Universiteit
Leiden
The Netherlands

Automated machine learning for neural network verification

König, H.M.T.

Citation
König, H. M. T. (2025, October 9). *Automated machine learning for neural network verification*. Retrieved from https://hdl.handle.net/1887/4266921

Version:	Publisher's Version
License:	Licence agreement concerning inclusion of doctoral thesis in the Institutional Repository of the University of Leiden
Downloaded from:	https://hdl.handle.net/1887/4266921

Note: To cite this publication please use the final published version (if applicable).

Chapter 4

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4.1 Background

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

4.1.1 MIP-Based Neural Network Verification

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

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

subject to

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

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

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

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

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

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

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

4.1.2 Automated Configuration of MIP Solvers

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

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

4.1.3 Automatic Portfolio Construction

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

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

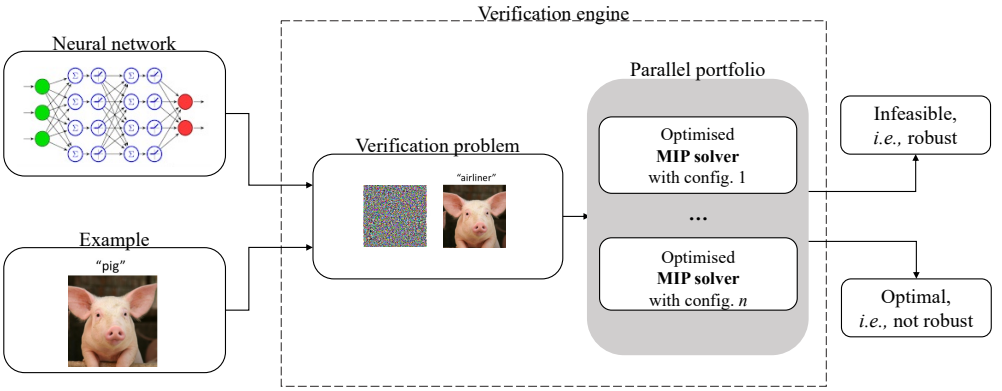
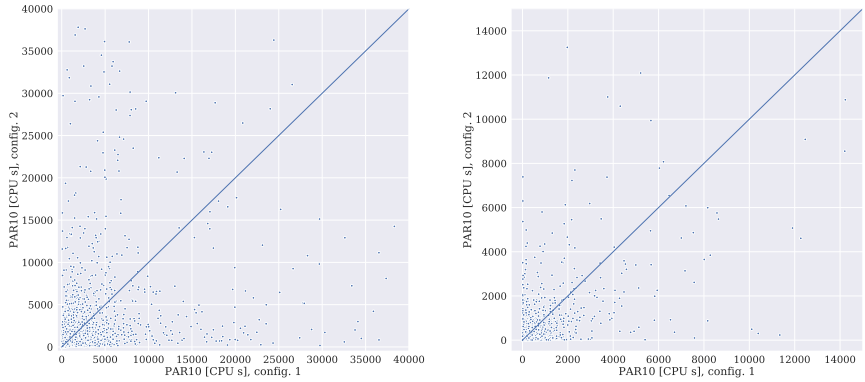


Figure 4.1: Schematic diagram of the proposed framework.

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

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

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



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

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

4.2 Network Verification with Parallel MIP Solver Portfolios

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

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

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

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

4.2.1 Configuration Procedure

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

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

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

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

4.2.2 MIP Solver

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

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

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

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

4.3 Setup of Experiments

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

environment in which experiments were carried out.

4.3.1 Configuration Objective

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

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

4.3.2 Details of the Configuration Procedure

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

4.3.3 Data

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

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

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

4.3.4 Execution Environment and Software Used

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

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

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

solver version 9.0.1.

4.4 Results

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

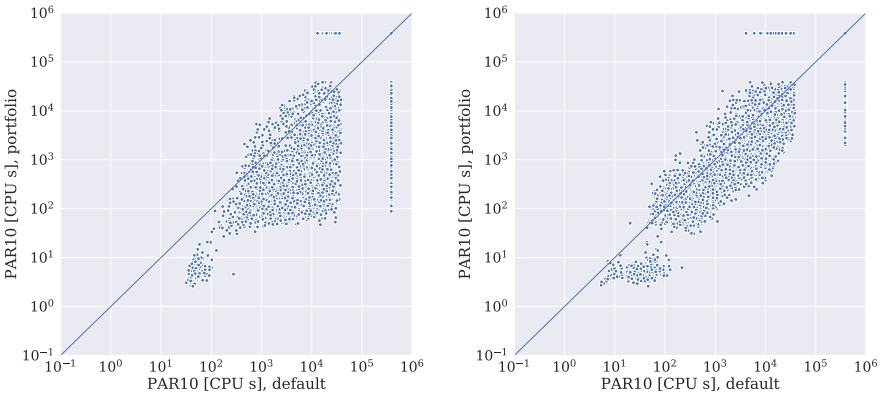
4.4.1 MIPVerify

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

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

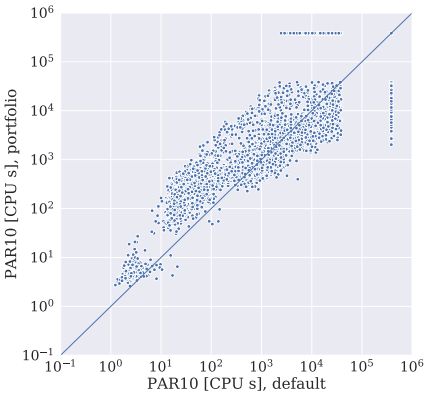
The default configuration of Gurobi running on 4 cores was also clearly outperformed by our portfolio in terms of CPU time (8 478 *vs* 22 065 CPU seconds). Furthermore, the portfolio was able to reduce the number of timeouts (14.96% *vs* 17.74%), while improving on the upper bound (23.86% *vs* 27.49%).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

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

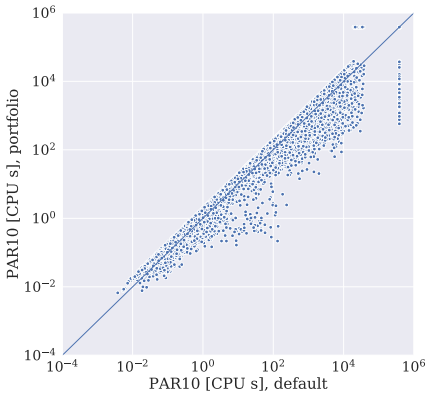


(a)
$\mathrm{SDP_dMLP_A}$
Cores = 32

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



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



(d)
mnistnet
Cores = 1

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

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

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

4.4.2 Venus

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

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

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

utilised by the portfolio.

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

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

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

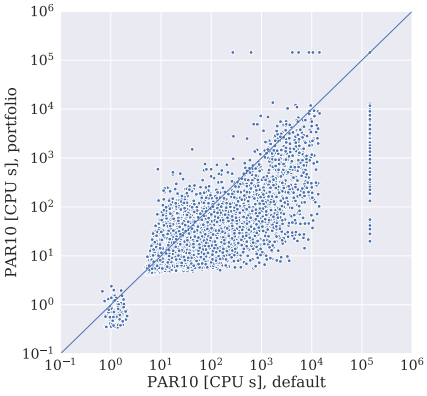
On the $\mathrm{SDP_dMLP_A}$ benchmark, the default baseline, *i.e.*, Venus with default settings, was outperformed by the portfolio in terms of PAR10 (636 *vs* 6 534 CPU seconds) as well as the fraction of timeouts (6.10% *vs* 9.76%). In this setup, the default timed out on 379 instances solved by the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

4.5 Conclusions and Future Work

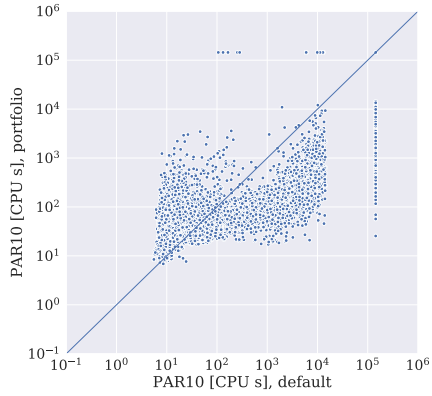
In this study, we have demonstrated the effectiveness of automated algorithm configuration and portfolio construction in the context of neural network verification, thereby providing an answer to the second research question (RQ2) of whether we can improve the performance of a MIP-based verification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

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

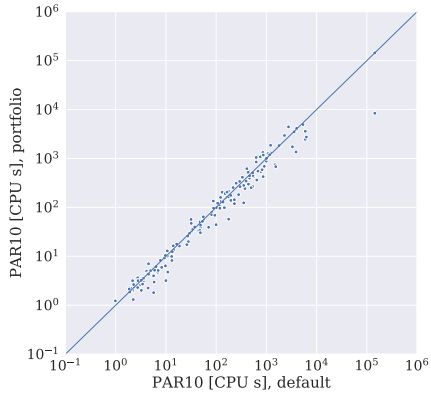
Our results show that the portfolio approach can significantly reduce the CPU time required by these systems on various verification benchmarks, while reducing the



(a) mnistnet | Default*



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



(c) ACAS Xu | Default*

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

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

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

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

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

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

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