Universiteit Leiden
The Netherlands

Opleiding Informatica

Investigating the effectiveness of complementary NNENUM portfolios for neural network verification using Auto-Verify

Tristan C. Cotino

Supervisors:
Jan N. van Rijn, Annelot W. Bosman

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

02/06/2025

Abstract

Due to their versatility and efficiency, machine learning algorithms have become more important in the field of computer science in recent years. However, evaluating their performance and reliability can be challenging due to their black box nature. This is especially true for difficult to interpret models such as neural networks. Specifically, these neural networks may be vulnerable to adversarial attacks. In these attacks, seemingly insignificant alterations to the input can cause incorrect predictions. With neural network verification, we can assess the robustness of a neural network against such an attack. This thesis investigates the effectiveness of neural network verification using Auto-Verify, a tool that can utilize a portfolio of various neural network verification tools to improve efficiency. By using a training set to measure the performance of a verification tool in a specific configuration, efficiency can be improved further. By doing this, Auto-Verify should be able to construct portfolios of verification tools where the strengths of the configurations of each verification tool complement each other. The software offers an interface to configure and run various verification tools in parallel. One of these verification tools is NNENUM, a CPU-based verification tool. Although Auto-Verify uses portfolio construction to improve efficiency, prior work suggests that there are still improvements to be made. Our research will focus on analyzing the performance of a portfolio consisting of complementary configurations of NNENUM. We focus on NNENUM to simplify the experimental setup and to run the experiments on a larger scale because it is CPU-based. With our experiments, we aim to determine to what extent Auto-Verify is able to improve the efficiency of portfolio construction using only configurations of NNENUM. To do this, we will analyze the impact of various parameters that Auto-Verify uses to construct a portfolio. These include the number of configurations of verification tools that should end up in the portfolio, the number of problem instances that are used during the tuning process, and the number of neural networks that are taken into account during the tuning process. These experiments are all run on a dataset of MNIST images, and we use the wall-clock time to evaluate the performance. In the process of setting up these experiments, we improved the quality of the code by patching several mistakes in the code that could have prevented the portfolio-building process. Although we did find a major bug that prevented Auto-Verify from using portfolios with the correct configurations, all results indicate that Auto-Verify is not able to take advantage of portfolios with complementary configurations of verification tools for the tested verification tool NNENUM.

Contents

1 Introduction

Machine learning algorithms have become increasingly important in the field of computer science in recent years. These algorithms implement methods that enable systems to learn from data without the input of a programmer. The application of these algorithms allows the extraction of patterns from large amounts of complex data and can be applied in numerous fields, such as healthcare [Cha+17], finance [Wei19], and engineering [KR24]. Training such an algorithm is relatively simple, but assessing the performance and reliability of machine learning algorithms can be challenging. One prominent class of these machine learning algorithms is neural networks [LBH15]. These neural networks consist of interconnected layers of nodes (neurons) that learn how to process input data. Analyzing this class of machine learning algorithms can also be challenging. Training such a neural network and using it to perform predictive tasks might give good results, but they operate as a black box. This means that it is difficult to understand how it makes their decisions. This makes it possible to perturb the input to a neural network to get a different prediction. These perturbations can be so small that a human might not be able to make a distinction between them, but the neural network does. These perturbations are called *adversarial attacks*. There are specific methods to determine whether a neural network is robust enough against such an attack, called neural network verification.

The program Auto-Verify [Spe24] is a tool to perform neural network verification with different verification tools. It is essentially an interface to install, use, and configure parameters of different algorithms to perform neural network verification. These algorithms include the CPU-based verification algorithm NNENUM [Bak21] and multiple GPU-based verification algorithms such as VeriNet [HL20; HL21] and AB-Crown [Wan+21]. It also implements the concept of constructing a portfolio [KHR24] of algorithms in different configurations to improve verification efficiency. This not only provides the option of utilizing the strengths of different algorithms in parallel but also to utilize the strengths of differently configured algorithms in parallel. An algorithm with a specific configuration might perform well in some problem instances, but this often comes at the cost of performance loss on other problem instances. This concept of using multiple complementary configurations of the same algorithm is utilized by using them in a solver portfolio [KHR24]. Auto-Verify can optimize the configuration of a verification algorithm with the hyperparameter optimization method Hydra [Lin10]. This is achieved by using a set of input problem instances to find a configuration that is more efficient in solving these problem instances.

However, Auto-Verify is not performing as well as theorized [Spe24]. Some algorithms produce errors and the performance improvements reported by Spek [Spe24] from constructing a portfolio of algorithms are not as high as previous work [KHR24].

In this thesis, we will analyze the performance of a portfolio of different NNENUM configurations. The focus lies on specifically the NNENUM algorithm because it is CPU-based, which simplifies the experiment setup and hardware requirements.

This research aims to formulate an answer to the following research question: To what extent can Auto-Verify take advantage of the benefits of a portfolio of the NNENUM neural network verification algorithm?

The neural network verification process has many variables that can have an impact on its performance. To guide us through this process, the research will answer the following sub-questions:

1. How do the parameters of Auto-Verify influence the effectiveness of its neural network verification?
2. What is the performance gain in terms of CPU-time of using a portfolio of NNENUM configurations over using a singular configuration of NNENUM?

In this thesis we will answer these questions by first covering background information about neural network verification and Auto-Verify. Then we will discuss the implementation of Auto-Verify and describe how we have set up experiments. After this we can answer the research questions and give recommendations for further work on Auto-Verify.

2 Background and Related work

We will first describe the workings of neural networks. There are various types of neural networks, but for this research, we will focus on the basic concepts. In contrast to conventional algorithms, machine learning models are of a black-box nature. Once such an algorithm has been implemented, it does not need any more adjustments from a programmer to learn from input data. These networks consist of layers of nodes, and the connections between these layers have specific weights. During the training process, the input data is processed and adjustments are made to the weights in the neural network. This allows the neural network to detect patterns in the data and make predictions on data that falls outside the training dataset. A visualization of the structure of such a network can be found in Figure 1.

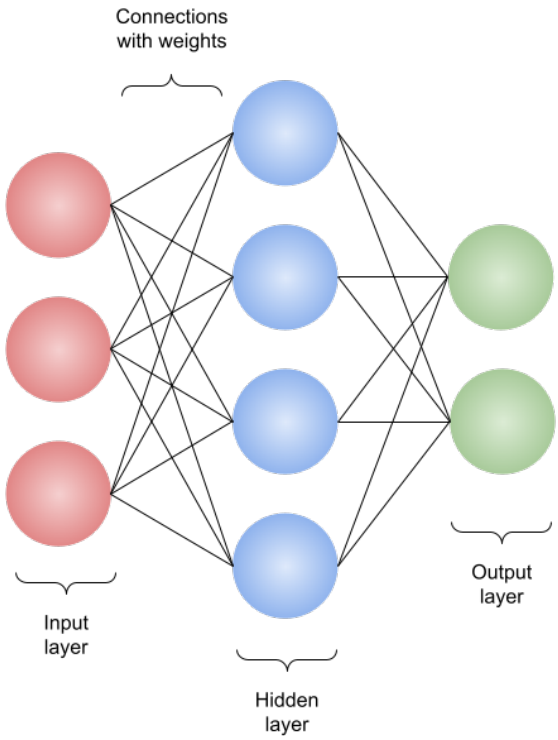


Figure 1: A simplified visualization of a neural network showing different types of layers

2.1 Adversarial attacks

It is necessary to evaluate the reliability of a neural network to ensure that it provides correct predictions. After training a neural network to make predictions on a dataset it is important to benchmark the neural network to verify if it can correctly predict all outcomes. For example, a neural network used to do image classification needs to be tested with each type of classification that it should be able to predict. This ensures that it is able to make every type of prediction, but there might also be cases where it produces unexpected predictions.
While it is reasonable to say that a network trained for image classification may correctly interpret an image, we cannot say the same if we make small alterations to the image. If a seemingly insignificant

alteration to the image is made, the neural network should not change the interpretation. But if it does change its interpretation, the neural network might have a robustness problem.
This fact can be used to intentionally change the prediction outcome with specific perturbations. If it is discovered at which point a neural network changes its decisions, the amount of perturbations can be minimized. By doing this you can make the perturbations look insignificant. For instance, take a neural network used to perform image recognition. By determining which specific series of pixels need to be changed to make the neural network classify the image differently, the neural network can be forced to output any classification. While this does involve altering the input, you might not need to alter the image drastically. These perturbations with the goal of changing the output of a neural network are called adversarial attacks.
One type of adversarial attack is a Fast Gradient Sign Method (FGSM) [GSS15]. This method calculates how much the outcome changes for each of the input nodes when altering the values. This is then used to change each of the inputs just enough to make sure that the neural network changes its decision. For image recognition, this might just look like the image has some added noise, while the image looks nearly identical. Other methods, such as a Jacobian-based Saliency Map Attack (JSMA) [Pap+16] focus on changing the lowest number of input nodes. This means that it might be more obvious to human inspection, but is still able to influence the decision process. These types of attacks are visualized in Figure 2.

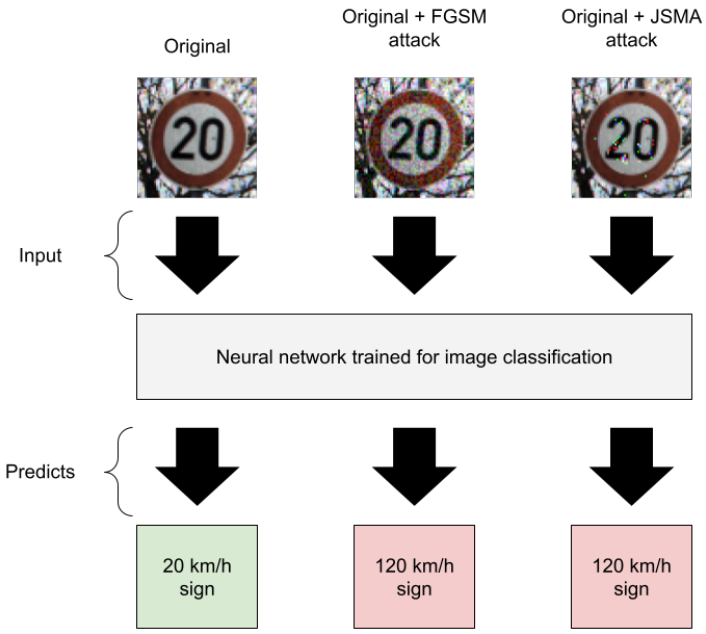


Figure 2: A visualization of how adversarial attacks can change the prediction of a neural network. Source: https://kennysong.github.io/adversarial.js/

2.2 Robustness verification

There are methods to determine if a neural network is robust enough to withstand an adversarial attack. To do this, neural networks can be analyzed with robustness verification techniques [SN20]. These neural network verification techniques can provide a mathematical proof of whether specific conditions hold. Based on this proof we can say if a neural network is prone to adversarial attacks [Liu+21].
This proof can be accomplished by proving that for every possible input problem instance that can be used for a neural network, another input problem instance within a certain radius will result into the same output. Given a neural network with k_0-dimensional input, k_m-dimensional output, input $x \in D_x \subseteq \mathbb{R}^{k_0}$ and output $y \in D_y \subseteq \mathbb{R}^{k_m}$, where D_x and D_y are the domains of possible values for x and y. By defining the neural network like this, you can have it represent a function f, as $f(x) = y$. To complete a proof for a verification problem, we will have to determine if the input-output relationships of f hold. We do this by verifying if input problem instances x_0 that fall within range of a predefined radius of perturbations ϵ measured from x result in the same output class. The problem can be formulated as follows:

$$\forall x : ||x - x_0||_\infty < \epsilon \Rightarrow f(x) = f(x_0)$$

Robustness verification techniques are characterized by soundness and completeness. Formal verifications can be characterized as sound if it only states that a property holds, if it actually holds. They can be characterized as complete if it correctly states that a property holds, whenever it holds.

Verification algorithms can produce three different results to a verification query: satisfiable, unsatisfiable and unknown. If a property is unsatisfiable, there does not exist an adversarial perturbation that causes a misclassification. If the result is satisfiable, it means that there exists an adversarial perturbation that causes a misclassification. Sound algorithms may produce the result "unknown" while complete algorithms are guaranteed to state if a property holds with enough time and resources. An additional result is usually added to verification algorithms because verification techniques can take a long time to run. This is the *timeout* result, which is a predetermined amount of time that is allowed to be spent on the verification query before the process is stopped.
One example of a robustness verification tools is the Neural Network Enumeration Tool (NNENUM) [Bak21]. This tool uses optimized abstraction refinement to analyze the behavior of networks. It is particularly efficient at verifying ReLU neural networks and relies on utilizing CPU power rather than GPU power.

2.3 Hyperparameter optimization

For these verification tools, optimizing their configurations can improve performance. These tools have a set of hyperparameters that can be configured to change how they perform. In the case of NNENUM, some hyperparameters affect the branching mode and approximation methods.
There are tools that can optimize the configurations of verification tools. SMAC [Lin+22] is a *sequential model-based optimization* (SMBO) approach that can optimize the configuration of an algorithm or tool for a given set of problem instances. By measuring the performance of a verification tool with different configurations, it can approximate how the verification tool should

be configured to optimize its performance on these problem instances. If these problem instances are well-selected, it results in performance increases outside the training set.

2.4 Portfolio-based Neural Network Verification

Every neural network verification tool with an optimized configuration has its strengths and weaknesses. Hyperparameters can be adjusted to improve the performance for specific problem instances, but it can also come at the cost of performance over other problem instances. When analyzing the performance of neural network verification tools, we can find evidence for strong performance complementarity over differently configured verification tools [KHR24]. This means that performance improvement over a subset of problem instances might be significant, but comes at the cost of performance in other problem instances.
We can take advantage of this by using multiple instances of a verification tool in different configurations to improve verification efficiency [KHR24]. While it does come at the cost of dividing computing power over multiple configurations of a verification tool, the performance gains outweigh the negatives of the division of resources.
Hydra [Lin10] is a method for portfolio construction. It is a greedy algorithm that chooses the next algorithm and algorithm configuration based on its performance. Let P_i be the portfolio after iteration i, with $P_0 := \{\}$ being an empty portfolio. After iteration 1 of the algorithm, we obtain the first configuration θ_1 using a configurator like SMAC and we get the portfolio $P_1 := \{\theta_1\}$. In the next iteration we measure the performance of the portfolio with an added configuration. If this measured performance is worse than before we added the configuration, we keep the old portfolio. If the measured performance is better than before we added the configuration, we keep the new configuration in the portfolio. By doing this, the performance of the portfolio is evaluated in each iteration and the portfolio is updated with improved performance. This results in a new portfolio $P_i = P_{i-1} \cup \{\theta_1, ..., \theta_n\}$.
Portfolio construction can be used with multiple configurations of the same verification tool, but also in combination with different algorithms. By doing this, you can take advantage of the strengths of specific verification tools. However, we will focus only on using one type of verification tool in a portfolio in this research.

2.5 Auto-verify

The software package Auto-Verify aims to provide a way to perform neural network verification with different tools [Spe24]. The Python library provides an interface to install, configure, and run different verification tools. Together with an implementation of Hydra and usage of a SMAC library, it makes it possible to create portfolios of verification tools with optimized configurations. This makes it a complete package to perform experiments with neural network verification tools, but also with a portfolio of tools.
Auto-Verify contains interfaces to several verification tools. These include interfaces to the CPU-based verification tool NNENUM [Bak21] and the GPU-based verification tools α, β-CROWN [Wan+21], VeriNet [HL20; HL21] and OvalBab [Bun+20; Pal+21]. The software can use different configurations for all of these tools and can also use them in a portfolio.
Aside from making these tools more accessible, it provides interfaces to work with data that is used in the VNNCOMP benchmarks [Bri+24]. This competition is an annual event that provides

a variety of datasets to benchmark the performance of verification tools. The structure of these benchmarks is standardized, which simplifies the benchmarking process. Each benchmark contains a series of problem instances. These problem instances comprises an ONNX file, containing data about the neural network structure [Zha+23], and a VNNLIB file with data about the network nodes [Dem+23]. The number of neural network structures present over all problem instances in each benchmark may vary. This varying number of neural network structures will also be something we will be looking into in this research. Additionally, there is a specified timeout in each problem instance indicating the maximum amount of wall-clock time a verification tool is allowed to spend on the problem instance.

While Auto-Verify provides an accessible framework for neural network verification, it still needs work and research to improve its performance. While there is evidence that it achieves performance gains when using multiple verification algorithms in a portfolio [Spe24], it does not match the theorized performance improvements [KHR24].

3 Tool and Implementation

The Auto-Verify software package offers a comprehensive solution for performing neural network verification using various verification tools. It is written in Python and uses many modern Python libraries to improve the developer experience such as tox[1], several continuous integration tools, and automated documentation generation.

3.1 Analysis of opportunities

There are a few reasons why the performance of Auto-Verify might not match the theorized performance increase that solver portfolios should make possible. These can be divided into technical opportunities that need software engineering work to be analyzed, and opportunities that require experimentation to see how the software can be used optimally.

3.1.1 Technical opportunities

On the technical side of the implementation of Auto-Verify, the framework may limit the performance of the software. The library makes use of virtual environments to run each verification tool in a portfolio. This extra layer of abstraction might hurt the performance as more resources are used that could have been used for the verification process.
Additionally, the resource division strategy between verification tools can be improved. Auto-Verify distributes the amount of resources evenly across all verification tools in the portfolio. While this can improve performance, it may not be optimal. Some tools might be better at solving a specific type of problem instance while using fewer resources. Optimizing the resource allocation strategy is something that has not yet been explored.

3.1.2 Experimental opportunities

To start tuning a portfolio for neural network verification with Auto-Verify, some parameters can be set that change the tuning process and have an impact on the performance of a portfolio. These parameters include the available verification tools, the number of network types used, and the number of example problem instances used by Hydra in the tuning process. Although other parameters exist, this research focuses on these specific ones.
Auto-Verify provides interfaces to a multitude of verification tools, but it is still unknown if it can use multiple configurations of the same tool in different configurations to take advantage of portfolio-based verification. While there are noticeable performance gains when using different tools, there is no evidence that the complementary configurations of the same verification tool would improve performance.
Having multiple network structures present in the training problem instance during the tuning process can also affect performance. While the structures of several networks might be similar, they might behave differently when analyzed. If the portfolio tuner needs to take large differences between the training problem instances into account, it may cause performance issues.

[1] https://tox.wiki/

The number of problem instances used during the tuning process significantly impacts the performance of the portfolio beyond the training set. Using a few problem instances might result in overfitting, even though the portfolio becomes efficient at solving those problem instances. Conversely, using many problem instances prevent overfitting, but requires more tuning time.

3.2 Resolution of dependency issues

Dependency issues are usually one of the first issues that will arise when assessing a software package. To see if a package is still behaving as intended, the first thing to try might be running the test suite. For Auto-Verify, this test suite failed without being able to run the actual tests. The issue stemmed from the `pytest-lazy-fixture` dependency, which required freezing at version 0.6.3. Additionally, the `pytest` package itself needed freezing at version 7.3.2 for compatibility. After doing this, all the functionalities of the test suite have been restored. However, this is a temporary fix because the newest version of `pytest` package is already a major version ahead. If Auto-Verify is to be developed further, replacing `pytest-lazy-fixture` or rewriting its functionalities in Auto-Verify's code base is something to take into consideration.

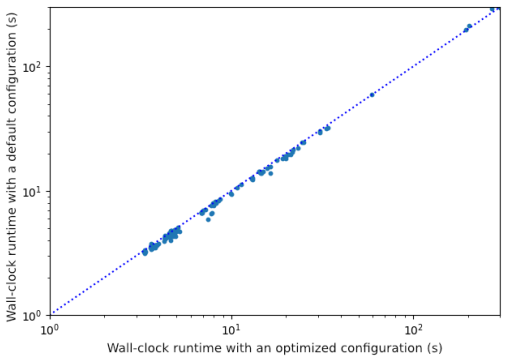
3.3 Impact of External Verification Tool Usage

Auto-Verify utilizes optimization techniques to improve the performance of neural network verification tools, but its effectiveness relies on the performance of external verification tools. If a tool malfunctions or yields poor results, Auto-Verify cannot optimize their configurations effectively. Different verification tools may crash more often than others, which might cause problems during the tuning process. Deciding how encountered errors during a verification process should be handled can be a difficult choice. Some tools may also not be suited to solve certain verification problems, which also makes it hard to compare performance improvements when constructing a portfolio. While these errors may not be fatal to the tuning process, it does make the tuning process less efficient.

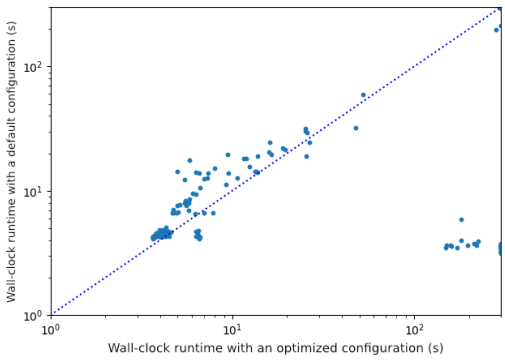
These errors that external verification tools may cause are unfortunately hard to take into account. Auto-Verify has no role in the development of these tools, and it is a difficult task to determine how it should run the verification tools to minimize the negative effects on portfolio creation of these errors. One solution could be to investigate if there are ways to analyze if certain hyperparameters cause more errors during the tuning process. However, investigating processes within Auto-Verify to identify hyperparameter configurations that increase error rates falls outside the scope of this research.

3.4 Resolution of portfolio verification bug

While testing Auto-Verify for bugs, it became apparent that using the verification tools in different configurations was not functioning properly. In Figure 3a, we tested an instance of α, β-CROWN by optimizing its configuration with problem instances of neural networks for the CIFAR dataset [Kri09]. After doing this, we see that it performs almost identically to an instance of α, β-CROWN with an unmodified default configuration. This observation is supported by the fact that all results lie around the diagonal of Figure 3a. This suggests that Auto-Verify does not parse or apply configurations correctly to the verification tools.



(a) Performance before bug fix



(b) Performance after bugfix

Figure 3: Performance of AB-Crown on the VNNCOMP CIFAR dataset

When investigating this behavior, we discover that this is caused by a bug in the code. The bug occurs during runtime when Auto-Verify submits jobs to a task scheduler for parallel execution of verification tools. The specific configuration for each tool is not correctly passed to the scheduler, resulting in the default configuration being used. Adding line 17 of Listing 1 to the code resolves this issue by ensuring that the correct configurations are passed to the verification tools.

```
1  def verify_instances(
2      self,
3      instances: Iterable[VerificationInstance],
4      *,
5      out_csv: Path | None = None,
6      vnncompat: bool = False,
7      benchmark: str | None = None,
8      verifier_kwargs: dict[str, dict[str, Any]] | None = None,
9      uses_simplified_network: Iterable[str] | None = None,
10 ) -> dict[VerificationInstance, VerificationDataResult]:
11
12     (...)
13
14     future = executor.submit(
15         self._verifiers[cv].verify_instance,
16         target_instance,
17         config=cv.configuration,
18     )
```

Listing 1: Code snippet of Auto-Verify where configurations of a portfolio are being parsed

After applying this bug fix, we can see that using a different configuration has an impact on the performance. In Figure 3b we can see that the configured instance of α, β-CROWN behaves differently in comparison to the instance with a default configuration. Whether the performance has increased or decreased is not relevant to this issue as we just need evidence that behavior changes when applying a different configuration.

3.5 Overview of contributions

To improve the Auto-Verify software package, the following contributions have been made to the code. These can all be found on its GitHub repository[2].

- **Resolution of a bug preventing portfolio verification**[3]
  There was a bug that caused Auto-Verify to be unable to pass the correct configuration of a verification tool to the task scheduler. Correctly adding a parameter to the task creation process fixed this issue.

- **Resolution of dependency issues**[4]
  The deprecated `pytest-lazy-fixture` library was incompatible with newer versions of the `pytest` library. This caused the test suite to fail and made it difficult to verify if the package was in a working state. Freezing the dependencies to a specific version fixed the issue.

- **Update the VeriNet installer**[5]
  The installation process of the VeriNet verification tool was malfunctioning. Updating a library for this tool was necessary to work with the VeriNet integration of Auto-Verify.

- **Documentation improvements**[6]
  Some parts of the documentation were not easy to understand as a new user. The documentation has been restructured and some parts were added.

- **Update formatting in full code base**[7]
  An update in formatting rules of the Continuous Integrations tests caused the pipeline to fail. This contribution made sure that all files followed the correct formatting rules.

[2] https://github.com/TrisCC/auto-verify-experiments
[3] https://github.com/ADA-research/auto-verify/pull/98
[4] https://github.com/ADA-research/auto-verify/pull/97
[5] https://github.com/ADA-research/auto-verify/pull/96
[6] https://github.com/ADA-research/auto-verify/pull/92
[7] https://github.com/ADA-research/auto-verify/pull/87

4 Experimental Setup

To investigate the effectiveness of Auto-Verify's portfolio creation, we set up a series of experiments where the parameters of the tuning process are varied. These experiments aim to determine how these parameters have an impact on the performance of the tuned portfolios.

4.1 Objective

Using multiple different configurations of the same verification tool can improve overall performance [KHR24]. Multiple configurations of the verification tool MIPVerify [TT19] have been used in a portfolio to improve overall performance. However, there is no evidence that this concept works when working with Auto-Verify.
It is uncertain if Auto-Verify is able to create a portfolio of complementary configurations for a single verification tool because of how Auto-Verify was used in previous work [Spe24]. While there is evidence of performance improvements when using different types of verification tools to construct a portfolio, there are no results that suggest that it can improve performance by creating a portfolio with multiple complementary configurations of the same verification tool.
To determine Auto-Verify's effectiveness at creating a portfolio with complementary configurations of a single verification tool we will be doing experiments with the verification tool NNENUM. This CPU-based verification tool requires less resources and crashes less often in comparison to the other supported verification tools. This will ensure that we can easily run a large number of experiments and the tuning process is not constrained by errors that might occur when solving verification problems.

4.2 Experiment parameters

There are three parameters of the portfolio tuning process that we are going to focus on in our experiments. These parameters have an impact on how Auto-Verify will be constructing the portfolios and should affect the performance of the resulting portfolios.

- **Number of NNENUM configurations**
  Before Auto-Verify starts the portolio construction process, the number of configurations that will end up in the portfolio needs to be specified. In these experiments, the resulting portfolios will consist of either one, two, or three configurations of NNENUM.

- **Number of verification problem instances**
  To start the automated tuning process, an input dataset is required to tune a configuration. The size of this dataset can be an important factor for the effectiveness of the resulting portfolio. In the experiments the number of verification problem instances are varied between 8 and 512.

- **Number of neural network structures**
  Having a portfolio that is created to solve problem instances for a specific neural network structure can be more efficient or needs less configuration time than a portfolio created to solve problem instances for multiple neural network structures. In the original VNNCOMP MNIST benchmark, three different neural network structures are being used. In these experiments

the number of neural network structures is reduced to one to observe if it has an impact on portfolio performance.

After creation of the portfolios with different input parameters, they are ran against a set of test problem instances. These test problem instances consists of a large number of problem instances that have also been verified using one instance of NNENUM with its default configuration. By doing this, we can compare the wall-clock time that each portfolio takes to solve a problem instance, and compare it to a default configuration. We also run NNENUM in its default configuration on the training set. This will give us an indication of the changes in performance caused by the tuning process.

4.3 Dataset

The dataset of networks that are used for all experiments is derived from a type of network that works with the MNIST dataset [Den12]. It is a well-known dataset and the problem instances generated with this dataset are relatively quick to solve. NNENUM performs well on this dataset, and it does not often run into crashes during the verification process. This makes it a suitable dataset to run a large number of experiments.
We need to create two different datasets to differentiate between the different numbers of neural networks used. The problem instances of the dataset with multiple neural networks follow the same timeouts as given in the VNNCOMP benchmarks. This is done to differentiate between the network types because less complex neural networks should be solved in a shorter amount of time. For the dataset with a single type of neural network, we increased the timeout of all problem instances to make sure that Hydra had enough time to tune the NNENUM configurations.

4.4 Hardware and software specifications

Both the tuning process portfolios and the testing of the portfolios on the datasets are run on nodes of a computing cluster. Each experiment run gets its own node with the following hardware available to it:

- Intel Xeon E5-2683 CPU, 2.10 GHz, 32 cores
- 64 GB RAM

We give Hydra 24 hours to train each portfolio on a single node with the latest version of NNENUM. The latest version of Auto-Verify is being used (0.1.3), together with solutions to the problems given in Section 3. Each node runs on CentOS Linux 7, and we use the Slurm workload manager to schedule the experiment tasks.
After training, we give the portfolios as much time as they need to solve enough problem instances to provide us with meaningful data. This data is presented in Section 5.

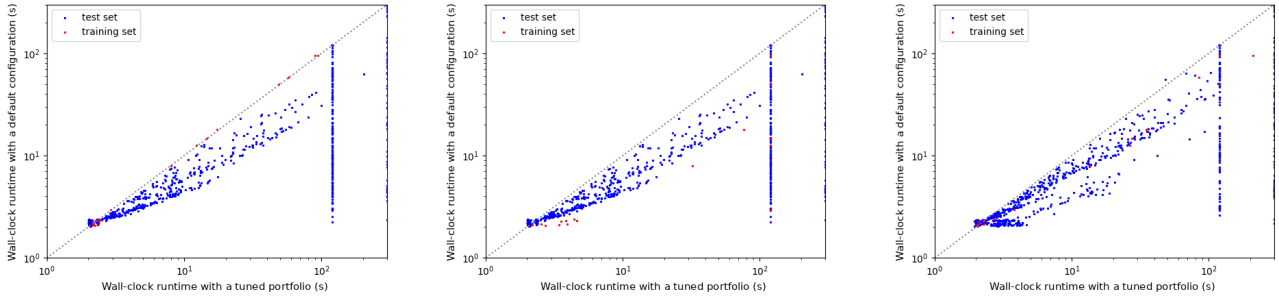
5 Results

As described in Section 4, each experiment first creates a configured portfolio where we alter some input parameter for the portfolio construction process. After doing this we can run the portfolio against the test dataset and analyze how it compares to a singular instance of NNENUM in its default configuration.
To do this we look at the wall-clock time that a portfolio needs to solve an MNIST problem instance and put the resulting data in a scatter plot. Each of these plots has been set up by putting the wall-clock time an experimental portfolio needs to solve a problem instance along the logarithmic x-axis and the wall-clock time a default configuration of NNENUM needs to solve a problem instance along the logarithmic y-axis. In these plots each blue dot represent a specific problem instance of the test set, and each red dot represents a specific problem instance of the training set. A diagonal line is also added to each plot to provide a clear view on how an NNENUM in its default configuration compares to a portfolio of NNENUM instances.
Not all experiments will be shown in this section, but the full set of experiments can be found in Appendix A. The scripts used to run these experiments, the constructed portfolios and results to the experiments can be found in the GitHub repository [8] of this project.

[8] https://github.com/TrisCC/auto-verify-experiments

5.1 Configuration tuning using different numbers of NNENUM configurations

In this set of experiments, we configure portfolios with different numbers of NNENUM configurations. We do this by specifying that Hydra should tune a specific number of NNENUM configurations to end up in the portfolio.
Figure 4 shows the outcomes of these experiments. Notably, almost all problem instances in the test set indicate that the tuned portfolio performs worse than NNENUM in its default configuration. We can see this by looking at the diagonal line in each of the graphs. If the majority of the results are below the diagonal of the graph, we can say that the constructed portfolio performs worse than NNENUM in its default configuration. The portfolios were not able to improve performance in the training set either. It is expected that at least some of the training problem instances have been solved quicker because they were used in the training process, but this does not seem to be the case. This might indicate that there is a problem with the tuning process. It is especially noteworthy to see that a singular tuned configuration of NNENUM performs worse than an instance of NNENUM with its default configuration. This implies that Auto-Verify actively makes the performance of a specific configuration worse instead of improving it.

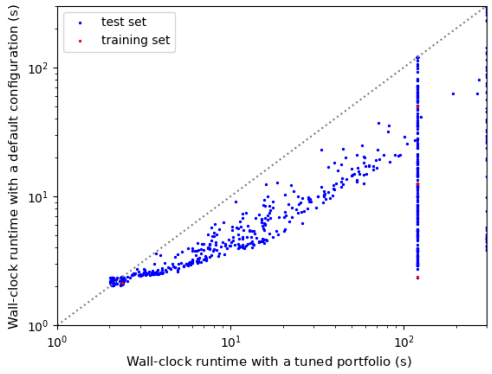


(a) One NNENUM configuration (b) Two NNENUM configurations (c) Three NNENUM configurations

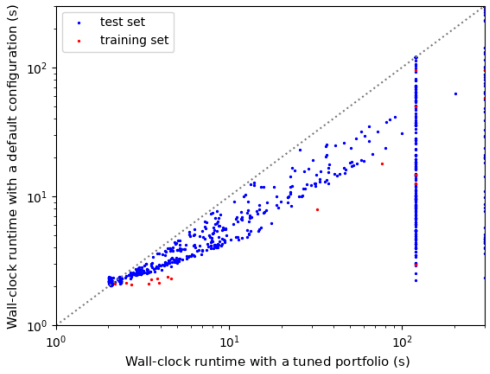
Figure 4: Performance of portfolios with different numbers of NNENUM configurations.

5.2 Configuration tuning using different sizes of training sets

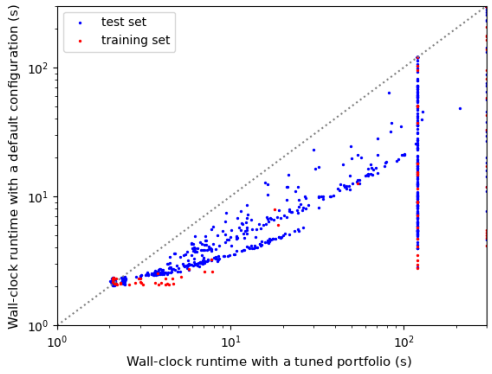
By using different sizes of training sets, there should be differences in performance from the configured portfolios. Given that all portfolios had the same amount of time to be tuned, the smaller training set sizes should have results that stand out more because the portfolios had more time to focus on each problem instance. The results of these experiments can be seen in Figure 5. Each of these portfolios was set to be configured with 2 configurations of NNENUM.
With the parameters that have been set, there should be differences in performance between using different training set sizes. However, the graphs do not suggest that any of the experiments were able to improve the performance of the portfolios. All graphs have results below the diagonal, which indicate that there were no performance improvements by using the portfolio. This is the case for both the results from the test set, and the training set. While the clusters of results are placed slightly different in each graph, these differences don't seem significant enough to conclude that there were any performance differences between the experiments. This can be caused by either Auto-Verify not being able to effectively construct portfolios with complementary configurations, or the tuning process being limited by the restricted tuning time.



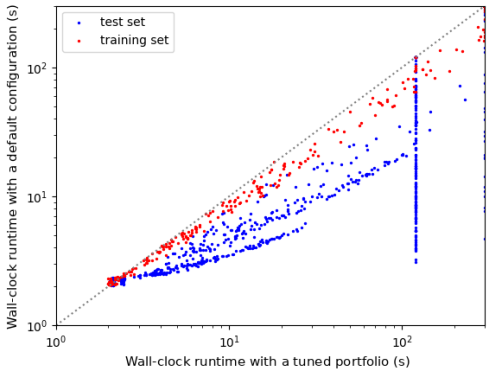
(a) 8 problem instances



(b) 32 problem instances



(c) 128 problem instances

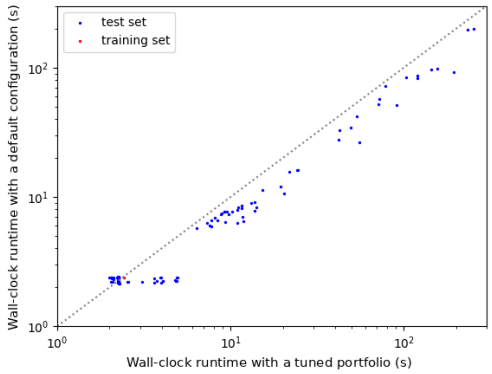


(d) 512 problem instances

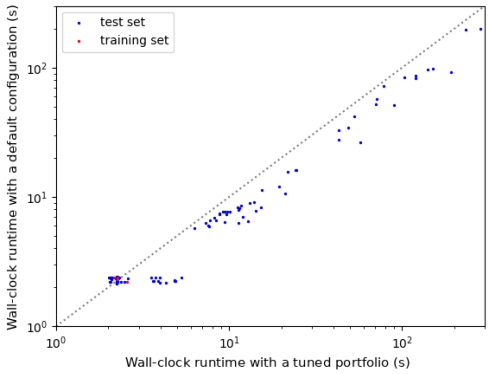
Figure 5: Performance of portfolios with different numbers of training problem instances.

5.3 Configuration tuning using a single neural network structure

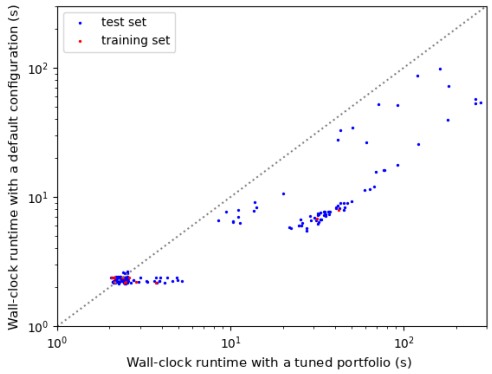
In this series of experiments, a single neural network structure has been used to see if this change has an impact on the performance of constructed portfolios. By having one neural network structure to work with, the tuning process should run more efficiently and create better performing portfolios. The results of these experiments can be found in Figure 6.
This set of results looks different from the other sets of experiments because the change in timeout has an impact on the shape of the clusters in the plots. This change was made to give the tuner more time to process each problem instance and adjust the configurations more efficiently. The observations are similar to the previous experiments. In this set of experiments, the majority of the results are placed below the diagonal, which indicates that the portfolio did not outperform NNENUM in its default configuration. This is the case for both the test set results and the training set results. Each graph has clusters of results in slightly varying places which indicates that the constructed portfolios were different from each other. These observations mean that the constructed portfolios are not better than using NNENUM in its default configuration. The different shapes of clusters do imply that the tuning process results into unique portfolios, but these differences don't have a major impact on the performance of each portfolio.



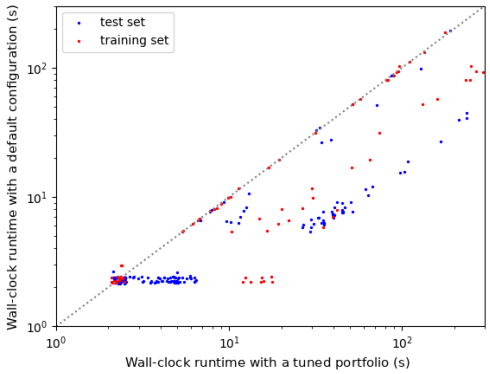
(a) 8 problem instances



(b) 16 problem instances



(c) 128 problem instances



(d) 64 problem instances

Figure 6: Performance of a portfolio of 2 configurations of NNENUM tuned with 1 network structure. Each portfolio is trained with a different number of problem instances.

5.4 Portfolio performance overview

To accompany the visual analysis of the experiments, some statistics of the experiments can be analyzed. After completing all experiments, some statistics were calculated that can inform about the performance of the constructed portfolios. A sample of these statistics can be found in Table 1 and the full table of statistics can be found in Appendix B. For each combination of the amount of NNENUM configurations present in a portfolio and the number of problem instances used in the tuning process, we can find the mean wall-clock time, the number of timeouts and the PAR10 score. An often used metric in algorithm configuration literature is the PAR10 score. This score is calculated similarly to calculating the mean of wall-clock time each problem instance needed to be solved. The only difference is that the wall-clock time of problem instances that reached the timeout cutoff were multiplied by 10. This penalizes portfolios that are unable to solve more problem instances within the specified time than other portfolios. This is needed because a portfolio should be able to solve as much problem instances as possible

In its default configuration, NNENUM has a better PAR10 score than all the constructed portfolios. It means that constructed portfolios have not been able to improve upon a default configuration of NNENUM. Both with a single tuned configuration of NNENUM and multiple configurations of NNENUM this was not possible. The number of timeouts and mean wall-clock also seems to be significantly worse when they are being solved with constructed portfolios. This matches our findings in the visualizations from the previous sections.

Portfolio properties		**Performance results**		
# NNENUM configurations	**# problem instances**	**Mean wall-clock time (s)**	**# timeouts**	**PAR10 score (s)**
1 (default)	-	47	143	339
1	16	79	374	490
1	32	80	374	531
1	64	84	389	538
2	16	79	374	490
2	32	80	374	531
2	64	84	389	538
3	16	78	320	496
3	32	82	324	624
3	64	81	346	527

Table 1: Overview of the performance of the constructed portfolios for solving verification problem instances. The number of problem instances refers to the amount of problem instances that were being used during the tuning process of the portfolio.

6 Conclusions and Further Work

This research had the goal of analyzing the effectiveness of Auto-Verify's ability to create portfolios of neural network verification tools. By running these verification tools in parallel in complementary configurations, we should see performance improvements during the verification process. The software package combines various components to create these portfolios and provides an interface to run neural network verification experiments. While Auto-Verify is able to create portfolios of configurations of verification tools and execute portfolio-based neural network verification, the effectiveness of complementary configurations is still unknown.
To analyze the effeciveness of Auto-Verify's portfolio construction, portfolios were constructed with several numbers of configurations of NNENUM tuned with problem instances of the MNIST dataset. We chose NNENUM because it requires less resources than other verification tools, and also because it crashes less during the verification process than the other supported verification tools for Auto-Verify. This minimizes any problems with the verification tool itself. In the experiments the performance of these portfolios were measured against a singular default configuration of NNENUM. By analyzing the differences in performance between these two methods of performing neural network verification, we can say if Auto-Verify is able to effectively create portfolios of neural network verification tools with complementary configurations. These portfolios consisted of one, two and three configurations of NNENUM and used different numbers of problem instances to tune their configurations. We also used different numbers of neural network structures to further compare performance differences.
Based on these experiments, we draw several conclusions about the effectivity of Auto-Verify and can give some recommendations for further work on Auto-Verify.

6.1 Conclusions

Our first conclusion is that the number of NNENUM configurations available for portfolio construction do not have a positive impact on the performance of the resulting portfolio. Compared to NNENUM in its default configuration, all portfolios consisting of optimized configurations of NNENUM performed worse. It seems that Auto-Verify is unable to effectively optimize any configuration of NNENUM because both the test set and training set used to benchmark the portfolios seem to do worse compared to a singular default configuration of NNENUM. This means that Auto-Verify is not able to optimize configuration of a verification tool.
Experimenting with the number of training problem instances used during the tuning process did not yield positive results for Auto-Verify's ability to construct portfolios. We varied the number of MNIST problem instances used during the tuning process between 8, 16, 32, 64, 128, 256 and 512 to see if that had any impact on the performance of the portfolios. We observed no significant improvements of the portfolios over NNENUM in its default configuration as all results from the portfolios seem to be worse. Auto-Verify does not seem to be able to optimize the configurations of verification tools when presented with different sizes of training sets.
Using a different number of network structures during the tuning process did not result in better performing portfolios as well. Instead of using 3 different network structures in the training set of problem instances, we used a single network structure. While the results seemed to look different from the other experiments, the conclusion we can draw from them is the same. Almost all results from the portfolios are worse than the results from the singular default configuration of NNENUM.

In this set of experiments, Auto-Verify does not seem to be able to improve the configuration of the verification tools.
In all experiments conducted in this research, none of them suggested that any of the portfolios constructed by Auto-Verify with tuned configurations of NNENUM were more efficient at solving verification problem instances than a singular default configuration of NNENUM. It is notable that every created portfolio scores worse than NNENUM in its default settings. This can mean that there is a major bug that causes problems with the tuning process or there are other factors causing performance issues that need to be explored further.

6.2 Further work

While the core functionalities of Auto-Verify work, there are still several issues with the software. The software has lots of parts working together, which might cause integration issues to occur. Each of the functionalities of this software must be evaluated thoroughly to examine if there are bugs that have a major impact on the effectiveness of Auto-Verify. The experiments suggest that the tuning process actively makes the configurations of the portfolios worse, so a deeper look at the implementation of Hydra is a good first step when looking to improve Auto-Verify. Other components that are a good starting point to look for performance issues are the resource strategies class which computes which resources are allocated to a specific verifier, and the portfolio runner class which sets up the execution of a portfolio of verifiers.
While we have looked at the behavior of NNENUM in a portfolio with the VNNCOMP MNIST benchmark as a dataset, other algorithms and dataset combinations can be explored further to see if they produce the same results. We focussed on NNENUM and MNIST because they offered a reliable way to experiment with Auto-Verify, but other verification tools and datasets might interact better with the configuration optimization strategy of Auto-Verify.

References

[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.

[Bri+24] Christopher Brix et al. "The Fifth International Verification of Neural Networks Competition (VNN-COMP 2024): Summary and Results". In: *arXiv preprint* arXiv:2412.19985 (2024).

[Bun+20] Rudy Bunel et al. "Branch and Bound for Piecewise Linear Neural Network Verification". In: *Journal of Machine Learning Research* 21.42 (2020), pp. 1–39.

[Cha+17] Gabriel Chartrand et al. "Deep Learning: A Primer for Radiologists". In: *RadioGraphics* 37.7 (2017), pp. 2113–2131.

[Dem+23] Stefano Demarchi et al. "Supporting Standardization of Neural Networks Verification with VNNLIB and CoCoNet". In: *Proceedings of the 6th Workshop on Formal Methods for ML-Enabled Autonomous Systems*. 2023, pp. 47–58.

[Den12] Li Deng. "The MNIST Database of Handwritten Digit Images for Machine Learning Research". In: *IEEE Signal Processing Magazine* 29 (2012), pp. 141–142.

[GSS15] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. "Explaining and Harnessing Adversarial Examples". In: *Proceedings of the 3dr International Converence on Learning Representations (ICLR 2015)*. 2015, pp. 1–11.

[HL20] Patrick Henriksen and Alessio Lomuscio. "Efficient Neural Network Verification via Adaptive Refinement and Adversarial Search". In: *Proceedings of the 24th European Conference on Artificial Intelligence (ECAI 2020)*. 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)*. 2021, pp. 2549–2555.

[KHR24] Matthias König, Holger H. Hoos, and Jan N. van Rijn. "Critically Assessing the State of the Art in Neural Network Verification". In: *Journal of Machine Learning* 25.12 (2024), pp. 1–53.

[KR24] Raman Kumar and Priyanka Rani. "Machine Learning Strategies in Real-World Engineering Applications: A Comprehensive Survey". In: *International Journal of Intelligent Systems and Applications in Engineering* 12.16s (2024), pp. 131–140.

[Kri09] Alex Krizhevsky. *Learning Multiple Layers of Features from Tiny Images*. 2009.

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

[Lin+22] Marius Lindauer et al. "SMAC3: A Versatile Bayesian Optimization Package for Hyperparameter Optimization". In: *Journal of Machine Learning Research* 23 (2022), pp. 2475–2483.

[Lin10] Kevin Leyton-Brown Lin Xu Holger H. Hoos. "Hydra: Automatically Configuring Algorithms for Portfolio-Based Selection". In: *Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI 2010)*. AAAI Press, 2010, pp. 210–216.

[Liu+21] Changliu Liu et al. "Algorithms for Verifying Deep Neural Networks". In: *Foundations and Trends in Optimization* 4.3-4 (2021), pp. 244–404.

[Pal+21] Alessandro De Palma et al. "Improved Branch and Bound for Neural Network Verification via Lagrangian Decomposition". In: *arXiv preprint* arXiv:2104.06718 (2021).

[Pap+16] Nicolas Papernot et al. "The Limitations of Deep Learning in Adversarial Settings". In: *Proceedings of the European Symposium on Security and Privacy (IEEE)*. 2016, pp. 372–387.

[SN20] Samuel Henrique Silva and Peyman Najafirad. "Opportunities and Challenges in Deep Learning Adversarial Robustness: A Survey". In: *arXiv preprint* arXiv/2007.00753 (2020).

[Spe24] Corné Spek. "Auto-Verify: A framework for portfolio-based neural network verification". MA thesis. LIACS, Leiden University, 2024.

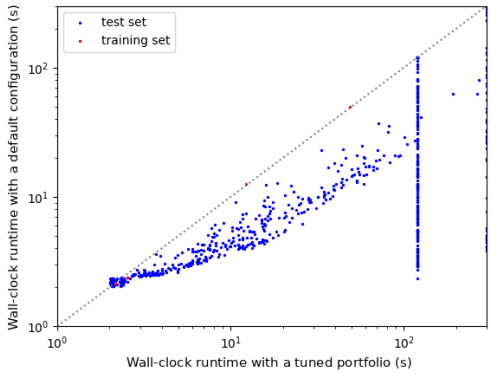
[TT19] Vincent Tjeng and Russ Tedrake. "Verifying Neural Networks with Mixed Integer Programming". In: *Proceedings of the 7th International Conferene on Learning Representations (ICLR 2019)*. 2019, pp. 1–21.

[Wan+21] Shiqi Wang et al. "Beta-CROWN: Efficient bound propagation with per-neuron split constraints for complete and incomplete neural network verification". In: *Advances in Neural Information Processing Systems (NeurIPS 2021)* 34 (2021), pp. 29909–29921.

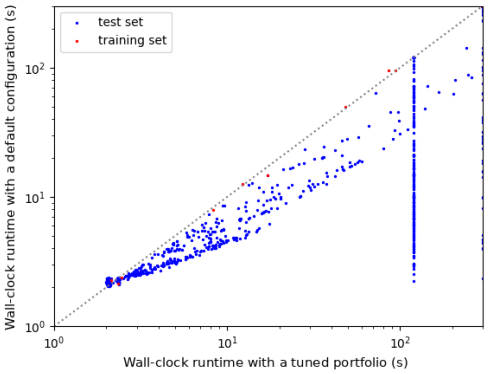
[Wei19] Zhen Wei. "Machine learning applications in finance: some case studies". PhD thesis. Imperial College London, 2019.

[Zha+23] Yong Zhao et al. "ONNXExplainer: an ONNX Based Generic Framework to Explain Neural Networks Using Shapley Values". In: *arXiv preprint* arXiv:2309.16916 (2023).

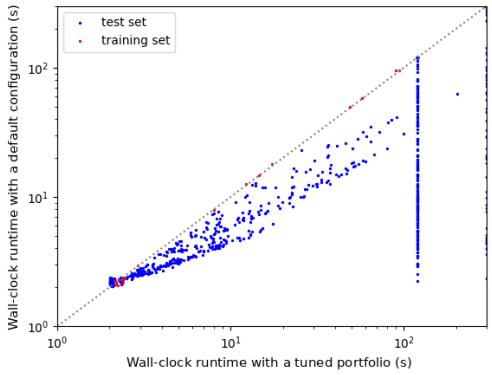
A Portfolio performance visualizations



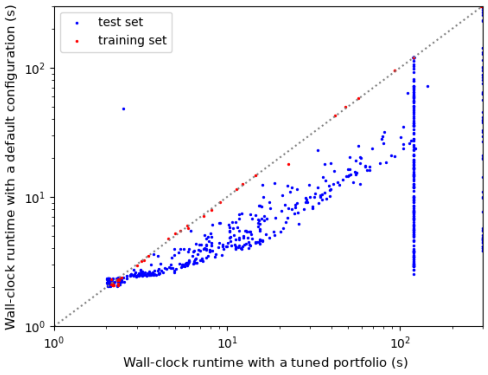
(a) Tuned portfolio using 8 problem instances



(b) Tuned portfolio using 16 problem instances

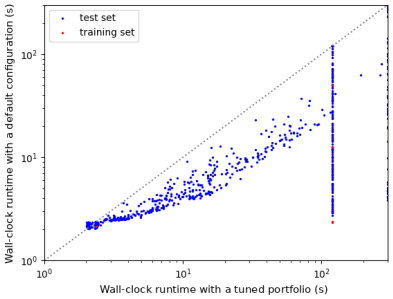


(c) Tuned portfolio using 32 problem instances

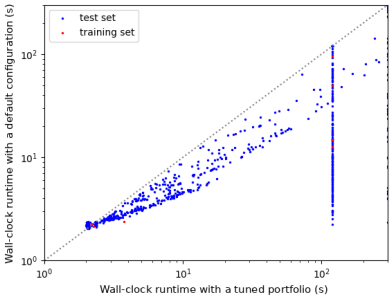


(d) Tuned portfolio using 64 problem instances

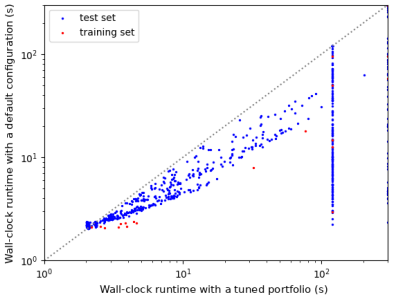
Figure 7: Performance of a portfolio of 1 configuration of NNENUM tuned with a dataset with 3 network structures



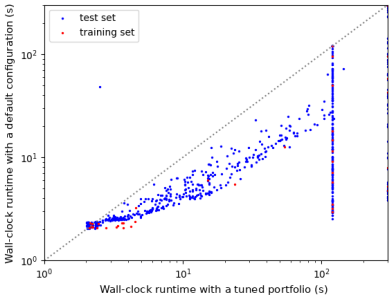
(a) Tuned portfolio using 8 problem instances



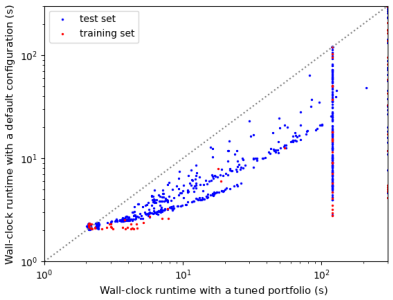
(b) Tuned portfolio using 16 problem instances



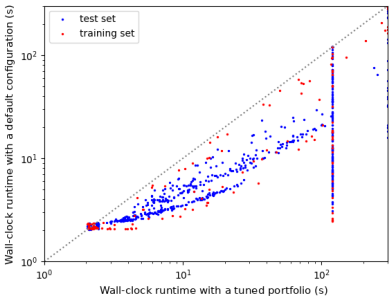
(c) Tuned portfolio using 32 problem instances



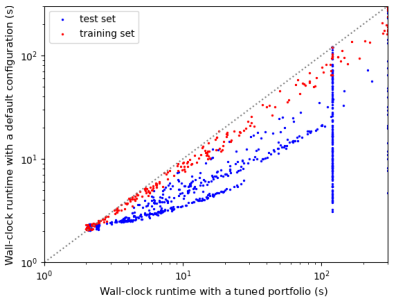
(d) Tuned portfolio using 64 problem instances



(e) Tuned portfolio using 128 problem instances

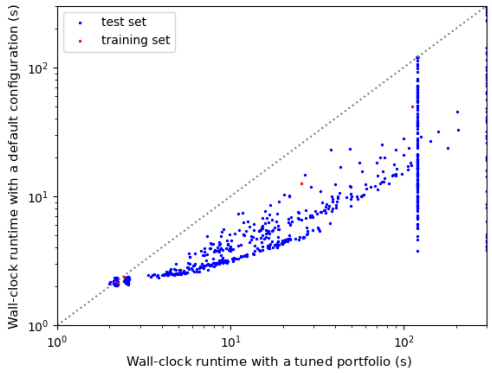


(f) Tuned portfolio using 256 problem instances

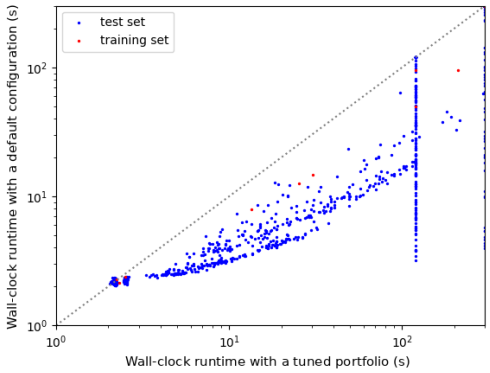


(g) Tuned portfolio using 512 problem instances

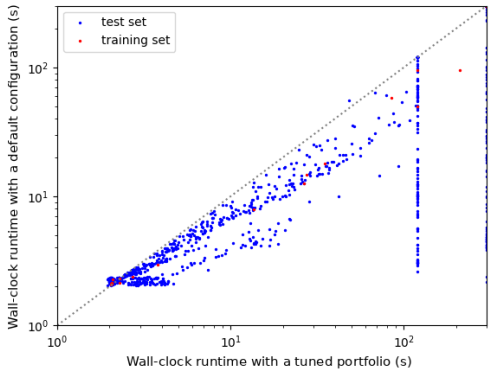
Figure 8: Performance of a portfolio of 2 configurations of NNENUM tuned with a dataset with 3 network structures



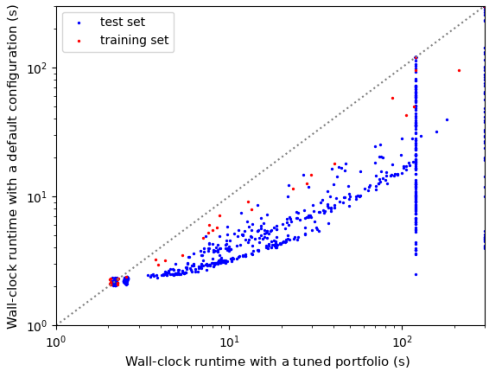
(a) Tuned portfolio using 8 problem instances



(b) Tuned portfolio using 16 problem instances

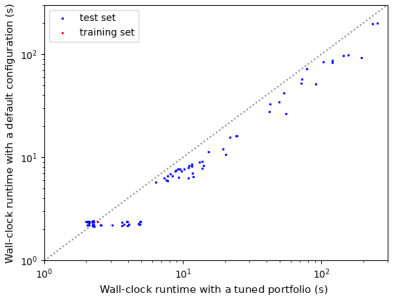


(c) Tuned portfolio using 32 problem instances

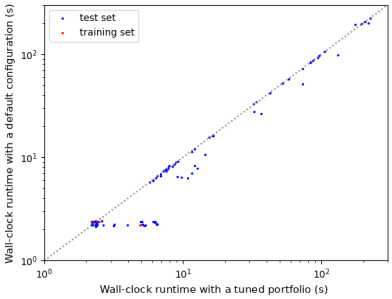


(d) Tuned portfolio using 64 problem instances

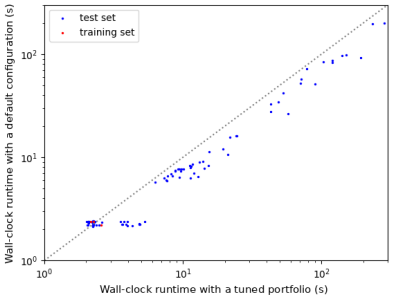
Figure 9: Performance of a portfolio of 3 configurations of NNENUM tuned with a dataset with 3 network structures



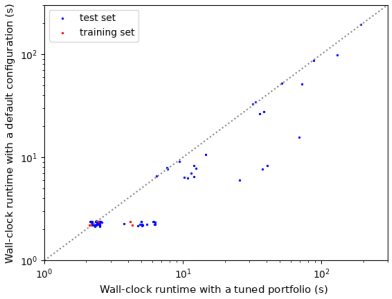
(a) Tuned portfolio using 8 problem instances



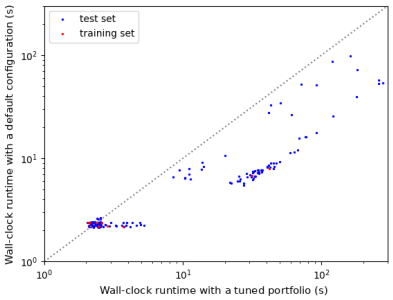
(b) Tuned portfolio using 16 problem instances



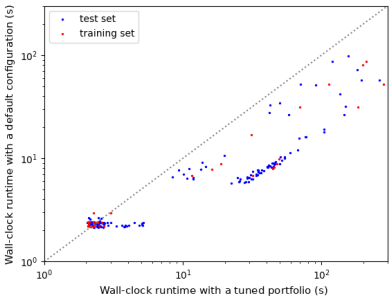
(c) Tuned portfolio using 32 problem instances



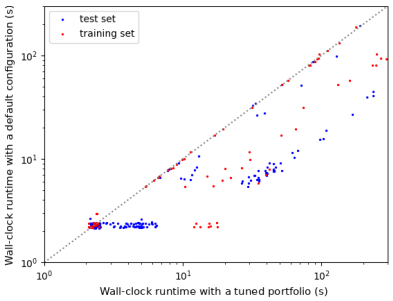
(d) Tuned portfolio using 64 problem instances



(e) Tuned portfolio using 128 problem instances



(f) Tuned portfolio using 256 problem instances



(g) Tuned portfolio using 512 problem instances

Figure 10: Performance of a portfolio of 2 configurations of NNENUM tuned with a dataset with 1 network structure

B Performance of constructed portolios

# NNENUM configurations	# problem instances	Mean wall-clock time (s)	# timeouts	PAR10 score (s)
1 (default)	-	47	143	339
1	8	84	381	537
1	16	79	374	490
1	32	80	374	531
1	64	84	389	538
1	128	51	161	356
1	256	93	433	568
1	512	88	404	523
2	16	79	374	490
2	32	80	374	531
2	64	84	389	538
2	128	67	287	358
2	256	65	276	346
2	512	66	274	357
3	8	78	332	508
3	16	78	320	496
3	32	82	324	624
3	64	81	346	527

Table 2: Overview of the performance of the constructed portfolios. The number of problem instances refers to the amount of problem instances that were being used during the tuning process of the portfolio.