound phase to achieve completeness and implements further optimisations such as gradient-based local search, optimal relaxations and node splitting. VeriNet supports a wide array of activation functions and can make use of the GPU.

OVAL-BaB [Bun+18; Bun+20; Pal+21a; Pal+21b; Pal+21c] OVAL-BaB uses a specialised branch-and-bound framework for the verification of neural networks. Methods used include various branching strategies such as FSB [Pal+21c], and different bounding techniques such as β-crown [Wan+21] and active set [Pal+21a]. Counter-examples are found using MI-FGSM [Don+18]. OVAL-BaB is GPU optimised.

Chapter 5

Experiments

To assess the performance of parallel portfolios of verification tools constructed and executed by Auto-Verify, we set up a number of experiments on existing benchmarks from the neural network verification literature. The following sections will explain our choices and detail the experimental setups.

5.1 Benchmarks

To study the effectiveness of parallel portfolios, we selected three well-studied and broadly supported benchmarks. The main criteria we used to select these benchmarks were their support for verification tools that are included by default in Auto-Verify, and if they had been used in existing studies on neural network verification, to ensure that verification algorithms are actually tested and studied on these networks. Note that not all verification tools support all types of operations used in neural networks, with most only supporting RELU-based networks [Kön+23]. Using these criteria, we selected the following three benchmarks:

- **MNIST**
  We took 25 networks trained on the MNIST [Den12] dataset from existing neural network verification literature, and 100 local robustness properties ($\epsilon = 0.012$) for each network, leading to a total of 2500 verification instances. Each instance has a 5-minute timeout.

- **CIFAR**
  From the existing literature on neural network verification, we selected 33 networks trained on the CIFAR-10 [KH09] dataset. Similar to the MNIST benchmark, we used 100 local robustness properties ($\epsilon = 0.012$) for each network, which created 3300 verification instances in total. Each instance has a 5-minute timeout.

- **TLL Verify Bench**
  TLL Verify Bench is a benchmark that consists of Two Level Lattice [FKS22] networks, which are networks that adhere to a certain architecture that favours verifiability. The benchmark contains 32 TLL networks and 32 local robustness properties (1 per network), totalling 32 verification instances. Each instance has a 10-minute timeout.

5.2 Experiments

To determine the performance of parallel portfolios of verification tools, we compare their running times against running four individual verification tools: α,β-CROWN, Oval-BaB, nnenum, and VeriNet. Both the portfolio and individual verification tools get access to all resources of a single compute node, during which we measure in wall-clock time how long it takes to solve each instance in the benchmark. While work by König et al. [KHR22; Kön+23] measured CPU-time, the verification tools we experiment with also make use of the GPU (except nnenum). To compare the performance of the verification tools and parallel portfolios, we give each full access to all the hardware on a node and measure performance in wall-clock time. Two types of parallel portfolios constructed by Hydra are tested:

- Portfolio of verification tools with their default configurations
- Portfolio of configured verification tools

As described in Section 4.4.1, a hyperparameter α was introduced to control the amount of time Hydra spends on selecting and configuring a verification tool each iteration. For constructing the portfolios of verification tools with default configurations α is simply set to 0, meaning Hydra spends all its budget each iteration on selecting a verifier to add to the portfolio. For constructing portfolios of configured verification tools, α is set to 0.9, indicating that 90% of the budget is spent on configuring the verification tool that could be added to the portfolio, and 10% was spent on selecting this verification tool from the available candidates. Note that we do not have the resources to determine the optimal value for α, but went for a rather large value of α as the hyperparameter space of a verification tool is a significantly larger search space than choosing from a number of verification tools (four options in the case of our experiments). Future work could study the effect α has on the portfolio construction procedure by studying how much time the selection and configuration phases both need compared to each other to find the best-performing verification tool or configuration and if this has a significant impact on performance.

5.3 Evaluation

Our evaluation is centred around two key concepts: The speed at which instances are solved and the fraction of solved instances. To get an accurate picture of these two metrics, we:

- Compute the sum, mean and median of the running times over all instances (where timeout values are used if an instance times out, see Section 5.1 for exact values);
- report the fraction of solved, timed out and crashed instances;
- plot an Empirical Cumulative Distribution Function (ECDF) plot, which shows an estimate for the distribution of solving times for the different verification tools and parallel portfolios.

Runs on verification instances inside the configurator were scored with wall-clock time. Following best practices in algorithm configuration literature, runs that time out are penalised by multiplying the cost by some factor k. Conforming with the existing literature on algorithm configuration, we use $k = 10$, meaning if a run times out at 300 seconds, the total cost becomes $300 * 10$. This cost metric is referred to in the literature as penalised average running time k; PAR10 in this case.

5.4 Experimental Setup

Each evaluation, be that a parallel portfolio or individual verification tool, gets access to one full compute node. The nodes we used in this study ran on CentOS Linux 7 and included the following hardware:

- Intel Xeon E5-2683 CPU, 2.10 GHz, 32 cores
- 2x NVIDIA GeForce GTX 1080 Ti, 11 GB Memory
- 94 GB RAM

Hydra was given 24 hours per benchmark to construct a parallel portfolio. We used the most up-to-date (as of September 2023) publicly available versions of all verification tools we studied. For the configurator, we used SMAC3 version 2.0.2.

Once the portfolios were constructed they were run on the aforementioned benchmarks, and their wall-clock running times and instance results were recorded. The results are presented in the next section.

Chapter 6

Results

This section shows and describes the outcome of our experiments. A detailed discussion of the results can be found in Section 7. In the tables and figures, portfolios of verification tools with their default hyperparameters are referred to as "Portfolio [D]", and portfolios of configured verification tools as "Portfolio [C]".

As previously described, Hydra was given a budget of 24 hours for each experiment to construct a parallel portfolio. We show the results per benchmark, where we show ECDF plots containing the performance of the individual verification tools and the parallel portfolios. Table 6.1 shows the composition of each portfolio and the fraction of instances for each benchmark on which a verification tool was the fastest. Note that in portfolios of verification tools with their default hyperparameters, the same verification tool cannot appear twice, because it would be an exact duplicate, and thus have the same performance. In the configured portfolios this is not the case, as the same verification tool with different configurations will behave differently. Table 6.2 shows the number of instances on which a verification tool or portfolio produced errors that halted the verification process. In the subsequent plots and tables, these errors are grouped with timeouts for simplicity.

Table 6.1: Contents of portfolios and the fraction of solved instances on which they were the fastest for the MNIST (2500 instances), CIFAR (3300 instances) and TLL Verify Bench (32 instances) datasets.

MNIST		CIFAR		TLL	
Portfolio [D]	**Frac. Fastest**	**Portfolio [D]**	**Frac. Fastest**	**Portfolio [D]**	**Frac. Fastest**
nnenum	0.898	α,β-CROWN	0.465	nnenum	0.591
Oval-BaB	0.093	Oval-BaB	0.340	α,β-CROWN	0.318
α,β-CROWN	0.009	nnenum	0.196	Oval-BaB	0.091
Portfolio [C]	**Frac. Fastest**	**Portfolio [C]**	**Frac. Fastest**	**Portfolio [C]**	**Frac. Fastest**
nnenum	0.917	α,β-CROWN-1	0.494	α,β-CROWN	0.455
Oval-Bab	0.083	α,β-CROWN-2	0.470	nnenum	0.409
		nnenum	0.037	Oval-BaB	0.136

Table 6.2: Fraction of instances for the MNIST (2500 instances), CIFAR (3300) instances and TLL Verify Bench (32 instances) on which the verification tools or portfolios produced errors that halted verification.

MNIST		CIFAR		TLL	
Verification Tool	**Frac. Errors**	**Verification Tool**	**Frac. Errors**	**Verification Tool**	**Frac. Errors**
nnenum	0.000	nnenum	0.818	nnenum	0.344
Oval-BaB	0.000	Oval-BaB	0.000	Oval-BaB	0.000
α,β-CROWN	0.000	α,β-CROWN	0.089	α,β-CROWN	0.000
VeriNet	0.081	VeriNet	0.223	VeriNet	0.025
Portfolio [D]	0.000	Portfolio [D]	0.130	Portfolio [D]	0.000
Portfolio [C]	0.000	Portfolio [C]	0.245	Portfolio [C]	0.281

6.1 MNIST

Table 6.3, and Figures 6.1 and 6.2 show the results on the MNIST dataset. The portfolio of verification tools with default configurations performs the best, achieving the best score in all categories. This is also visible from the ECDF plot, where all the graphs for the individual verification tools are largely contained inside the portfolio graph. Out of the individual verification tools, α,β-CROWN performs the best, followed by nnenum. Both Oval-BaB and VeriNet perform significantly worse. Compared to the best individual verification tool, the parallel portfolio achieved a speedup by a factor of 1.5 in terms of total time, and reduced the number of timeouts by a factor of 1.08. From Table 6.1 we can see that in close to 90% of solved instances, nnenum was the fastest. Table 6.2 shows that VeriNet was the only verification tool that produced errors on the MNIST dataset, on 8.1% of the instances.

The constructed portfolio of configured verification tools contained a configuration for nnenum and Oval-BaB, see Table 6.1. From the data we can see that the configured parallel portfolio performs worse than the unconfigured portfolio, having a higher total time and more timeouts than the unconfigured portfolio. Performance decreased by a factor of 0.92, timeouts by a factor of 0.95, and on more than 90% of the solved instances, nnenum was the fastest.

Table 6.3: Aggregated measures of individual verifiers and parallel portfolios on the MNIST dataset (2500 instances). "Portfolio [D]" is the portfolio with verification tools and their default configurations, "Portfolio [C]" is the portfolio with configured verification tools.

Verifier	**Total (s)**	**Mean (s)**	**Median (s)**	**Timeouts**
nnenum	59715	23.9	2.6	141
α,β-CROWN	55210	22.1	9.4	99
Oval-BaB	369441	147.8	8.0	1199
VeriNet	146234	58.8	11.1	370
Portfolio [D]	36929	14.8	2.5	92
Portfolio [C]	40024	16.0	2.5	104

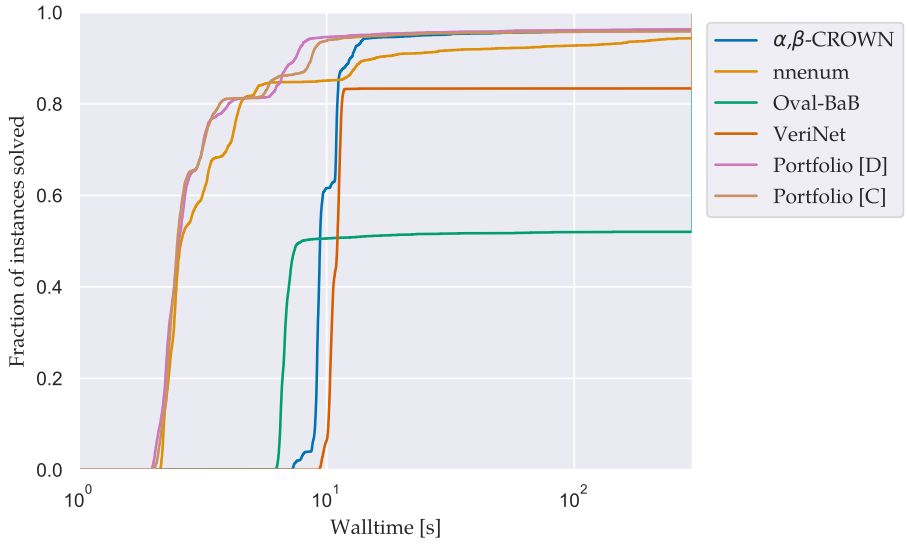

Figure 6.1: ECDF Plot of individual verification tools and two parallel portfolios showing the fraction of solved instances on the MNIST benchmark (2500 instances). The parallel portfolios were constructed using the Hydra algorithm. "Portfolio [D]" is the portfolio with verification tools and their default configurations, "Portfolio [C]" is the portfolio with configured verification tools

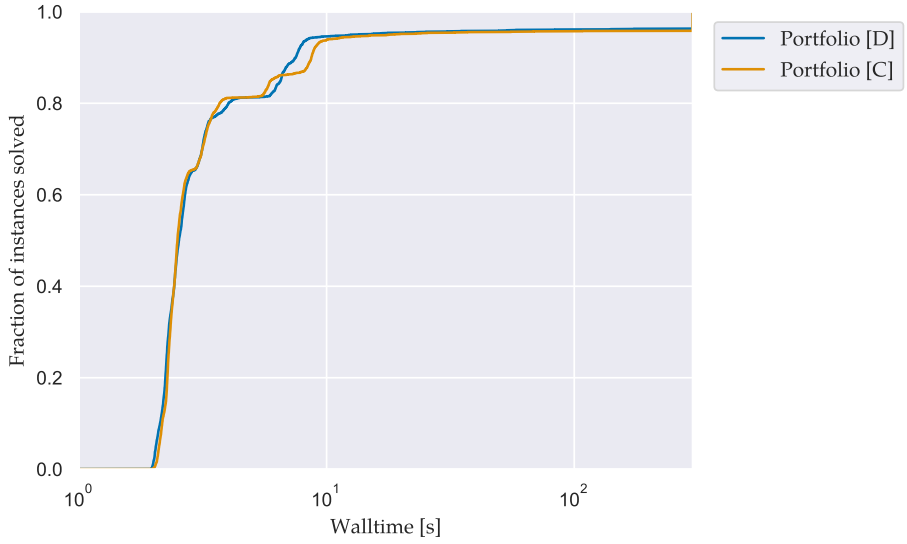

Figure 6.2: ECDF Plot of the portfolio of default configurations and the configured portfolio, showing the fraction of solved instances on the MNIST benchmark (2500 instances). Portfolios were constructed with Hydra ($\alpha = 0.9$). "Portfolio [D]" is the portfolio with verification tools and their default configurations, "Portfolio [C]" is the portfolio with configured verification tools.

6.2 CIFAR

Table 6.4, and Figures 6.3 and 6.4 show the results on the CIFAR dataset. Similar to MNIST, the parallel portfolio of verification tools with default configurations performs the best on each metric. The speedup factor over the best individual verification tool, once again α,β-CROWN, is 1.1. The number of timeouts compared to the best individual verification

tool was reduced by a factor of 1.12. The margin between the parallel portfolio and α,β-CROWN on the CIFAR dataset is smaller than on the MNIST dataset. As visible from Table 6.1, α,β-CROWN is the fastest verification tool on the majority of solved instances (46.5%), but not by a large margin. The other two verification tools are the fastest on a significant fraction of the solved instances as well (Oval-BaB on 34% and nnenum on 19.6%). From Table 6.2, we can see that Oval-BaB was the only verification tool that did not produce any errors on the CIFAR dataset. We can see that most verification tools produce a significant amount of errors on the CIFAR dataset, with the highest being nnenum, which produced errors on 81.8% of instances.

Contained in the portfolio of configured verification tools were two configurations for α,β-CROWN and one configuration for nnenum (see Table 6.1). The performance of the configured portfolio is noticeably worse than the unconfigured portfolio, with the number of timeouts increasing by a factor of 1.88 and total time by a factor of 1.74. Both α,β-CROWN configurations solved close to the same fraction of instances in the fastest time, with the nnenum configuration only solving 3.7% of instances the fastest.

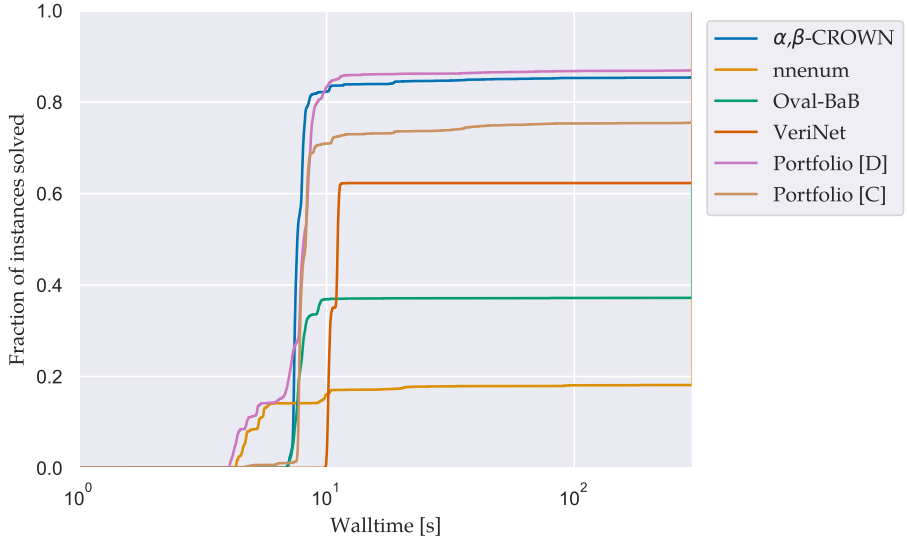

Figure 6.3: ECDF Plot of individual verification tools and two parallel portfolios showing the fraction of solved instances on the CIFAR benchmark (3300 instances). The parallel portfolio were constructed using the Hydra algorithm. "Portfolio [D]" is the portfolio with verification tools and their default configurations, "Portfolio [C]" is the portfolio with configured verification tools.

Table 6.4: Aggregated measures of individual verifiers and parallel portfolios on the CIFAR dataset (3300 instances). "Portfolio [D]" is the portfolio with verification tools and their default configurations, "Portfolio [C]" is the portfolio with configured verification tools.

Verifier	Total (s)	Mean (s)	Median (s)	Timeouts
nnenum	814942	246.9	300.0	2701
α,β-CROWN	168240	51.0	7.6	482
Oval-BaB	631454	191.3	300.0	2072
VeriNet	394956	119.7	11.1	1237
Portfolio [D]	152958	46.4	8.1	430
Portfolio [C]	265894	80.6	8.2	808

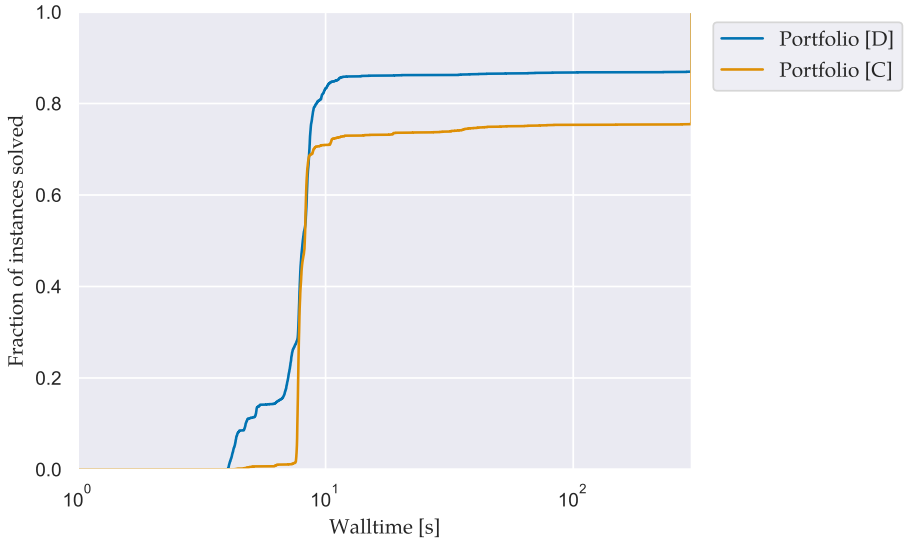

Figure 6.4: ECDF Plot of the portfolio of default configurations and the configured portfolio, showing the fraction of solved instances on the CIFAR benchmark (3300 instances). Portfolios were constructed with Hydra ($\alpha = 0.9$). "Portfolio [D]" is the portfolio with verification tools and their default configurations, "Portfolio [C]" is the portfolio with configured verification tools.

6.3 TLL Verify Bench

Table 6.5, and Figures 6.5 and 6.6 show the results on the TLL Verify Bench dataset. This time, the portfolio of configured verification tools has the best performance, although only by a factor of 1.007 compared to the best individual verification tool. The number of timeouts remained the same. Performance between verification tools for this benchmark is a lot closer compared to the MNIST and CIFAR benchmarks, which is also due to the fact the benchmark has only 32 instances. Regardless, the portfolio is able to leverage the fact that a single verification tool is not the fastest on every instance, even on a smaller benchmark, since the total verification time is still lower compared to running one verification tool. Table 6.1 confirms this, where we can see that all three verification tools contribute to being the fastest on at least a couple of instances. In Table 6.2 we can see that nnenum and Portfolio [C] both produce a large number of errors on the TLL Verify Bench dataset, while the other verification tools do not produce any errors.

The portfolio of configured verification tools contained a configuration for α,β-CROWN, nnenum and Oval-BaB (see Table 6.1). The configured portfolio was able to slightly improve over the unconfigured portfolio in total time but still had an equal amount of timeouts. Out of the verification tools in the portfolio (see Table 6.1), α,β-CROWN was the fastest on 45.5% of solved instances followed closely by nnenum on 40.9% of instances, Oval-BaB was the fastest on 13.6% of the solved instances. We can see in Table 6.2 that nnenum, VeriNet, and Portfolio [C] produced errors.

Table 6.5: Aggregated measures of individual verifiers and parallel portfolios on the TLL Verify Bench dataset (32 instances). "Portfolio [D]" is the portfolio with verification tools and their default configurations, "Portfolio [C]" is the portfolio with configured verification tools.

Verifier	Total (s)	Mean (s)	Median (s)	Timeouts
nnenum	6822	213.2	20.1	11
α,β-CROWN	6213	194.2	9.6	10
Oval-BaB	6865	214.5	19.0	11
VeriNet	12715	397.4	600.0	15
Portfolio [D]	6188	193.4	9.7	10
Portfolio [C]	6169	192.8	8.1	10

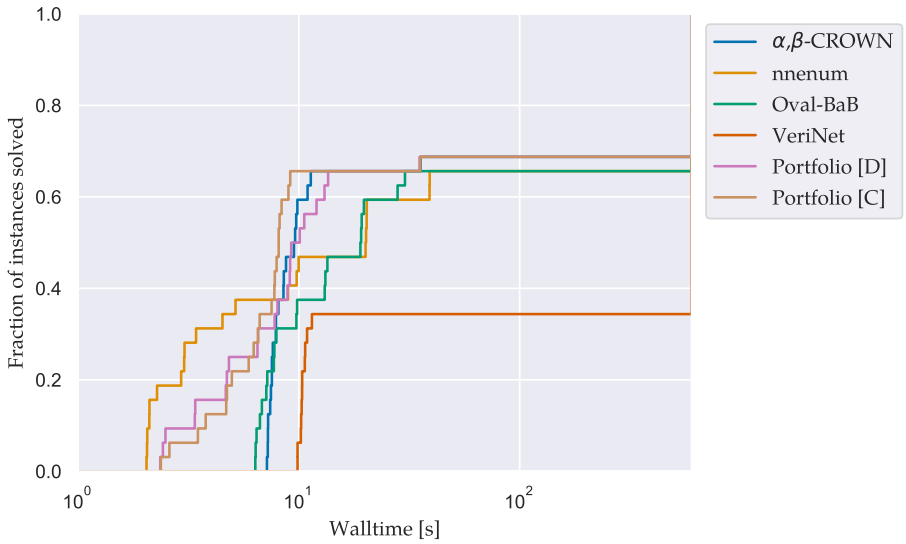

Figure 6.5: ECDF Plot of individual verification tools and two parallel portfolios showing the fraction of solved instances on the TLL Verify Bench benchmark (32 instances). The parallel portfolio were constructed using the Hydra algorithm. "Portfolio [D]" is the portfolio with verification tools and their default configurations, "Portfolio [C]" is the portfolio with configured verification tools.

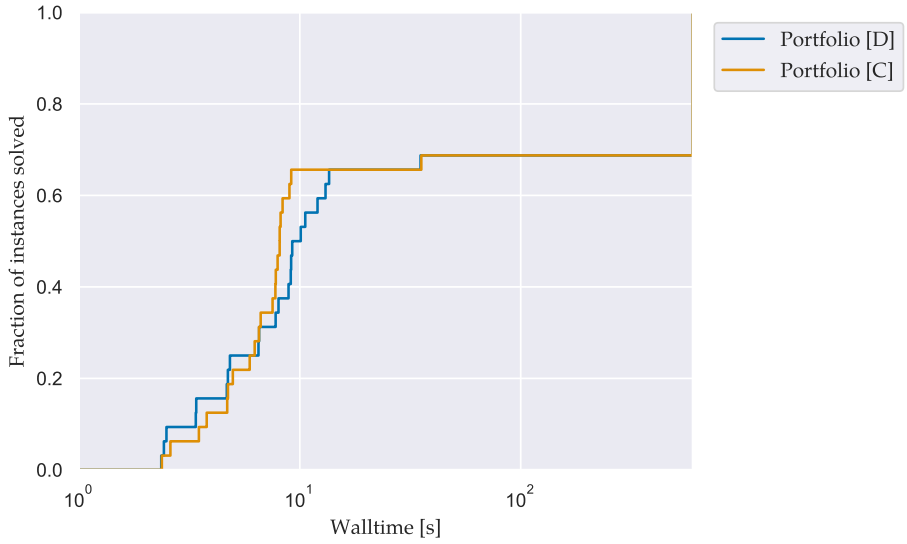

Figure 6.6: ECDF Plot of the portfolio of default configurations and the configured portfolio, showing the fraction of solved instances on the TLL Verify Bench benchmark (32 instances). Portfolios were constructed with Hydra ($\alpha = 0.9$). "Portfolio [D]" is the portfolio with verification tools and their default configurations, "Portfolio [C]" is the portfolio with configured verification tools.

Chapter 7

Discussion

Our results confirm the findings of prior work by König et al. [KHR22; Kön+23] that there is not one single verification tool that dominates all others on every instance. While the degree of complementarity varied per benchmark, combining the verification tools in a principled way always led to an improvement in the total time spent on verification and almost always to an improvement in the number of solved instances.

On the MNIST, CIFAR and TLL Verify Bench benchmarks we were able to achieve improvements in total time by factors of 1.5, 1.1, and 1.007 respectively. The timeouts were reduced by factors of 1.08 and 1.12 on the MNIST and CIFAR benchmarks, while the timeouts on the TLL Verify Bench stayed the same. These improvements were from running portfolios of default configurations in parallel, and thus do not require much additional effort or resources to perform. Our configuration procedures never ended up selecting VeriNet to be in the parallel portfolios. This indicates VeriNet is often outperformed by one of the other three verification tools, indicating there is little to no complementarity with the other three verification tools we used. The usefulness of parallel portfolios seems to grow with the size and diversity of benchmarks because this is where the complementarity of a portfolio can be maximally exploited.

Despite this, automatically configuring the verification tools still poses a significant challenge. From our experiments, the main factors contributing to the difficulty of the problem were the following three:

- Long running times
- Heterogeneous instances
- Error-prone verification tools

Even though we limited our timeouts to 5-10 minutes per instance, this still means the process of configuration might take a long time if benchmarks have thousands of instances. Adding to this, instances are often heterogeneous which makes it difficult to assess the performance of a configuration after only evaluating a subset of all instances. This is also why we think the parallel portfolio of configured verification tools can turn out to be worse than the default configurations; the configurator can make incorrect judgements about the performance of a configuration if the budget does not allow for each configuration to be evaluated on all instances.

Other problems that we encountered were of a more technical nature. Verification tools often return (undefined) errors when trying to automatically configure their hyperparameters. The space of valid hyperparameters was not strongly defined for any of the verification tools we used, and can also depend on the instance (e.g. certain hyperparameters only work for certain network architectures). For example, by sampling random configurations for α,β-CROWN and using these configurations on randomly sampled instances of the MNIST benchmark, we found that in 30% of all cases α,β-CROWN would produce an undefined error. Having to consider these factors meant that within our available resources, achieving meaningful performance improvements proved very difficult. Furthermore, we can see from Table 6.2 that benchmarks can sometimes be largely unsupported by verification tools. For example, nnenum produced errors on 81.8% of instances in the CIFAR dataset. Despite this, nnenum was still selected to be in both portfolios, indicating that it performed well on the instances that it did not crash on. The CIFAR dataset in particular seems to be the least well-supported among the verification tools, with each producing a number of errors, except for Oval-BaB. Having broader benchmark support for each verification tool would also benefit the performance of portfolios of verification tools, as it would not exclude some of the verification tools in the portfolio from being ran on some of the instances.

We think that verification tools would greatly benefit from strongly defining hyperparameter spaces, from both a performance and user-friendliness perspective. To achieve the best performance in VNNCOMP, three out of the four verification tools in Auto-Verify used a different configuration for each benchmark (all except nnenum), with α,β-CROWN sometimes using multiple configurations per benchmark. This highlights the importance the configurations have on maximising performance. While authors of verification tools can use their expertise to create these configurations, users without the required domain knowledge cannot. As discussed, attempting to find a well-performing configuration in an automated fashion, is also challenging. The organisers of VNNCOMP also discussed this issue in their report reflecting on the first three years of the competition [Bri+23]. They acknowledge that tuning verification tools per benchmark or even per instance is problematic and time-consuming when trying to adapt these verification tools to new problems. Because of this, they suggest that future iterations of VNNCOMP might restrict tuning for some benchmarks to encourage tools to implement auto-tuning strategies.

Chapter 8

Conclusions and Future Work

8.1 Conclusion

In this thesis we presented Auto-Verify: a tool for portfolio-based verification of neural network properties. Auto-Verify provides uniform interfaces, parameter spaces, parallel portfolios, and further utilities to make using neural network verification easy and convenient. By using Auto-Verify, it becomes possible to quickly iterate and try out new ideas without having to deal with the many pain points of neural network verification. We tested Auto-Verify by constructing two different types of parallel portfolios and showed that we can improve performance on well-known benchmarks by factors of up to 1.5 in terms of total time, and reduce timeouts by a factor of 1.12.

Automatically configuring the hyperparameters of neural network verification tools remains a challenge, due to the long running times, heterogeneous instances, and loosely defined hyperparameter spaces. With the resources at our disposal, we were not able to meaningfully improve performance by configuring the neural network verification tools in our portfolios.

Employing verification tools on a wide range of benchmarks can still be a time-consuming process. The lack of support for different architectures among verification tools makes it difficult to meaningfully compare them on anything except the most well-known benchmarks. Defining the hyperparameter spaces of verification tools more strictly would further streamline the usage of algorithm configuration, since determining the hyperparameter space takes quite some effort, due to the lack of documentation and ranges of valid values provided for these hyperparameters. Another observation we made is that benchmarks with large networks quickly run out of GPU memory, even on relatively small batch sizes. Needing expensive GPU hardware to run large benchmarks is something that could hamper the broad adoption of neural network verification.

8.2 Future Work

Different automated algorithm configuration and portfolio methods could be studied in the context of neural network verification. Many such methods, such as algorithm selection [Ker+19], have already been shown to work in the context of SAT solving [Xu+08] (which is related to neural network verification, see Section 3.1). Studying automated algorithm configuration methods that have been proven to work in other optimisation fields could further analyse which methods are most suited to increasing the performance of neural

network verification tools.

Current experiments can be easily scaled up by using better hardware and giving configuration procedures more time and including more benchmarks. Additional verification tools can also be added to further make use of the complementarity of verification tools.

Creating strongly defined hyperparameter spaces in verification tools would improve the efficiency of applying automated algorithm configuration procedures to neural network verification, by reducing the time spent on evaluating invalid configurations and reducing the size of the hyperparameter space that has to be searched.

Furthermore, combining existing state-of-the-art techniques into one single verification tool would greatly simplify the process of algorithm configuration. Having to rely on multiple independent, sometimes unmaintained, verification tools introduces a lot of overhead which makes it easy for methods that perform well on a smaller subset of instances to be overlooked. Alternatively, a framework that facilitates implementing verification methods could also achieve a similar desired outcome: combining the work of many authors to make neural network verification more efficient for everyone.

References

[GSS15] 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)*. 2015, pp. 1–11.

[SVS19] Jiawei Su, Danilo Vasconcellos Vargas, and Kouichi Sakurai. "One Pixel Attack for Fooling Deep Neural Networks". In: *IEEE Transactions on Evolutionary Computation* 23.5 (2019), pp. 828–841.

[Sun+22] Weidi Sun, Yuteng Lu, Xiyue Zhang, and Meng Sun. "DeepGlobal: A framework for global robustness verification of feedforward neural networks". In: *Journal of System Architecture* 128 (2022), p. 102582.

[Xu+20] Kaidi Xu, Zhouxing Shi, Huan Zhang, Yihan Wang, Kai-Wei Chang, Minlie Huang, Bhavya Kailkhura, Xue Lin, and Cho-Jui Hsieh. "Automatic Perturbation Analysis for Scalable Certified Robustness and Beyond". In: *Advances in Neural Information Processing Systems 33 (NeurIPS 2020)*. 2020, pp. 1129–1141.

[KHR22] 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". In: *Machine Learning* 111.12 (2022), pp. 4565–4584.

[Kön+23] Matthias König, Annelot 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 (SafeAI 2023)*. Vol. 3381. CEUR Workshop Proceedings. CEUR-WS.org, 2023, pp. 1–10.

[Liu+21] Changliu Liu, Tomer Arnon, Christopher Lazarus, Christopher A. Strong, Clark W. Barrett, and Mykel J. Kochenderfer. "Algorithms for Verifying Deep Neural Networks". In: *Foundations and Trends in Optimization* 4.3-4 (2021), pp. 244–404.

[Hin05] Geoffrey E. Hinton. "What kind of graphical model is the brain?" In: *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI 2005)*. Professional Book Center, 2005, pp. 1765–1775.

[LBH15] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning". In: *Nature* 521.7553 (2015), pp. 436–444.

[Sze+14] 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)*. 2014, pp. 1–10.

[Yua+19] Xiaoyong Yuan, Pan He, Qile Zhu, and Xiaolin Li. "Adversarial Examples: Attacks and Defenses for Deep Learning". In: *IEEE Transactions on Neural Networks and Learning Systems* 30.9 (2019), pp. 2805–2824.

[Ues+18] Jonathan Uesato, Brendan O'Donoghue, Pushmeet Kohli, and Aaron van den Oord. "Adversarial Risk and the Dangers of Evaluating Against Weak Attacks". In: *Proceedings of the 35th International Conference on Machine Learning (ICML 2018)*. Vol. 80. Proceedings of Machine Learning Research. PMLR, 2018, pp. 5025–5034.

[ACW18] Anish Athalye, Nicholas Carlini, and David A. Wagner. "Obfuscated Gradients Give a False Sense of Security: Circumventing Defenses to Adversarial Examples". In: *Proceedings of the 35th International Conference on Machine Learning (ICML 2018)*. Vol. 80. Proceedings of Machine Learning Research. PMLR, 2018, pp. 274–283.

[TXT19] 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)*. 2019, pp. 1–21.

[Kat+17] 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)*. Vol. 10426. Lecture Notes in Computer Science. Springer, 2017, pp. 97–117.

[Kat+19] 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)*. Vol. 11561. Lecture Notes in Computer Science. Springer, 2019, pp. 443–452.

[Sch+15] 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)*. Sächsische Landesbibliothek, 2015, pp. 30–40.

[Wan+18a] 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)*. 2018, pp. 6369–6379.

[XTJ18] Weiming Xiang, Hoang-Dung Tran, and Taylor T Johnson. "Output Reachable Set Estimation and Verification for Multilayer Neural Networks". In: *IEEE Transactions on Neural Networks and Learning Systems* 29.11 (2018), pp. 5777–5783.

[Bak21] Stanley Bak. "nnenum: Verification of ReLU Neural Networks with Optimized Abstraction Refinement". In: *Proceedings of the 13th International Symposium on NASA Formal Methods (NFM 2021)*. Vol. 12673. Lecture Notes in Computer Science. Springer, 2021, pp. 19–36.

[Wan+21] Shiqi Wang, Huan Zhang, Kaidi Xu, Xue Lin, Suman Jana, Cho-Jui Hsieh, and J Zico Kolter. "Beta-CROWN: Efficient bound propagation with per-neuron split constraints for complete and incomplete neural network verification". In: *Advances in Neural Information Processing Systems 34 (NeurIPS 2021)*. 2021, pp. 29909–29921.

[Fer+22] 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)*. 2022, pp. 1–15.

[Bot+20] 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)*. AAAI Press, 2020, pp. 3291–3299.

[Bas+16] 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)*. 2016, pp. 2613–2621.

[Bun+18] 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)*. 2018, pp. 4795–4804.

[Dvi+18] 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)*. AUAI Press, 2018, pp. 550–559.

[Ehl17] 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)*. Vol. 10482. Lecture Notes in Computer Science. Springer, 2017, pp. 269–286.

[Geh+18] 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)*. IEEE, 2018, pp. 3–18.

[HL20] Patrick Henriksen and Alessio R. Lomuscio. "Efficient Neural Network Verification via Adaptive Refinement and Adversarial Search". In: *Proceedings of the 24th European Conference on Artificial Intelligence (ECAI 2020)*. Frontiers in Artificial Intelligence and Applications. IOS Press, 2020, pp. 2513–2520.

[HL21] Patrick Henriksen and Alessio Lomuscio. "DEEPSPLIT: An Efficient Splitting Method for Neural Network Verification via Indirect Effect Analysis". In: *Proceedings of the 30th International Joint Conference on Artificial Intelligence, (IJCAI 2021)*. ijcai.org, 2021, pp. 2549–2555.

[Bro00] Michael W Browne. "Cross-Validation Methods". In: *Journal of Mathematical Psychology* 44.1 (2000), pp. 108–132.

[XZZ20] Chang Xiao, Peilin Zhong, and Changxi Zheng. "Enhancing Adversarial Defense by k-Winners-Take-All". In: *Proceedings of the 8th International Conference on Learning Representations (ICLR 2020)*. 2020, pp. 1–29.

[Ma+18] Xingjun Ma, Bo Li, Yisen Wang, Sarah M. Erfani, Sudanthi N. R. Wijewickrema, Grant Schoenebeck, Dawn Song, Michael E. Houle, and James Bailey. "Characterizing Adversarial Subspaces Using Local Intrinsic Dimensionality". In: *Proceedings of the 6th International Conference on Learning Representations (ICLR 2018)*. 2018, pp. 1–15.

[Guo+18] Chuan Guo, Mayank Rana, Moustapha Cissé, and Laurens van der Maaten. "Countering Adversarial Images using Input Transformations". In: *Proceedings of the 6th International Conference on Learning Representations (ICLR 2018)*. 2018, pp. 1–12.

[VS19] Gunjan Verma and Ananthram Swami. "Error Correcting Output Codes Improve Probability Estimation and Adversarial Robustness of Deep Neural Networks". In: *Advances in Neural Information Processing Systems 32 (NeurIPS 2019)*. 2019, pp. 8643–8653.

[Pan+19] Tianyu Pang, Kun Xu, Chao Du, Ning Chen, and Jun Zhu. "Improving Adversarial Robustness via Promoting Ensemble Diversity". In: *Proceedings of the 36th International Conference on Machine Learning (ICML 2019)*. Vol. 97. Proceedings of Machine Learning Research. PMLR, 2019, pp. 4970–4979.

[SRR20] Sanchari Sen, Balaraman Ravindran, and Anand Raghunathan. "EMPIR: Ensembles of Mixed Precision Deep Networks for Increased Robustness Against Adversarial Attacks". In: *Proceedings of the 8th International Conference on Learning Representations (ICLR 2020)*. 2020, pp. 1–12.

[Tra+20a] Florian Tramèr, Nicholas Carlini, Wieland Brendel, and Aleksander Madry. "On Adaptive Attacks to Adversarial Example Defenses". In: *Advances in Neural Information Processing Systems 33 (NeurIPS 2020)*. 2020, pp. 1633–1645.

[Car+17] Nicholas Carlini, Guy Katz, Clark Barrett, and David L Dill. "Provably Minimally-Distorted Adversarial Examples". In: *arXiv preprint arXiv:1709.10207* (2017).

[Che+18] 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)*. AAAI Press, 2018, pp. 10–17.

[Pap+16] 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)*. IEEE, 2016, pp. 582–597.

[SED21] 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)*. Vol. 12759. Lecture Notes in Computer Science. Springer, 2021, pp. 137–150.

[LXL23] Linyi Li, Tao Xie, and Bo Li. "SoK: Certified Robustness for Deep Neural Networks". In: *Proceedings of the 44th IEEE Symposium on Security and Privacy (IEEE S&P2023)*. IEEE, 2023, pp. 1289–1310.

[PT10] Luca Pulina and Armando Tacchella. "An Abstraction-Refinement Approach to Verification of Artificial Neural Networks". In: *Proceedings of the 22nd International Conference on Computer Aided Verification (CAV 2010)*. Vol. 6174. Lecture Notes in Computer Science. Springer, 2010, pp. 243–257.

[PT11] Luca Pulina and Armando Tacchella. "Checking Safety of Neural Networks with SMT Solvers: A Comparative Evaluation". In: *Proceedings of the 12th International Conference of the Italian Association for Artificial Intelligence (AI*IA 2011)*. Vol. 6934. Lecture Notes in Computer Science. Springer, 2011, pp. 127–138.

[PT12] Luca Pulina and Armando Tacchella. "Challenging SMT solvers to verify neural networks". In: *AI Communications* 25.2 (2012), pp. 117–135.

[Dan02] George B. Dantzig. "Linear Programming". In: *Operations Research* 50.1 (2002), pp. 42–47.

[CNR17] 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)*. 2017, pp. 251–268.

[LM17] Alessio Lomuscio and Lalit Maganti. "An approach to reachability analysis for feed-forward ReLU neural networks". In: *arXiv preprint arXiv:1706.07351* (2017).

[Dut+18] 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)*. 2018, pp. 121–138.

[FJ18] Matteo Fischetti and Jason Jo. "Deep neural networks and mixed integer linear optimization". In: *Constraints* 23.3 (2018), pp. 296–309.

[LD60] Ailsa H. Land and Alison G. Doig. "An Automatic Method for Solving Discrete Programming Problems". In: *Econometrica* 28.3 (1960), pp. 497–520.

[Xu+21] Kaidi Xu, Huan Zhang, Shiqi Wang, Yihan Wang, Suman Jana, Xue Lin, and Cho-Jui Hsieh. "Fast and Complete: Enabling Complete Neural Network Verification with Rapid and Massively Parallel Incomplete Verifiers". In: *Proceedings of the 9th International Conference on Learning Representations (ICLR 2021)*. 2021, pp. 1–15.

[Pal+21a] Alessandro De Palma, Harkirat S. Behl, Rudy Bunel, Philip H. S. Torr, and M. Pawan Kumar. "Scaling the Convex Barrier with Active Sets". In: *Proceedings of the 9th International Conference on Learning Representations (ICLR 2021)*. 2021, pp. 1–27.

[Pal+21b] Alessandro De Palma, Harkirat Singh Behl, Rudy Bunel, Philip H. S. Torr, and M. Pawan Kumar. "Scaling the Convex Barrier with Sparse Dual Algorithms". In: *arXiv preprint arXiv:2101.05844* (2021).

[WK18] 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)*. 2018, pp. 5286–5295.

[Sin+19] 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)*. Vol. 3. POPL. 2019, pp. 1–30.

[Don+18] Yinpeng Dong, Fangzhou Liao, Tianyu Pang, Hang Su, Jun Zhu, Xiaolin Hu, and Jianguo Li. "Boosting Adversarial Attacks With Momentum". In: *Proceedings of the 2018 IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2018)*. Computer Vision Foundation / IEEE, 2018, pp. 9185–9193.

[Bun+20] 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". In: *Journal of Machine Learning Research* 21 (2020), pp. 1574–1612.

[Zha+18] 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)*. 2018, pp. 4944–4953.

[De +21] 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". In: *arXiv preprint arXiv:2104.06718* (2021).

[Mül+22] 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". In: *arXiv preprint arXiv:2212.10376* (2022).

[Sin+18] 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)*. 2018, pp. 1–12.

[Wan+18b] 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)*. USENIX Association, 2018, pp. 1599–1614.

[KBS22] Philipp Kern, Marko Kleine Büning, and Carsten Sinz. "Optimized Symbolic Interval Propagation for Neural Network Verification". In: *arXiv preprint arXiv:2212.08567* (2022).

[YS20] Li Yang and Abdallah Shami. "On hyperparameter optimization of machine learning algorithms: Theory and practice". In: *Neurocomputing* 415 (2020), pp. 295–316.

[Sch+22] Elias Schede, Jasmin Brandt, Alexander Tornede, Marcel Wever, Viktor Bengs, Eyke Hüllermeier, and Kevin Tierney. "A Survey of Methods for Automated Algorithm Configuration". In: *Journal of Artificial Intelligence Research* 75 (2022), pp. 425–487.

[HHL11] 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)*. Vol. 6683. Lecture Notes in Computer Science. Springer, 2011, pp. 507–523.

[Lin+22] Marius Lindauer, Katharina Eggensperger, Matthias Feurer, André Biedenkapp, Difan Deng, Carolin Benjamins, Tim Ruhkopf, René Sass, and Frank Hutter. "SMAC3: A Versatile Bayesian Optimization Package for Hyperparameter Optimization". In: *Journal of Machine Learning Research* 23 (2022), pp. 2475–2483.

[Bre01] Leo Breiman. “Random Forests”. In: *Machine Learning* 1 (2001), pp. 5–32.

[SLA12] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. “Practical Bayesian Optimization of Machine Learning Algorithms”. In: *Advances in Neural Information Processing Systems 25 (NeurIPS 2012)*. 2012, pp. 2960–2968.

[Hut+09] Frank Hutter, Holger H Hoos, Kevin Leyton–Brown, and Thomas Stützle. “ParamILS: An Automatic Algorithm Configuration Framework”. In: *Journal of Artificial Intelligence Research* 36 (2009), pp. 267–306.

[NFL18] Neil Newman, Alexandre Fréchette, and Kevin Leyton-Brown. “Deep optimization for spectrum repacking”. In: *Communications of the ACM* 61.1 (2018), pp. 97–104.

[XHL10] 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)*. AAAI Press, 2010, pp. 210–216.

[BLZ+23] Junjie Bai, Fang Lu, Ke Zhang, et al. *ONNX: Open Neural Network Exchange*. `https://github.com/onnx/onnx`. 2023.

[Gui+23] Dario Guidotti, Stefano Demarchi, Armando Tacchella, and Luca Pulina. *The Verification of Neural Networks Library (VNN-LIB)*. `https://www.vnnlib.org`. 2023.

[BFT16] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The Satisfiability Modulo Theories Library (SMT-LIB)*. `www.SMT-LIB.org`. 2016.

[Sco+22] Joseph Scott, Guanting Pan, Elias B. Khalil, and Vijay Ganesh. “Goose: A Meta-Solver for Deep Neural Network Verification”. In: *Proceedings of the 20th Internal Workshop on Satisfiability Modulo Theories (SMT 2022)*. Vol. 3185. CEUR Workshop Proceedings. CEUR-WS.org, 2022, pp. 99–113.

[Lin+19] Marius Lindauer, Katharina Eggensperger, Matthias Feurer, André Biedenkapp, Joshua Marben, Philipp Müller, and Frank Hutter. “BOAH: A tool suite for multi-fidelity bayesian optimization & analysis of hyperparameters”. In: *arXiv preprint arXiv:1908.06756* (2019).

[Xu+11] Lin Xu, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. “Hydra-MIP: Automated Algorithm Configuration and Selection for Mixed Integer Programming”. In: *Proceedings of the 18th RCRA Workshop on Experimental evaluation of Algorithms for Solving Problems with Combinatorial Explosion (RCRA 2011)*. 2011, pp. 16–30.

[Zha+22] Huan Zhang, Shiqi Wang, Kaidi Xu, Linyi Li, Bo Li, Suman Jana, Cho-Jui Hsieh, and J Zico Kolter. “General cutting planes for bound-propagation-based neural network verification”. In: *Advances in Neural Information Processing Systems 35 (NeurIPS 2022)*. 2022, pp. 1656–1670.

[Tra+19] Hoang-Dung Tran, Diago Manzanas Lopez, Patrick Musau, Xiaodong Yang, Luan Viet Nguyen, Weiming Xiang, and Taylor T. Johnson. “Star-Based Reachability Analysis of Deep Neural Networks”. In: *Proceedings of 3rd World Congress on Formal Methods (FM 2019)*. Vol. 11800. Lecture Notes in Computer Science. Springer, 2019, pp. 670–686.

[Tra+20b] Hoang-Dung Tran, Stanley Bak, Weiming Xiang, and Taylor T. Johnson. "Verification of Deep Convolutional Neural Networks Using ImageStars". In: *Proceedings of the 32nd International Conference on Computer Aided Verification (CAV 2020)*. Vol. 12224. Lecture Notes in Computer Science. Springer, 2020, pp. 18–42.

[Bak+20] 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)*. 2020, pp. 66–96.

[Pal+21c] 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". In: *arXiv preprint arXiv:2104.06718* (2021).

[Den12] Li Deng. "The MNIST Database of Handwritten Digit Images for Machine Learning Research". In: *IEEE Signal Processing Magazine* 29.6 (2012), pp. 141–142.

[KH09] Alex Krizhevsky and Geoffrey Hinton. "Learning multiple layers of features from tiny images". MA thesis. University of Toronto, 2009.

[FKS22] James Ferlez, Haitham Khedr, and Yasser Shoukry. "Fast BATLLNN: Fast Box Analysis of Two-Level Lattice Neural Networks". In: *25th ACM International Conference on Hybrid Systems: Computation and Control (HSCC 2022)*. ACM, 2022, pp. 1–11.

[Bri+23] Christopher Brix, Mark Niklas Müller, Stanley Bak, Taylor T. Johnson, and Changliu Liu. "First three years of the international verification of neural networks competition (VNN-COMP)". In: *International Journal on Software Tools for Technology Transfer* 25.3 (2023), pp. 329–339.

[Ker+19] Pascal Kerschke, Holger H. Hoos, Frank Neumann, and Heike Trautmann. "Automated Algorithm Selection: Survey and Perspectives". In: *IEEE Transactions on Evolutionary Computation* 27.1 (2019), pp. 3–45.

[Xu+08] Lin Xu, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. "SATzilla: Portfolio-based Algorithm Selection for SAT". In: *Journal of Artificial Intelligence Research* 32 (2008), pp. 565–606.

Appendix A

Configuration Spaces

Table A.1: ConfigurationSpace used for α,β-CROWN

Hyperparameter	Values
complete verifier	{bab, mip, bab-refine, skip}
enable incomplete verification	{true, false}
loss reduction func	{sum, min, max}
bound prop method	{alpha-crown, crown, forward, forward+crown, alpha-forward, init-crown}
bab branching method	{kfsb, babsr, fsb, kfsb-intercept-only}
bab branching reduceop	{min, max}
bab branching input split enable	{true, false}
bab branching input split enhanced bound prop method	{alpha-crown, crown, forward+crown, crown-ibp}
bab branching input split enhanced branching method	{naive, sb}
bab branching input split enhanced bound patience	[10, 120]
bab branching input split attack patience	[10, 120]
pgd attack order	{before, middle, after, skip}
enable mip attack	{true, false}
attack mode	{diversed_PGD, diversed_GAMA_PGD, PGD, boundary}

Table A.2: ConfigurationSpace used for nnenum

Hyperparameter	Values
single set	{true, false}
compress init box	{true, false}
eager bounds	{true, false}
contract zonotope	{true, false}
contract zonotope lp	{true, false}
contract lp optimized	{true, false}
overapprox near root max split	[1, 5]
overapprox gen limit multiplier	(0.0, 3.0)
inf overapprox min gen limit	{true, false}
overapprox min gen limit	(1, 100)
inf overapprox lp timeout	{true, false}
overapprox lp timeout	(0.0, 5.0)
overapprox both bounds	{true, false}
branch mode	{overapprox, ego, ego_light, exact}
try quick overapprox	{true, false}
split order	{largest, one_norm, smallest, inorder}
offload closes to root	{true, false}
split tolerance	(1e-9, 1e-7)
split if idle	{true, false}
glpk timeout	(10, 120)
glpk first primal	{true, false}
glpk reset before minimize	{true, false}
skip compressed check	{true, false}
skip constraint normalization	{true, false}

Table A.3: ConfigurationSpace used for VeriNet

Hyperparameter	Values
precision	{32, 64}
queue depth	(1, 10)
max children suspend time	(300, 900)
max accepted memory increase	(10, 30)
use one shot attempt	{true, false}
use pre processing attack	{true, false}
max estimated mem usage	$(64 * 10^8, 64 * 10^{11})$
optimised relu relaxtion max bounds multiplier	(1, 3)
use ssip	{true, false}
store ssip bounds	{true, false}
input node split	{true, false}
hidden node split	{true, false}
indirect hidden multiplier	(0.5, 1.0)
indirect input multiplier	(0.5, 1.0)
use bias separated constraints	{true, false}
perform lp maximisation	{true, false}
use optimised relaxation constraints	{true, false}
use optimised relaxation split heuristic	{true, false}
use simple lp	{true, false}
num iter optimised relaxations	(1, 5)
use lp presolve	{0, 1}
gradient descent interval	(1, 5)
gradient descent max iters	(1, 10)
gradient descent step	(0.01, 0.2)
gradient descent min loss change	(0.001, 0.1)

Table A.4: ConfigSpace used for Oval-BaB

Hyperparameter	Values
n1 bounding algorithm	{propagation}
n1 nb steps	(1, 10)
n1 initial step size	(0.1, 10.0)
n1 step size decay	(0.9, 1.0)
n1 joint ib	{true, false}
n1 type	{alpha-crown, beta-crown, gamma-crown}
n1 auto iters	{true, false}
n2 bounding algorithm	{dual-anderson}
n2 bigm	{init}
n2 cut	{only}
n2 bigm algorithm	{adam}
n2 nb iter	(200, 1000)
n2 cut frequency	(150, 750)
n2 max cuts	(2, 22)
n2 cut add	(1, 4)
n2 initial step size	(0.001, 0.1)
n2 final step size	(0.000001, 0.001)
n2 nb_outer_iter	(500, 1500)
n2 larger irl if naive init	{true, false}
n2 restrict factor	(0.5, 3.0)
n2 auto iters	{true, false}
n2 hard overhead	(1, 20)
bounding do ubs	{true, false}
bounding parent init	{true, false}
ibs use lb	{true, false}
ibs tight ib	{null}
ibs fixed ib	{true, false}
ibs joint ib	{true, false}
ub method	{mi_fgsm}
ub iters	(250, 750)
ub lr tensor	{true, false}
ub num adv ex	(50, 150)
ub check adv	(1, 3)
ub mu tensor	{true, false}
ub decay tensor	{true, false}
branching heuristic type	{FSB, SR}
branching max domains	(25000, 75000)
branching bounding algorithm	{propagation}
branching best among	{KW, crown, naive}
branching bounding type	{best_prop, crown}