Development and Application of Global Optimization Methods

PhD Thesis

Dániel Zombori

Supervisor:
Dr. Balázs Bánhelyi

Doctoral School of Computer Science
Department of Computational Optimization
Faculty of Science and Informatics
University of Szeged

Szeged
2025

Acknowledgements

First and foremost I would like to thank my supervisor, Dr. Balázs Bánhelyi, who have guided my scientific path through many stages of my academic studies. I would also like to thank Dr. Tibor Csendes, who generously provided knowledge and expertise in numerous topics throughout my PhD studies. Without their expertise, feedback, and guidance my doctoral work would not have been possible.

I am grateful for the opportunity to have worked with the many colleagues from the departments of Computational Optimization, and Technical Informatics. While a list of everyone would not fit here, I greatly value the shared work in many projects, the insightful lunch breaks, and the relaxing coffee breaks.

I owe gratitude toward my wife, Izabella Zombori-Benczur, for her invaluable patience and support throughout this long project.

I am thankful for the support of colleagues and TNG Technology Consulting GmbH as a whole, who continuously encouraged me and understood the importance of finishing my doctoral work.

I would like to thank my sister, Dóra Zombori, for the fun illustrations that ornament the chapter title pages in the personalized version of this work, which I happily share with anyone interested.

Last, but not least, the success of this work reflects the support and encouragement of my family, who have enabled my years of learning and working in academia.

Contents

Abbreviations

AA Affine arithmetic vi

AI artificial intelligence 44

B&B branch and bound 48, 50

CSP constraint satisfaction problem 49

FIFO first in first out 20

GlobalJ Java implementation of Global 11

GUI graphical user interface 12

IA interval arithmetic vi, 48, 49, 68, 70, 73

JVM Java virtual machine 23

LP linear programming vi, 3, 50

MILP mixed integer linear programming vi, 4, 50, 51, 57, 68

NFEV number of function evaluations 24, 32, 102

SAT boolean satisfiability problem 50

SCIP-Ex “Solving Constraint Integer Programs” with rational arithmetic vi

Preamble

"*While we strive for perfection without ever reaching it,*
we should not forget to accomplish the attainable."

(note to self)

As the popular saying goes,

"*to reach the top of a staircase, one has to take it step by step.*"

Life, science, and global optimization is more akin to an Escher staircase. Before one reaches the top, things might turn in unexpected directions, and the journey to this dissertation is not an exception. While my former studies focused on engineering and electronics, my scientific work started with experiments on a global optimization algorithm, then it ventured to verification of neural networks. Through the meander of topics, I hope to introduce the reader to these worlds, take a deep dive in some specific questions and understand some broader connections.

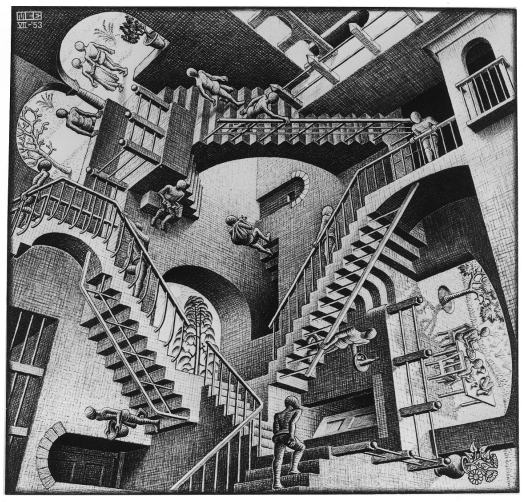

Figure 1: *Relativity (M.C. Escher, 1953)*[1]

[1]All M.C. Escher works © 2022 The M.C. Escher Company - the Netherlands. All rights reserved. Used by permission. www.mcescher.com

Chapter 1 discusses introductory topics that help to understand the present work. We start with a brief overview on global optimization and black-box optimization, to understand the variety of problems and algorithms. A short introduction to parallel computing principles will help us understand the challenges of a multi-threaded optimization algorithm. Interval arithmetic is a cornerstone in reliable computing, its core principles will help us in verification. Affine arithmetic is also used in verification, it's modeling principles can be found in multiple algorithms with similar capabilities. To understand the successes and failings of neural network verification, we need to be familiar with linear programming (LP) and mixed integer linear programming (MILP) problems. A brief introduction is given tailored to the topics of the dissertation.

In Chapter 2 parallelization of the GlobalJ algorithm is presented, with multiple approaches. An algorithm running on multiple *CPU* threads provides scaling for easy to compute objective functions. Computationally hard functions might require an entire computer to evaluate, we provide an integration approach of the multi-threaded algorithm. An additional flavor of the multi-threaded approach provides a lightly coupled algorithm, that is better for objective functions with long runtimes. Finally, we propose a distributed algorithm skeleton for multi-computer environments, where a single computer will not suffice for the optimization task.

Chapter 3 presents a vulnerability of neural network verification methods, that are based on MILP problems. We can exploit roundoff errors committed in the verification algorithm and the MILP solver. These rounding errors can hide arbitrary behavior of the neural network, therefore they break verification. There are more resilient and also rigorous verification methods, like Branch'&'Bound based on interval arithmetic (IA) or Affine arithmetic (AA), error free MILP solvers like SCIP-Ex, Symbolic Interval Propagation combined with an LP or MILP solver. The problem of these algorithms is complexity, which hinders their use on practical size neural networks. A new limitation on the usability of interval arithmetic is presented. Different error sources are explored in algorithms based on MILP solvers, heuristic improvements are suggested that could improve reliability of otherwise vulnerable methods. An efficiency improvement is presented for MILP solver based verifiers, reducing the required number of model evaluations.

Chapter 1

Introduction

Computing is a widespread tool in science, that enhances our capabilities based on mathematics, calculations, and proofs. In many applications we want to compile the available information into simple statements, that further our understanding. One field of research to achieve this is global optimization. Global optimizer algorithms search for specific inputs, that are deemed "good" by an objective function, which encodes the user's interests. The objective function is used to guide the optimizer in finding interesting inputs, or to find what the possible extreme outputs are. Depending on the form of the objective function and the specific algorithm used, the strength of a result can range from a possibly correct guess to a definitive answer with absolute certainty to be correct.

Global optimization is not the only field that operates on evaluations of a function of interest. Verification answers questions about the existence of properties, that are expressible by a model of specific form, suitable for proofs. The two fields have a deep underlying connection. Given a strong enough global optimizer, any verification problem would be expressible in terms of an optimization problem.

The field of numeric analysis is strongly related to scientific computations. The results of computations not only carry useful information, but often also contain some amount of noise. The noise can originate from the input due to uncertainties in measurement, and propagate throughout the computations. The computations themselves can also introduce noise, which might stay at manageable levels, or in certain cases might grow indefinitely. Numeric analysis provides tools to estimate and give bounds to error growth caused by the algorithms.

In both global optimization and verification, the practical problems are sizeable and require large amounts of computing power. One possibility to increase the

amount of available computing resources is multi-threading. Computationally expensive parts of the algorithms usually allow for parallel processing, there is an opportunity to scale our algorithms with the increase of available computing power.

1.1 Global optimization on black-box functions

Global optimization problems are very diverse in nature, the specific problem formulation often requires a special algorithm to solve efficiently. As industries and scientific research moves towards computational approaches, the use of simulations and machine learning makes understanding and tackling optimization problems an increasingly hard task.

Formally, we would like to solve optimization problems of the form

$$\begin{aligned} \min_x \ & f(x) \\ & h_i(x) = 0,\ h_i(x) \in \mathcal{H} \\ & g_j(x) \leq 0,\ g_j(x) \in \mathcal{G} \\ & a \leq x \leq b, \end{aligned}$$

where $\mathcal{H}$ and $\mathcal{G}$ are the sets of constraint functions, and we search for the global minimizer point of the $f(x)$ objective function. The $h_i(x) = 0$ and $g_j(x) \leq 0$ equality and inequality, as well as the a and b bounding vectors define the feasible search area, where the objective function is minimized. By optionally specifying the a and b vectors, the feasible space can be easily forced to be a finite volume, by element-wise limiting the x vector.

In practice, finding feasible points using the $h_i(x) = 0$ and $g_j(x) \leq 0$ expressions can be extremely hard. Luckily, we have options to find practical solutions to the problem [1]. In short, we can introduce penalty functions to represent our constraints. If modeled correctly, the penalties will guide the optimizer toward the feasible space, without modifying function values of the feasible space. The $f(x)$ objective function is replaced by a new $F(x)$ function, that contains the penalties and eliminates the need for the $h_i(x) = 0$ and $g_j(x) \leq 0$ constraints.

$$\begin{aligned} \min_x \ & F(x) \\ & a \leq x \leq b \end{aligned}$$

The formulation with penalties allows for algorithms that are much easier to create,

algorithms presented in this work also expect this formulation.

While a lot of problems can be described in a model suitable for well-known efficient algorithms, in many cases the analytical form of the objective function is unknown. Complicated simulations are infeasible to express with algebraic tools, nonetheless we can evaluate them. The above formulation for global optimization problems does not rely on the structure of $F(x)$. A function where only the output can be determined for a given input is called a black-box function, or black-box model. A function specified by an algorithm implementation does satisfy the black-box model.

In this work we focus on black-box optimizers. They do not require knowledge about the objective function, based on evaluations they are capable of navigating the unknown N-dimensional landscape.

1.2 Linear programming

A special case for global optimization is linear programming (LP). Opposed to black-box optimization LP problems are white box. The optimizer has access to an analytical formulation of the objective function. The objective function of a linear programming problem consists of two parts. A linearly bounded search space called the "model" given by linear inequalities specifies the feasible solutions. On the space of feasible solutions a linear expression is evaluated. The linear expression is referred to as the "objective function" in scope of a linear programming problem. Solvers try to minimize or maximize the function value in the search space.

LP problems are limited in modeling capability, as the name suggests only linear relationships can be used to define the problem. This compromise enables efficient and guaranteed solution of the global optimization problem[1]. A guaranteed optimal solution is very useful, as it ensures that the optimization process reaches a definitive optimal result with a clear limit on how good a solution can get.

There are different kinds of algorithms to solve LP problems. The most well known is the Simplex algorithm by Dantzig [2]. It iterates through feasible extreme points of the search space until the optimum is reached. A class of algorithms to solve LP problems are the interior-point methods [3], that instead of traveling on the search space surface move into it, and take a more direct path to the optimum. Depending on the problem size, as well as sparsity and the condition number of the

[1]The Simplex algorithm for solving linear programming problems is NP-complete, the average case however is solvable in P time.

matrix describing the problem, both pivoting methods and interior point methods can be advantageous [4, 5, 6].

An extension of the LP problem is the mixed integer linear programming (MILP) problem, MILP models build on LP models by enabling integer constraints. On top of the other constraints variables can be required to have an integer value. The integer constraint enables a vastly more diverse set of feasible spaces to be used. While LP problems are only capable of encoding a linearly bounded convex set, MILP models can also encode linearly bounded concave and disjoint sets.

With the more complex geometries that can be expressed solving the model also becomes harder. The MILP model can have in the number of integer variables an exponential number of disjoint LP models to solve. There are heuristics to reduce the model complexity by cuts that eliminate parts of the search space, while they preserve feasibility and optimality of the original solution [7].

There are a lot of different solvers available to tackle LP and MILP problems. The most used commercially available ones are Gurobi and CPLEX [8, 9]. There are also open-source solvers to use, like CBC, GLPK, and SCIP [10, 11, 12]. For specific problems there can be significant differences in performance between solvers.

For a lot of practical problems, the solutions based on LP and MILP models satisfy the requirements, LP and MILP solvers are considered to be robust. On the other hand, the performant algorithms are numeric methods with varying amounts of error in the solution. In mathematical proofs optimum candidates cannot be relied on without accounting for the errors committed during the model creation and model solving steps. Modeling these errors is not a trivial task, we will learn more about their effects in Chapter 3.

1.3 Multi-threaded computations

Threads in computing represent the fundamental unit of execution. A thread operates on a single sequence of instructions that is in effect[2] evaluated strictly sequentially. Modern CPU cores have a very intricate structure, where a thread is only the fundamental unit of execution in context of the execution model, while the actual implementation optimizes execution time.

[2]In modern CPUs many optimizations happen that alter the actual execution order. An important rule is that the reordered and/or parallelized computations result in the same effects and outputs. [13, 14, 15]

These complications are necessary because of the technological limits in our current hardware. Due to constraints of available electronic components and the laws of heat dissipation, the physical volume of practical CPU designs is constrained. Heat dissipation and signal travel times pose a limit on computing power of a single thread. Usual CPUs run at around 3 GHz, or a 0.33 nanosecond clock cycle. As we know, physical signals need time to propagate, at this speed light in a vacuum travels around 10 centimeters. Since a CPU is not empty space and logic gates also slow down signal propagation, electrical signals travel slower than this limit. Inside a CPU core distances are on the order of $1-2$ centimeters, but to reach memory modules, signals have to travel on the magnitude of 10 centimeters. This already poses a hard limit on how often we can read a random address in the available system memory. [16, 17, 18]

As the computing power of a thread is around its limits, our hardware and therefore software needs other strategies to enable larger volumes of computations. From the old days of using co-processors to achieve specialized tasks, we progressed to using multiple generic processing units called cores. Originally a core was interchangeable with the notion of a thread, in current CPU designs a core usually has multiple virtual threads, further complicating the nomenclature.

In our software it is not trivial to utilize the multi-threaded environment. While programming languages support easy creation of asynchronous threads, and therefore easy access to parallelization, performance is often impacted by a suboptimal setup. The difficulty of creating a good parallel software implementation depends on the problem at hand. For simple operations that have to act on large amounts of data parallelization is relatively easy. “Only” a good partitioning, distribution of data, and collection of results is required. Based on a good design implementing the parallel evaluation processes is relatively easy. For a more intricate algorithm that is not easily separated into threads acting on a separate batch of data, parallelization is a challenging task. There are many other aspects of an algorithm that can be compiled into a parallel version, we will discuss the relevant ones in Chapter 2.

One important aspect of parallel computations is information sharing. There are two types of approaches, they are used depending on the problem at hand. The conceptually and implementation-wise simpler approach is inter-process messaging. In this information sharing model processes can send messages to other processes, as they see fit. Either a message handler, or the recipient process has to store the messages until they are processed, while the sender process can continue execution. When the answer is needed, the sender wait until it is received. One drawback of

messaging is that the active attention of the recipient process is needed to incorporate the transferred information. A more complicated approach is the use of shared memory between multiple processes. An advantage of this approach is that the information transferred to the shared workspace is instantly available to anyone, there are no in-between states in the collective workspace. The drawback is that access to the workspace has to be managed in order to avoid race conditions and the resulting data corruption. An important topic that we will not delve into is the problem of deadlocks, where a multi-threaded system cannot continue operation due to mishandled access to shared resources. While any process of the system could execute on its own, no process in the collection has control over all necessary resources. Outside of deadlocks, other issues can also arise in a multi-threaded implementation. They lead to slower than expected execution, we discuss the relevant issues in Chapter 2.

Many computational tasks require more resources than a single CPU can offer. In these cases computing clusters can be utilized, where multiple processing nodes are linked in a low-latency, high throughput network. Nodes can be purpose-built servers, or simple computers connected in a network. This computing environment can host a distributed system. In many aspects distributed systems are very similar to applications on multi-threaded CPUs. The major difference in the computing model is the lack of shared memory, and the increased communication delay. If an application state has to be maintained that is consistent at all times, a node has to offer its memory as the place to store the singular source of truth. Distributed systems most often use messaging as the form of communication instead of shared memory, as it eliminates the need for a single source of truth, and slow access to shared memory.

1.4 Interval arithmetic

Interval arithmetic [19, 20, 21] is a tool in computing that provides bounds on the results of floating-point computations. The main problem of floating-point computations is that the numbers have a finite precision representation. A consequence is that results of floating-point computations can, and often do differ from the algebraic result. The introduced errors can be very small, close to the available precision, in other cases the computing error can quickly grow with computations.

A well known example is the $2025 + 10^{-13} - 2025$ computation. In most programming languages the default implementation of this expression produces 0 as a result, instead of the correct value of 10^{-13}. While an error on the magnitude of 10^{-13} can

be considered small – any instant in the year 2025 can be expressed with only a few microseconds off –, compared to the analytical result this means 100% relative error. In simple computations this is not noticed, however when high accuracy is required or the amount of sequential computations is large, the result can be significantly off. Even worse is the fact that this computation performed on 32-bit floating-point representation instead of 64-bits can have an error of 10^{-4}. Using the lower resolution only allows us to represent a time instant in the year with a resolution of one hour, a significant drop in precision. In practical applications this difference can make a well behaved algorithm unstable. Certain numeric problems are prone to uncertainties during the computations, they are numerically unstable. Typical examples of the affected computations are repeated operations on ill-conditioned matrices[3], numeric simulations, or digital filters from the field of signal processing.

Interval arithmetic cannot eliminate computing errors, as it has to rely on the same floating-point arithmetic as other computations. However, it can be an alternative for floating-point computations, when true bounding of the analytical result is of interest, and the exact result value is less important. Instead of using a single floating-point value, interval arithmetic defines an interval with two of them. It ensures that the derived result intervals always contain the exact result value.

Many software packages implement interval arithmetic [23, 24, 25], and provide implementations from basic ($+$, $-$, $*$, $/$) to advanced ($\exp(x)$, $\sin(x)$, etc.) mathematical operations.

A nice feature of interval arithmetic is its correctness. Since the IEEE 1788 [26] standard for interval operations is defined using floating-point operations specified in IEEE 754 [27], we can be sure that the given guarantees apply on most CPUs. The interval operations are designed so that the resulting interval always contains the analytical result, which would require infinite precision to obtain. This property gives interval arithmetic the strength to be applied in mathematical proofs. As we will see in Chapter 3, the ability to determine a correct bounding for the result of floating-point computations is very important.

[3]The condition number is a measure for stability of computations. The conditioning of matrix operations heavily depends on the matrix values, hence a matrix can be ill-conditioned [22].

Chapter 2

Parallel global optimization

2.1 Introduction

Black-box global optimization is a widely used tool with a diverse problem space. Problems often arise with vast search spaces and hard to compute objective functions. With the decline of Moore's law for single-thread CPU performance we have to investigate the utilization of multi-thread execution. While this is not a recent idea in global optimization [28, 29], scientific tools often focus on simplicity for ease of research. Tools applied in practice are not always implemented to their fullest potential, parallel execution is often overlooked. As Törn and Žilinskas state, there are many obstacles in the parallelization of an optimizer algorithm, as optimizers are often tailored for linear execution. They discuss principles of different strategies, *Geometric Parallelism*, *Task Parallelism* and *Algorithmic Parallelism* from [30, 31], as a new addition they introduce *Monte Carlo Parallelism*.

Every strategy should divide the computation tasks to disjunct pieces. A universal limiting factor for parallelization is that some algorithms – or parts of an algorithm – cannot run in parallel. In these cases a middle ground solution usually exists where using limited information permits a parallel version. For example, simultaneous read and write access to a data store can cause problems, while separating the read and write operations gives a middle ground solution. Read operations have no effect on each other, but a write operation has to halt any other operation to proceed. This principle can be applied for steps of an optimization algorithm, more computing power becomes available if the use of incomplete information is acceptable.

Geometric parallelism divides the search space so that every point is assigned to a CPU core. When a point has to be evaluated, its position dictates the executing

core. This leads to uneven loading when some areas of the search space are more interesting than others. In their work it is not mentioned, but hashing – or another statistics based static distribution function – could resolve the problem of uneven CPU load, with the cost of computing the hash on all input points and increased worker-to-worker communication. Dynamic distribution of the workload is a much more popular solution, it guarantees the most even utilization of CPU cores.

An extension of geometric parallelism is task parallelism, where the algorithm is divided into sub-tasks. A master coordinates the distribution of data and generates tasks to ensure the correct execution of the algorithm steps. A sub-task can be any independent execution unit of the algorithm, while this model also permits strong coordination of algorithm parts. The need for strong worker coordination is a drawback of this approach.

Algorithmic parallelism executes consecutive steps in a linear fashion, but the steps themselves are parallelized. Steps are mainly divided to equal parts of data and assigned to threads. This guarantees good parallelism for suitable parts of the algorithm, however the core utilization may be very uneven. Some parts of an algorithm might not be suitable for any parallelism in this framework. According to [28] this is hard to implement, but modern programming tools provide simple solutions, this should not be a problem anymore.

Monte Carlo parallelism is a special case of parallelism. If sampling is random and processing of a sample is independent of previous samples, then running the algorithm for N samples is equivalent to running the algorithm N times for one sample. The algorithm is similar to its parts, which results in a very easy parallel implementation. The only aspect that needs attention is collecting and merging the sub-results.

2.2 Contents and contributions

In this chapter I present the newly developed parallel implementations of the Global algorithm. A brief overview is given on the Global algorithm's origin, followed by an in-depth description of the algorithm mechanics. Knowing the structure of the algorithm is important for understanding the motivations behind the design decisions of the parallel versions and for comparing with the single-thread version.

Next, SynchronizedGlobal – the first parallel algorithm version – is presented. I write about the thought process leading to the concrete algorithm design, and I describe the algorithm mechanics. After discussing the importance of algorithmic ef-

ficiency, the computational results obtained from SynchronizedGlobal algorithm are presented. Stress tests are evaluated using well known objective functions to determine usefulness of the algorithm in lifelike scenarios. The results contain interesting findings on the algorithm runtime and the number of function evaluations. Single-thread performance of SynchronizedGlobal is compared to Global, in the number of function evaluations. For correct evaluation of the multi-threaded implementation it is important to know what differences come from parallel execution. Stress test of the synchronized clustering module is presented, where effects of the load size and the number of threads are analyzed.

My contributions regarding SynchronizedGlobal with valuable input from my Supervisor are construction of the main algorithm, implementation of the main algorithm including the adaptation of the clustering and local search modules as integral parts of the GlobalJ optimization framework. I implemented the test framework – excluding implementation of the objective functions – and performed the algorithm evaluation.

The next presented algorithm is ParallelGlobal, another parallel algorithm based on Global. I discuss the motivations for a second parallel algorithm version, and I describe the algorithm mechanics. After a short summary the computational results for ParallelGlobal are presented. A high load and high CPU count stress test is conducted, with an intriguing outcome on the observed runtimes. With further analysis the unexpected results are explained. Results for a regular test function and a test function specially made for ParallelGlobal are explored in detail. The presented data provides an insight to the conditions on whether parallel global optimization can be effective.

My contributions regarding ParallelGlobal are construction of the main algorithm, implementation of the main algorithm including the adaptation of the clustering and local search modules as integral parts of the GlobalJ optimization framework. I implemented the test framework and conducted tests using the already available test functions, I introduced the specially made Spikes function. Evaluating the test results is mainly my contribution.

The final work presented in this chapter is the concept of the DistributedGlobal algorithm. Motivations for this algorithm are presented, alongside insights that are gained from the strong similarities with ParallelGlobal. After a brief overview on messaging in distributed systems, the DistributedGlobal algorithm is described. This algorithm never had a true distributed implementation, hence we lack numeric results.

My contributions regarding DistributedGlobal are construction of the main algorithm based on the existing clustering and local search modules of ParallelGlobal. As a first step for use on computing clusters, I integrated the JPPF library [32], hence the GlobalJ toolbox supports remote function evaluations.

Finally, I conclude the chapter with reiterating the main achievements and results. The most important insights are shortly discussed that I gained regarding parallel global optimization, alongside interesting research directions that could advance capabilities in this research topic.

2.3 A brief overview on the history of Global

`GLOBAL` is originally a Fortran subroutine (hence the upper case name) introduced by Csendes in [33] implementing the algorithm by Boender et al. described in [34].

First implementations of the Global algorithm were written in Fortran. In that era computing power and memory were major obstacles. This is reflected in the fact that the original implementation hard-coded 20 stored local minima for the global optimization process. Since then system resources became a smaller issue, however we naturally want to solve the biggest possible problems, optimization tasks always expand to the limits of the computer. After the Fortran version implementations in C and MATLAB enabled the broader use of Global. The MATLAB version was still lacking modern programming paradigms and mostly reflected the original Fortran implementation. This includes a fixed limit of stored local minimizers and no separation between Global and the local search method, Unirandi.

The current best implementation of Global is found in GlobalJ, a Java package created using modern programming techniques [35, 36]. The main authors of the package are Balázs Lévai and Balázs Bánhelyi. They converted the algorithm to a modular system and improved the overall implementation. Besides the ancient algorithm origins, several factors motivated a new version. MATLAB is oriented towards experts, general users might not have the opportunity to use a MATLAB implementation. As it is a niche language, the user-base is relatively small. MATLAB also has a quite significant licensing cost, not reasonable for one-off uses, especially with a plethora of freely available mature programming languages beyond it. Speed is an important question, depending on the type of problem MATLAB is either very fast, or very slow. Matrix operations are highly optimized, and with a big enough problem size MATLAB outperforms naive low level implementations. But MATLAB's script

language is interpreted, which makes simple control flow programs orders of magnitude slower than a compiled version. Another issue is its weak ability of modeling. While any operation concerning matrices is either implemented or simple to achieve, support for other use cases is awkward. *For example the standard way to encode a polynomial is to create a vector of coefficients in reverse power order, instead of an explicit solution. Multiplying polynomials is not obvious without knowing about convolution. But it works, and with knowledge on the mathematical background it is perfectly clear why.* Although an object-oriented system is implemented in MATLAB, it received a lot of criticism, it is still not used very often. *A clear indication of the problem is that I have worked with MATLAB for over 5 years and just found out that it has an OOP system by accident.* In simple terms MATLAB is tailored towards experts in the field of science and engineering with occasional exceptions. Global has potential for a much wider audience, in this regard using MATLAB is a serious limiting factor.

The main alternative languages for re-implementing `GLOBAL` were C, C++, Java, and Python. C is not an object-oriented language, without a type system its modeling power is low compared to other languages. C++ is a good candidate, the downside is its high learning curve that would prevent easy accessibility for inexperienced users. Both Java and Python are object-oriented languages that are trivial to use in most environments. They both have a large user-base and require relatively limited knowledge of the language for effective use. Python is an interpreted language, which inherently makes it much slower than a native binary. Java is almost as fast as compiled languages, as it is half-compiled to an intermediate binary representation. The byte code is executed by the Java virtual machine, which is supported on almost any hardware platform.

Combining these factors the decision for a new implementation was made and the chosen language was Java. This choice made it even easier to use Global. As it later turned out, we can build algorithms from simple configuration files using the Java reflection package. Implementing a generic algorithm configurator, with a GUI is also a possibility. Applications can use the type information to create properly initialized algorithm objects, without the need to hard code any parameters. Extending the package with new algorithms including out-of-box support is very easy.

2.4 The Global algorithm

Global is a stochastic algorithm for solving problems with black-box objective functions. Global utilizes the multi-start strategy, a simple solution to perform global optimization tasks with a local search algorithm. Random initial points are generated, then evaluated using the local search algorithm of choice. Starting at the seed point, the local search algorithm finds the local minimizer x^* in the region of attraction. Usually this mathematical description of local optimizers is not followed in implementation. Searching exactly on the function gradient is an expensive process, otherwise known as numeric integration. The goal is to find a local minimizer with a minimal number of function evaluations, hence both gradient descent and stochastic methods can move between regions of attraction. Algorithms are only guaranteed to settle near an x^* local minimum, the region of attraction of which is not well-defined.

2.4.1 Unirandi

Global is usually used with the `UNIRANDI` local search algorithm, described in Algorithm 1. It is a derivative free stochastic method utilizing random search directions. Unirandi makes a compromise between descent speed and the number of function evaluations. Instead of simply stepping in random directions with lower function values, a line search is used to improve robustness. A line search tries to improve the function value along a single direction, it is an optimizer for one dimensional problems.

Unirandi is started with the x seed point and the λ initial step length parameter. In lines 3-17 x is gradually moved towards a local minimizer. When the loop exits, according to the convergence criteria x is near the x^* local minimizer. In lines 4 and 5 a unit length $\vec{D}$ random direction is generated with uniform distribution on the surface of the N-sphere. From 6 to 10 a line search is attempted first in the $\vec{D}$, then in the opposite $-\vec{D}$ direction. If either one finds a point with lower function value than $F(x)$, x is updated to that point, and the λ step length is scaled up. If both the positive and negative line search directions fail, Unirandi retries with a new random $\vec{D}$ direction, without changing the step length. When two $\vec{D}$ directions fail, the step length is scaled down and the process is restarted.

The line search function starting in line 18 implements the doubling stepper strategy. In line 19 the x'_k candidate point is calculated. In line 20 x'_k is bounded to the

Algorithm 1 Unirandi (as implemented in GlobalJ)

Require: $x \leftarrow initial\ sample$
$failed\text{-}directions \leftarrow 0$
$\lambda \leftarrow$ InitialStepLength
while $convergence\text{-}criteria()$ **is not** $satisfied$ **do**
  $\vec{D} \leftarrow normal\text{-}distribution(dim(F), \vec{0}, I)$
  $\vec{D} \leftarrow \vec{D} \cdot \|\vec{D}\|_2^{-1}$
  if $line\text{-}search(x, \vec{D}, \lambda)$ **is** $success$ **then**
    continue
  else if $line\text{-}search(x, -\vec{D}, \lambda)$ **is** $success$ **then**
    continue
  end if
  $failed\text{-}directions \leftarrow failed\text{-}directions + 1$
  if $failed\text{-}directions < 2$ **then**
    continue
  end if
  $failed\text{-}directions \leftarrow 0$
  $\lambda \leftarrow \frac{\lambda}{2}$
end while

function $line\text{-}search(x, \vec{D}, \lambda)$
  $x'_k \leftarrow \lambda \cdot \vec{D} + x$
  $x_k \leftarrow bound\text{-}elements(x'_k, [-1, 1])$
  if $F(x_k) \geq F(x)$ **then**
    return $fail$
  end if
  while $F(x_k) < F(x)$ **do**
    $x \leftarrow x_k$
    $\lambda \leftarrow 2 \cdot \lambda$
    $x_k \leftarrow \lambda \cdot \vec{D} + x$
    $x_k \leftarrow bound\text{-}elements(x_k, [-1, 1])$
  end while
  $\lambda \leftarrow \frac{\lambda}{2}$
  return $success$
end function

search cube[1]. In every dimension separately x'_k is warped back to the search space. This yields the x_k point in the unit cube that is closest to x'_k.

Unirandi only stops when one of the convergence criteria is satisfied. The ρ relative convergence threshold (10^{-12} by default) prevents further iterations when the function value is not improving. If the relative change in function value falls below the threshold expressed as

$$\frac{F(x) - F(x_k)}{|F(x_k)|} < \rho$$

[1]For simplicity the search space is transformed to the unit cube, all algorithms and calculations operate in this space. Results are presented in the original problem space.

or $\lambda < \rho$ becomes satisfied, then convergence is declared. For cases where Unirandi does not converge quickly enough a limit is applied in the number of function evaluations. Regardless of the reason for stopping, the latest x point is considered the local minimizer.

2.4.2 Global

A simple multi-start algorithm based on Unirandi is an effective way of solving optimization problems. The number of function evaluations can be unnecessarily high, due to duplicate work on regions of interest in the search space. Taking samples is relatively cheap requiring one function evaluation per sample, while running a local search method needs hundreds to thousands. Quality of the objective function regions can have a big difference in the cost of local searches. Global reduces the length of local searches by taking a set of samples and discarding a portion with high function values. Starting from lower function values means fewer steps per local search, without excluding any potential global optimum from being found.

Another possibility for reducing the number of function evaluations arises when multiple local searches start in the same region of attraction. To solve this problem Global uses clustering. If a sample would start a local search near an already existing local minimizer, it is merged into the cluster preventing superfluous evaluations. A condition is needed to decide whether the sample is close to an existing cluster. If a constant d_c distance would be used, some minimizers might be covered by neighboring clusters making them hard or impossible to discover with local searches. Instead of setting a constant, d_c is determined by the statistic formula

$$d_c = clustering\text{-}distance(\alpha, N, dim(F)) = \left(1 - \alpha^{\frac{1}{N-1}}\right)^{\frac{1}{dim(F)}},$$

where N is the number of samples taken, $dim(F)$ is the number of dimensions of the objective function, and the $\alpha \in (0, 1)$ parameter controls how fast clusters should shrink. At first when only a few samples exist d_c is large, it prevents local search starts near each other. As more and more samples are taken d_c gets smaller and close by local searches become possible.

Algorithm 2 describes the so called “updated Global algorithm” found in GlobalJ. Global runs in iterations. First, it creates the set of samples that will be evaluated. Then the samples are filtered to reduce duplicate work. Finally, the remaining samples are used as seed points of a local search.

Algorithm 2 Global (as implemented in GlobalJ)

1: **while** $termination\text{-}criteria()$ **is not** $true$ **do**
2: $SampleSet \leftarrow SampleSet \cup \{uniform(lb, ub) : i \in [1, \text{NewSampleCount}]\}$
3: $SampleSet \leftarrow sort(SampleSet, ascending)$
4: $ReducedSet \leftarrow \{s_i \in SampleSet : i \in [1, \text{ReducedSetSize}]\}$
5: $SampleSet \leftarrow SampleSet \setminus ReducedSet$
6: **while** $ReducedSet$ **is not** $\emptyset$ **do**
7: $N \leftarrow |ReducedSet| + \sum_{C_k \in Clusters} |C_k|$
8: $d_c \leftarrow clustering\text{-}distance(\alpha, N, dim(F))$
9: **for** C_k **in** $Clusters$ **do**
10: $NewlyClustered \leftarrow \left\{r_i \in ReducedSet : \|r_i - c_j\|_\infty < d_c \wedge F(r_i) > F(c_j), c_j \in C_k\right\}$
11: **if** $NewlyClustered$ **is not** $\emptyset$ **then**
12: $C_k \leftarrow C_k \cup NewlyClustered$
13: $ReducedSet \leftarrow ReducedSet \setminus NewlyClustered$
14: **repeat iteration** k
15: **end if**
16: **end for**
17: **if** $r^* \leftarrow \underset{r_i \in ReducedSet}{argmin} F(r_i)$ **exists then**
18: $ReducedSet \leftarrow ReducedSet \setminus \{r^*\}$
19: $x^*_{local} \leftarrow local\text{-}search(r^*)$
20: $C_{local}, d_{min} \leftarrow \underset{C_k \in Clusters}{argmin} \left\| x^*_{local} - \underset{c_i \in C_k}{argmin}\, F(c_i) \right\|_\infty$
21: **if** $d_{min} < d_c/10$ **then**
22: $C_{local} \leftarrow C_{local} \cup \{x^*_{local}, r^*\}$
23: **else**
24: $Clusters \leftarrow Clusters \cup \{\{x^*_{local}, r^*\}\}$
25: **end if**
26: **end if**
27: **end while**
28: **end while**

In lines 2-5 the sampling and reduction takes place. A number of samples controlled by the NewSampleCount parameter is generated in the $dim(F)$ dimensional input cuboid $[P_{lb}, P_{ub}]$ with uniform distribution. The new samples are added to the $SampleSet$ and ReducedSetSize number of them with the lowest function values are moved to the $ReducedSet$. Global processes elements of the $ReducedSet$, while the $SampleSet$ accumulates all other samples taken.

In lines 6-27 samples are taken out of the $ReducedSet$ either by clustering, or by local searches as seed points, until it becomes empty. The d_c distance is calculated at the start of every loop, with samples in the $ReducedSet$ and in the clusters taken into account. While samples removed from the $ReducedSet$ always end up in a cluster leaving the sum unchanged, the x^* point increases it by one. Note that N is determined by counting, we can simply add ReducedSetSize after the sampling phase and increment it every time a local search happens.

The FOR cycle in 9-16 tries to clusterize the samples in $ReducedSet$. For every cluster the $NewlyClustered$ set is computed. An r_i sample of the $ReducedSet$ is moved to $NewlyClustered$ if a close by clustered sample exists that has a lower function value than r_i. The distance is measured with the ∞-norm (Manhattan distance). If $NewlyClustered$ is not empty, the samples are moved to the C_k cluster and the current k-th iteration of the FOR cycle is repeated. Repeating is necessary because C_k is modified, and the newly clustered samples could cause further r_i to join the cluster.

In lines 17-25 the r^* seed point with the lowest function value is taken from $ReducedSet$, if it is not empty. A local search is performed finding the x^*_{local} local minimizer, producing the $\{r^*, x^*_{local}\}$ cluster candidate. To avoid creating duplicate clusters, a check has to be performed whether a cluster at the local optimum already exists. The $c_i \in C_k$ point with the lowest function value is called the C_k cluster's center. The C_{local} cluster is determined which has its center closest to x^*_{local} at d_{min} distance. If the two center's distance is less than $d_c/10$, they are merged. Otherwise, the r^* seed point and the x^*_{local} minimizer create a new cluster.

After placing r^* and x^*_{local} in a cluster, Global tries to process the remaining points in $ReducedSet$. When it becomes empty, the main iteration is finished. The algorithm either exits, or starts sampling, clustering, and local searches in a new iteration.

Global has several termination criteria that limit the number of iterations. As they are evaluated in line 1, some overruns might occur. The termination criteria can limit the number of iterations directly, or indirectly by limiting the number of samples taken overall. The number of local searches and the number of local optima can also be limited, Global also exits if no new local optimum is discovered in an iteration. Global can also exit due to more general limits, like exceeding the maximum number of function evaluations or the maximum runtime.

2.4.3 Going multi-threaded

Implementing a parallel version of Global was the next logical step to better utilize the available resources of a typical computing environment. The Global optimizer is a tool that can benefit from a multi-threaded system, although some limitations apply due to the possible misalignment of optimization problems and parallel execution.

During my research I created parallel versions tailored to different environments. Due to some poor choices in the naming of algorithms their purpose might not be clear, hence the situation is clarified here. In the context of this dissertation I will use the revised names.

SerializedGlobal will be referred to as SynchronizedGlobal to reference the "*synchronized*" keyword in Java. ParallelGlobal is used to denote two similar algorithms. From here ParallelGlobal refers to the implementation presented in [37] and [38]. DistributedGlobal is at its core the same algorithm, however changes to accommodate the distributed environment induced some differences.

"ParallelGlobal with Low Thread Interactions" is the title of a conference paper that was later extended to a journal publication. We requested a name change that did not go through properly. As a result, [38] is referred to by the original title in some instances. The paper itself and some automatically generated indexes use the new "Effects of Pooling in ParallelGlobal with Low Thread Interactions" title. Please be aware of this.

2.5 SynchronizedGlobal

SynchronizedGlobal (formerly known as SerializedGlobal) is the first parallel implementation of Global. Our goal was to utilize the resources of a multi-threaded CPU environment. Naive parallelization techniques were ruled out because of the low gain in efficiency. While algorithmic parallelism would be easy to implement, it has several drawbacks that can be prevented with better algorithm design. For example, the sampling, clustering, and local search processes all have trivial parallel versions, we can simply process the data on multiple threads separately. Since there is no data dependency, the implementation is quite easy. However, in an algorithm like Global there are frequently occurring edge cases that decrease the performance significantly. If a sample is harder to evaluate, if a local search takes much longer than others, then the load distribution quickly becomes uneven. Besides the load being uneven it materializes the worst case scenario, where the bulk of the threads quickly enter idling while a few perform most of the work. This greatly reduces the algorithmic efficiency, the algorithm is not scalable.

To prevent threads unnecessarily idling, the strict notion of an iteration was sacrificed. If a new iteration is allowed to start before the last one finished, all threads can be utilized. This design decreases algorithmic efficiency, since information from an unfinished iteration might prevent future computations. Overall, a multi-thread algorithm can more effectively utilize the available resources, which is not guaranteed to, but can result in a faster optimizer.

2.5.1 SynchronizedGlobal worker algorithm

Task parallelism with a slightly altered approach is a fitting strategy for the problem at hand. Instead of a master governing tasks, every thread runs the same algorithm and communicates in shared memory. The algorithm state is shared through data containers and variables. In most places there are no flags or state variables, workers decide their actions based on the available data. Figure 2.1 shows the decision tree of a worker in one loop. When a task is done the worker thread restarts the loop, unless the termination criteria or an exit state is reached.

Three main parts of Global are identified and turned into tasks in the multi-threaded algorithm. Sample generation is a trivial subject for creating tasks. The worker checks if samples are needed, if yes it generates one sample. Clustering is a good candidate for a task, but not as straightforward. Checking just one *(sample, clustered sample)* pair as a task would create too much overhead. Checking one sample for a matching clustered sample in all stored clusters yields better task granularity. Local searches are also trivial parallel tasks. Given an origin point they require a non-trivial amount of computing resource, while the execution of local searches is completely independent.

2.5.2 Clustering

Sample generation and local searches simply use data parallelism, however the exact inner working of the clustering process is more complicated. Since local searches are comparably very expensive, we prioritize avoiding them in the first place. Superfluous local searches might occur in a situation where the clustering of sample points depends on the order of execution. Let's suppose there is a clustered sample C, and unclustered samples A and B, where neighboring letters meet the criteria for clustering. If the order of execution checks first A and then B, only B will be clustered and A will become an origin of a local search. If first B is checked, it will be clustered. Then A is checked, and it also meets the criteria compared to B. Depending on the order of checks, the set of clustered samples can be different. To eliminate sensitivity to the order of comparisons, newly clustered samples must also be taken into account. When a new sample is clustered, all unclustered samples have to be revisited and checked against the new cluster member. If there are no new additions to any cluster, the clustering is complete.

In a single thread environment it is easy to implement such an algorithm. In a

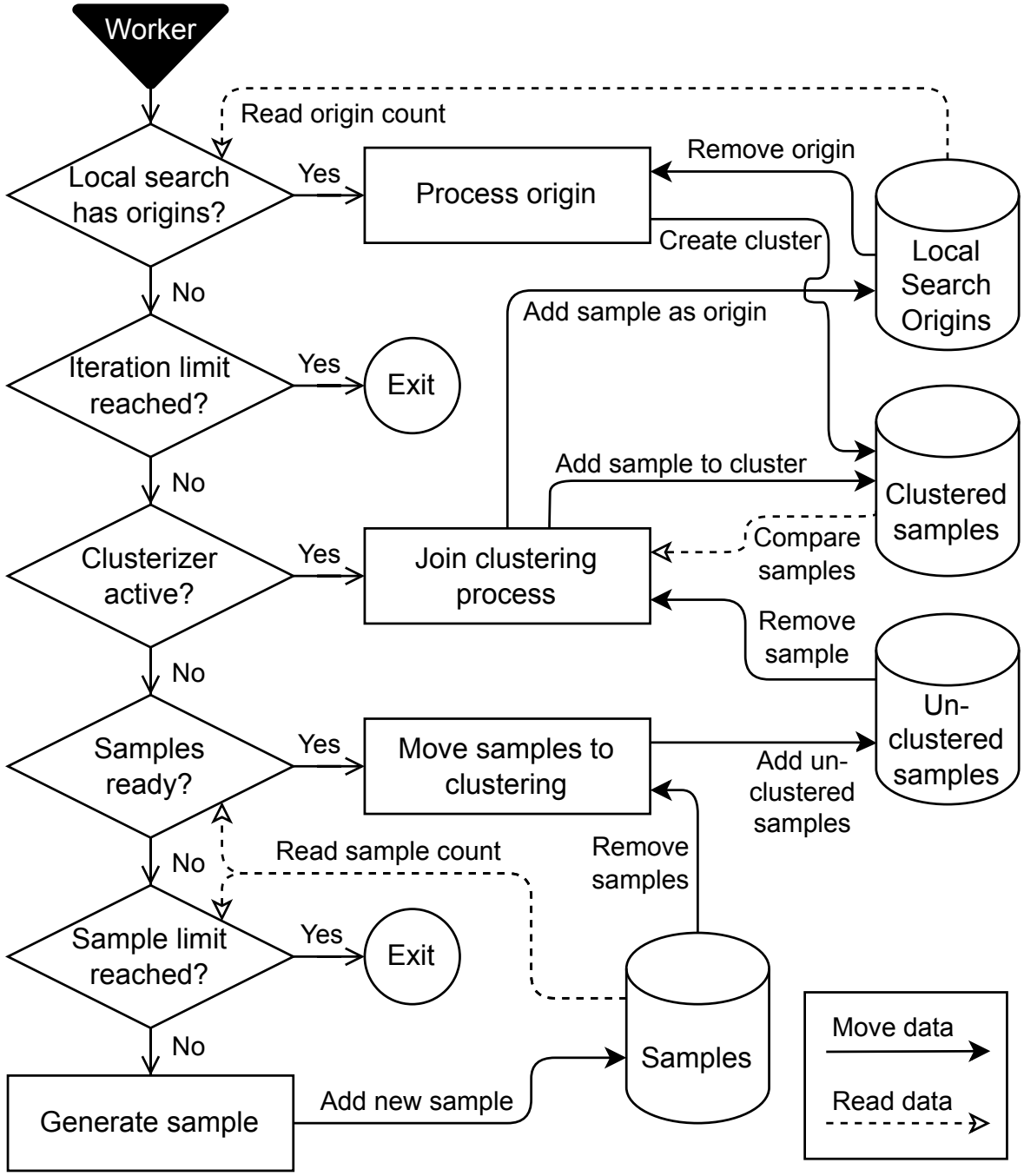

Figure 2.1: *SynchronizedGlobal worker algorithm*

multi-threaded environment ensuring the complete clustering is more complicated. A good solution is possible, based on a FIFO queue of unclustered samples, with additional information on progress. Every sample stores up to which clustered sample was it already processed. When the sample could not be clustered, it reenters

the queue as the last element. When a sample becomes clustered, already processed elements of the queue are checked against the newly clustered point. Samples are implicitly ordered in the queue based on how much clustered samples they were checked against. When the next sample in the queue is already fully processed, there is a high probability, that all remaining samples are fully processed.

When the sample taken from the queue is fully processed, threads could exit the clustering and begin a new loop in the main algorithm. However, until clustering is finished they cannot proceed, as they would have no seed points to start a local search. Busy checking of algorithm conditions until the next phase begins would create constant data races and waste CPU cycles. Instead, threads go to sleep in a gated system until the clustering is finished. In case of a new clustered point the threads wake up and try to resume unclustered samples. When the last active thread finds a fully processed unclustered sample, the clustering is finished. At this point unclustered samples are moved to the container of local search origins, the threads are resumed and released from the clusterizer.

2.5.3 Improving algorithmic efficiency

Local searches are independent and trivially separable tasks, since they only depend on the seed point and the immutable objective function. In contrast, the result of the local search will influence other threads. The newly found local optimum has to be checked against existing clusters to decide whether a new cluster is found. If a new cluster is created, then all unclustered samples waiting to be seed points have to reenter clustering, just as if an unclustered sample was clustered. With the system described above this is possible, as the source of a newly clustered point is irrelevant. Processing of local search results can be interlaced with the clustering process.

When unclustered samples are converted to local search origins, we have to be a bit careful. If all unclustered samples are converted, there is a lot of potential for superfluous local searches as a newly created cluster might have been able to absorb still unclustered samples. We have to feed the local searches with smaller portions of unclustered samples. The solution is to use a dynamic batch size. When N threads exit clustering we have the potential to start N local searches immediately. If we start one local search and wait for the result, the algorithmic efficiency is good, since we execute the fewest number of local searches overall, however the use of resources is poor. To utilize the available resources effectively, N unclustered samples will be converted to local search origins, their evaluation starts immediately. When a local

search is finished, the origin and the local optimum is put into a cluster and the worker-threads starts a new algorithm loop.

Since only as many unclustered samples were converted to origins as there are threads, the new loop can't execute a local search. If there are still unclustered samples, the thread will reenter clustering and resume the sample comparisons. When no unprocessed samples remain, N new samples are converted to origins, where N is the number of threads currently in clustering. If no other thread finished in the meantime N is one. Since there is only one thread available, a single origin is created to keep the thread busy.

2.5.4 Connecting the iterations

As Figure 2.1 shows, workers try to execute the algorithm backwards, first the local search, then clustering, and finally sample generation. This order of operation ensures that samples are pulled through the operation pipeline, they are processed in the first-came-first-served order.

When the clusterizer is empty and all samples are being processed by the local searches, some threads might take a lot more time to finish than the bulk of workers. If threads waited for all local searches to finish the effectiveness of resource usage could be greatly reduced. Such free workers can start a new algorithm iteration, since local searches consume unclustered samples and the resulting local optima are incorporated in the clusters directly afterwards.

Iterations are dependent of each other because the clustering of samples and clustering of local optima can interact with the already clustered points. Starting a new iteration only sacrifices a small amount of algorithmic efficiency when the next iteration's clustering finishes before the ongoing local searches end. Clustering phases of iterations can never mix, sampling cannot start before clustering of the already generated samples is finished.

There is a small chance that a local optimum found by a running local search would be in clustering distance of a new sample. If clustering finishes first a spurious local search will be executed.

2.5.5 Results and findings on SynchronizedGlobal

We studied the implementation of SynchronizedGlobal from two aspects. First, we verified that the parallelization beneficially affects the runtime, whether increasing

the number of worker threads results in a speedup. Second, we executed a benchmark test to compare the performance of SynchronizedGlobal to Global.

Parallelization tests measure the change of runtime on problems that are equivalent in problem complexity but differ in computation cost. To make these measurements we introduced the hardness parameter that changes the execution time of functions. We calculate the objective function $10^{hardness}$ times whenever we evaluate it. Thus, the numerical result remains the same, but the computing cost of a function evaluation is multiplied approximately by the powers of 10. The actual execution time greatly depends on how the evaluating system handles the computational hotspots. Code segments that make up significant portions of the execution time are optimized by the Java virtual machine (JVM), the real execution time can be lower than the theoretical 10 times slowdown. Considering a given hardness we can only measure the speedup compared to the single-threaded executions. Another noise factor is the nondeterministic nature of runtime. We repeated every test 10 times and calculated the average of our measurements to mitigate these interfering factors.

Tables 2.1 and 2.2 show the number of function evaluations and runtime in milliseconds, as a function of the hardness and the number of worker threads. As expected, an increase in the number of function evaluations was observed with the increasing number of threads, due to the loss in algorithmic efficiency. The overhead that we experienced was with a good approximation proportional to the number of threads, due to frequent synchronization during optimization. The runtimes show a more complex behavior, the connection to the problem hardness and algorithm threads is not linear. Our results show that for each tested objective function the runtime decreases up to 4 worker threads, even with the original function hardness. This remains true for up to 8 threads in most of the examined cases. From a point the optimization time starts to increase with the addition of threads. We call this phenomenon the saturation of the parallel optimization process. This happens when the synchronization overhead and algorithmic inefficiency overhead overcomes the gains of parallel execution. Additional threads only make things worse in such cases. The data series in Table 2.1 of the Easom function with 1x evaluation factor (zero hardness) demonstrates this phenomenon.

Saturation starts to manifest if a greater amount of threads is used, while the function evaluation cost is relatively low. Examination of the runtimes showed that increasing hardness shifts the minimum runtime to a setting with more worker thre-

Hardness	Threads	Ackley		Easom		Levy 3	
		NFE	Runtime	NFE	Runtime	NFE	Runtime
1x	1	100,447	3,553.7	10,120.7	122.7	101,742	3,245.8
	2	101,544	2,216.0	10,246.4	118.4	104,827	2,062.1
	4	102,881	1,515.3	10,506.2	112.0	112,351	1,473.0
	8	102,908	1,145.9	11,078.7	145.0	129,056	1,218.9
	16	110,010	1,319.3	12,335.9	149.0	156,907	1,548.4
10x	1	100,553	5,838.5	10,141.9	165.0	102,412	5,414.6
	2	101,510	3,370.6	10,273.7	132.0	106,339	3,057.4
	4	103,495	2,014.7	10,524.9	111.6	114,325	2,100.8
	8	105,977	1,480.0	11,096.6	135.7	126,848	1,592.3
	16	112,008	1,623.1	12,308.2	157.6	155,884	1,714.4
100x	1	100,516	27,868.7	10,117.7	413.2	102,227	25,364.9
	2	101,585	14,544.5	10,256.9	352.5	106,423	13,788.8
	4	103,420	7,806.3	10,546.1	323.9	115,158	8,030.5
	8	107,657	4,544.7	11,083.6	296.3	130,109	5,368.2
	16	115,264	3,648.3	12,313.7	257.8	167,983	5,205.0
1000x	1	100,567	258,690.0	10,198.5	1,722.4	102,028	236,066.5
	2	101,561	126,315.0	10,249.7	865.1	106,792	123,220.0
	4	103,616	68,691.5	10,521.7	508.7	114,847	70,399.6
	8	107,718	39,430.4	11,116.0	441.6	128,450	45,762.8
	16	115,875	27,021.5	12,358.5	389.3	160,364	37,713.4

Table 2.1: *Results obtained by running SynchronizedGlobal on the Ackley, Easom, and Levy 3 (as defined in Appendix B of [35]) test functions.*

ads. Saturation occurs later, as the ratio of synchronization overhead is lowered by the increased function evaluation times. On the evaluated functions at the high enough evaluation cost of 1000x the runtime is continuously decreasing up to 16 threads, which is not enough to reach the saturation point. The Easom and Shubert functions have lower runtimes and NFEV values, the optimizer finds the known global optimum with approximately 10^4 function evaluations. On other objective functions, roughly 10^5 function evaluations are indicated. For those functions the optimizer did not find the known global optimum, the 10^5 NFEV limit caused the termination. For the questions we want to answer about parallelization it is irrelevant whether the optimizer found the global optimum. Measuring speedup only requires that the evaluated computation problem is roughly identical.

On all functions and hardness values the number of function evaluations is increased due to the loss in algorithmic efficiency. About $0.5\% - 1.5\%$ more function evaluations happen per thread. The explanation is that the local searches do not stop after the global evaluation limit is reached. Every local search that started right

Hardness	Threads	Rastrigin-20		Schwefel-6		Shubert	
		NFE	Runtime	NFE	Runtime	NFE	Runtime
1x	1	100,479	2,869.3	100,155	3,027.4	10,085.1	145.0
	2	100,921	1,881.3	100,252	2,337.6	10,192.2	140.2
	4	101,436	1,413.2	100,470	1,555.4	10,381.4	128.0
	8	103,476	1,252.7	101,205	1,596.5	10,788.4	127.2
	16	107,937	1,256.9	102,285	1,408.3	11,686.9	145.2
10x	1	100,328	4,201.9	100,189	3,484.9	10,087.0	207.6
	2	101,015	2,514.4	100,362	2,334.2	10,181.5	164.9
	4	102,049	1,682.6	100,616	1,738.2	10,375.3	143.3
	8	104,993	1,364.4	100,691	1,586.7	10,842.9	141.8
	16	106,210	1,444.7	101,445	1,487.2	11,778.9	149.8
100x	1	100,354	17,438.0	100,185	7,524.7	10,091.1	836.5
	2	101,396	9,114.6	100,327	4,656.3	10,171.7	534.3
	4	102,800	4,958.8	100,782	2,858.5	10,370.2	347.8
	8	106,106	3,053.4	101,176	2,298.8	10,805.3	289.2
	16	112,977	2,627.7	102,331	1,932.1	11,718.0	246.4
1000x	1	100,485	135,399.0	100,248	44,046.3	10,091.4	6,436.0
	2	101,293	70,335.2	100,394	22,780.3	10,177.3	3,375.9
	4	103,041	37,275.3	101,183	12,169.5	10,383.5	1,847.3
	8	106,244	21,411.8	102,259	7,329.0	10,775.7	1,249.9
	16	113,135	14,740.0	104,775	5,466.7	11,679.3	979.4

Table 2.2: *Results obtained by running SynchronizedGlobal on the Rastrigin-20, Schwefel-6, and Shubert test functions.*

before the termination criteria are reached will execute normally, increasing number of function evaluations. This behavior can be prevented if the local searches consider the global limit of allowed function evaluations. It is not implemented in the current optimization framework, therefore overruns are possible.

In the next test we compared the number of function evaluations needed by SynchronizedGlobal and Global when only a single core is used. With no parallelization effects we expect identical performance, as the core algorithms should be identical.

A total of 378 configurations were tested, where a configuration consists of a global optimizer (Global, SynchronizedGlobal), a local optimizer (UnirandiCLS, NUnirandiCLS, RosenbrockCLS)[2], and one test function of the 63 available. The optimizers were run with the same parameters. The most important of which are the 400 new samples per iteration, the 15 remaining samples after sample reduction, the $\alpha = 0.01$ setting for the clusterizer, and the 10^{-8} relative convergence limit for the local optimizer. A given configuration is evaluated 100 times to counter the high

[2]The `...CLS` version of a local search algorithm is an implementation that permits configuration of the used line search algorithm. By default, the doubling stepper algorithm from Unirandi is used.

variance in the number of function evaluations. Figure 2.2 shows the distribution for the number of function evaluations, on the Easom function. In this showcase the average of an evaluation group varied between 1500 and 3300, with most values between 2000 and 2300.

The number of function evaluations is limited for every run at 10^5. If an evaluation exceeds the maximum, it is considered a failed attempt. We call the ratio of successful attempts robustness of a configuration. To acquire comparable results we only consider configurations where the robustness is 100% for both optimizers. To remove outliers configurations with runs exceeding the maximum number of function evaluations are excluded. The resulting data still has high variability, but trends are clear in an aggregated view.

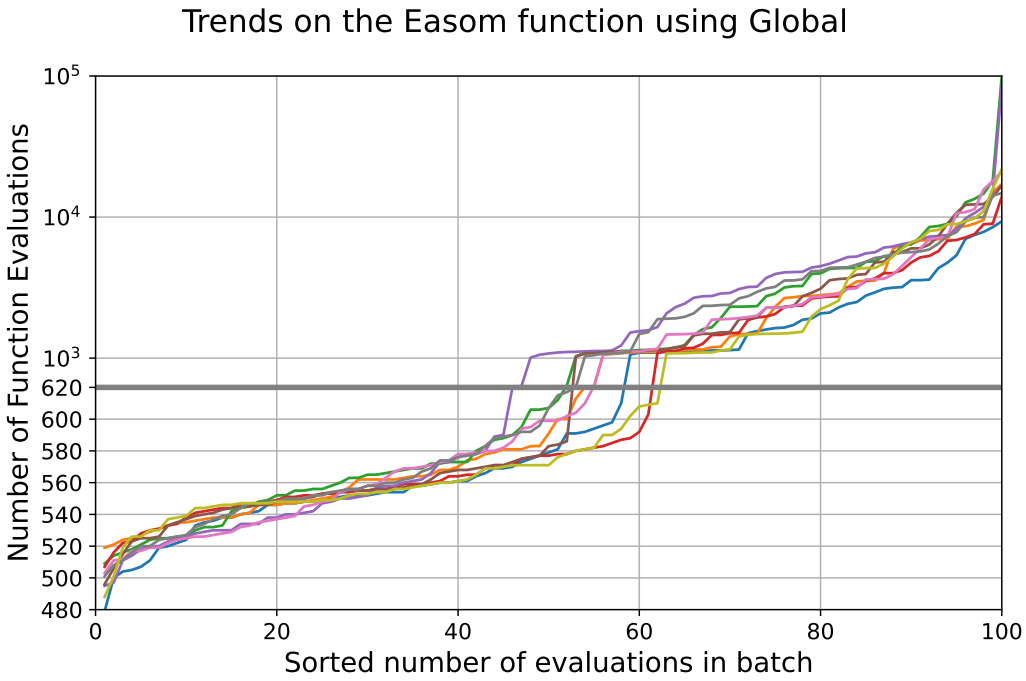

Figure 2.2: *Number of function evaluations using the NUnirandiCLS local optimizer. Colors denote the 9 groups of 100 evaluations, for which the NFEV distributions are shown. Up to 620 evaluations the Y axis is linear, above that it is logarithmic.*

Figure 2.3 visualizes the number of function evaluations needed by Global and SynchronizedGlobal, with single-thread configuration. In theory, the single-thread configuration should yield the same result, as the core algorithms are identical. Except for a few outliers the equality holds with less than 2% relative error, we can conclude the two algorithms to be identical in single-thread performance.

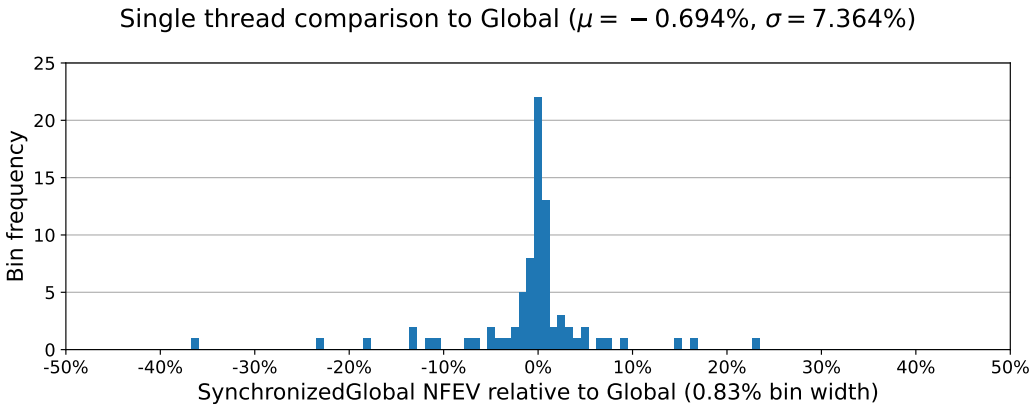

Figure 2.3: *Histogram showing NFEV results of SynchronizedGlobal relative to Global, with single-thread configurations and various objective functions. The plotted values are averages of 100 evaluations. Data points are calculated by the* $(n_{SG}-n_G)/n_G$ *expression.*

2.5.6 Stress test of synchronized clustering

The clustering module is separately tested to study the effects of parallelization on runtime. Although clustering is only loosely dependent on the underlying objective function, we wanted to study it on real data. N sample points are generated on the 5-dimensional Rastrigin function in the $x_i \in [-5, 5]$ interval. Random pairs are placed in a cluster as an origin and a local optimum point would be.

This preparation results in approximately $N/2$ clusters of two samples. The number of clusters can be slightly decreased, since two clusters with origins close enough origins are merged. As the second setup step the sample generation is repeated with N points, the new samples are loaded into the clusterizer as unclustered samples. The multi-threaded clustering is started, all threads execute the parallel clustering procedure from Section 2.5.2, without the local search step. When a thread exits the module it is stopped and cannot reenter the clusterizer.

Runtime of the clustering process was measured between its start and when all threads stopped working. The presented data is the average of 10 runs. Table 2.3 shows that some improvement is achieved even in the case of only 100 initial samples. It is also shown, that clustering easily reaches thread saturation, which at higher sample size manifests at higher thread counts. A possible solution to prevent saturation is to limit the number of threads allowed to enter clustering. After the test, the ratio of unclustered samples is around 93%, which means that the number of executed comparisons is greater than 93% of the theoretical maximum. At least $0.93 \cdot N^2$ comparisons are executed. We can conclude that parallelization of the clus-

Samples (N)	Threads	Runtime (ms)	Unclustered samples
10^2	1	7.3	92.5
	2	6.1	95.5
	4	6.6	94.3
	8	8.8	95.4
	16	12.9	94.1
10^3	1	132.5	938.7
	2	94.5	938.7
	4	77.0	936.3
	8	80.6	940.7
	16	120.1	938.3
10^4	1	11,351.5	9,372.5
	2	6,255.4	9,373.6
	4	3,549.3	9,369.7
	8	2,730.1	9,352.1
	16	2,183.1	9,354.8
10^5	1	1,996,087.9	93,685.0
	2	1,079,716.8	93,711.1
	4	533,583.0	93,690.8
	8	337,324.0	93,741.4
	16	224,280.0	93,698.7

Table 2.3: *Clusterizer stress test results showing runtime performance on different workloads and number of worker threads. Clustering starts from N clustered samples and N unclustered samples, where N is the number of initial samples.*

tering module is successful, while depending on the load a reduction in runtime or saturation is observed. For lower number of clustered samples which is typical in real configurations, saturation poses a problem. Estimating the number of comparisons and limiting the clustering threads accordingly could provide further improvement by mitigating this effect.

Module	Parameter	Value
SynchronizedGlobal	NewSampleSize	400
SynchronizedGlobal	SampleReducingFactor	0.03999
SynchronizedGlobal	MaxFunctionEvaluations	100,000 (default)
SynchronizedGlobal	LocalOptimizer	`NUnirandiCLS`
SynchronizedGlobal	Clusterizer	`SGSLClusterizer`
SGSLClusterizer	Alpha	0.01
NUnirandiCLS	MaxFunctionEvaluations	10,000
NUnirandiCLS	RelativeConvergence	0.00000001 (10^{-8})
NUnirandiCLS	LineSearchFunction	`LineSearchImpl` (default)

Table 2.4: *Configuration of the SynchronizedGlobal algorithm and sub-algorithms.*

Table 2.4 shows the parameterization of SynchronizedGlobal used in tests. Global received the same effective parameterization, with the modules coming from the original Global ecosystem.

2.6 ParallelGlobal

While SynchronizedGlobal uses every opportunity to avoid function evaluations to be efficient on expensive objective functions, ParallelGlobal aims to reduce the cost of inter-thread communications. When the objective function is cheap to compute, reducing the number of function evaluation has less relevance.

In this context thread interactions become more expensive for two reasons. First, given the same amount of evaluations and thread interactions the relative cost of interactions increases. Second, when an objective function is cheap we tend to use more function evaluations to ensure a more accurate or more robust result. In cases where the algorithm creates and tries to optimize more samples the number of thread interactions also increases. Sample evaluation and local searches are still independent tasks that can be executed without interactions, the choke point is in building a common sample pool and clustering.

2.6.1 ParallelGlobal worker algorithm

Unlike SynchronizedGlobal, the ParallelGlobal algorithm should have very few interactions between threads, a new approach of parallelization is needed to capture this design decision. The clusterizer has to execute $\mathcal{O}(n * m)$ number of operations with write locks applied m times, where m is the number of successfully clustered samples and n is the total number of samples. In SynchronizedGlobal the threads also have to wait for the completion of a full clustering cycle before they can continue with the local searches. In ParallelGlobal we can skip the strict notion of a clustering phase, hence additional function evaluations can be executed by the idling threads. We can expect both a speedup and more actual data.

As a consequence the ParallelGlobalWorker algorithm seen on Figure 2.4 shows much less complexity. In every loop the exit conditions and the algorithm steps are evaluated. The three algorithm phases remain, first samples are generated, then clustering happens, finally the local search is executed. The threads are no longer tied together in the iteration phases. Since there is no intricate ballet of tight clustering,

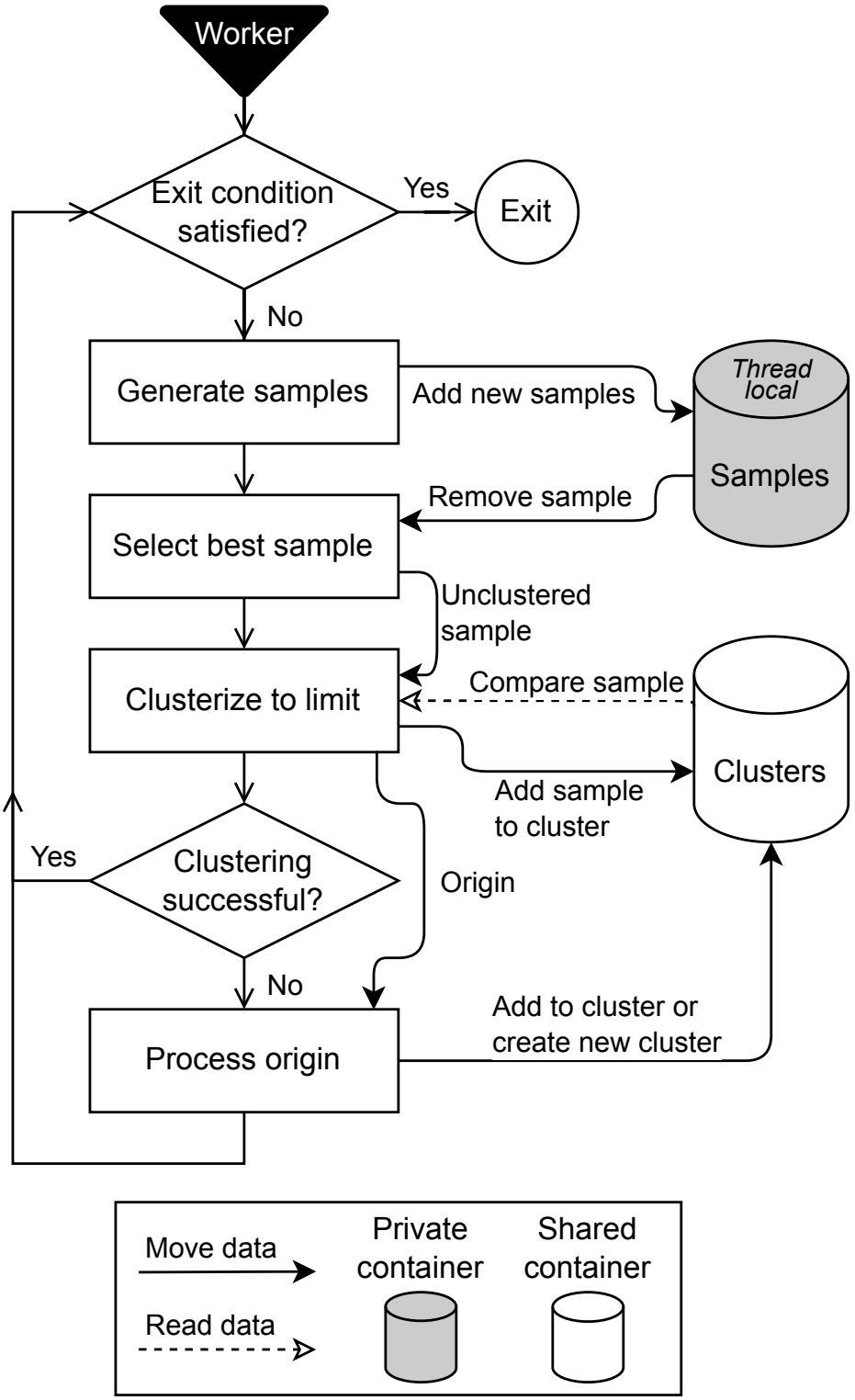

Figure 2.4: *ParallelGlobal worker algorithm*

data parallelism can take over the algorithm loop from sampling to local search.

With this change iterations are harder to define, to the point where they don't make much sense. In ParallelGlobal the number of iterations can only be counted per worker. On the algorithm level iteration phases are not present, the number of samples, local searches and function evaluations can limit the algorithm instead.

In SynchronizedGlobal sample generation is independent (only insertion to the shared container has to be synchronized). Selecting the reduced sample set (sorting the container and moving the best samples) requires coordination, only one thread can do this job and others have to wait for it to finish. To eliminate these interactions, ParallelGlobal can use a local sample pool for each thread. Because of the cheap function evaluations we expect a lot of sample generation, each thread can build up enough samples to have diversity. The biggest change is that a worker only selects one sample from the pool to work with in one "iteration". Instead of generating a set then selecting a subset based on the selection ratio, the process is reversed. The set is generated such that selecting a single sample will result in the correct ratio. A persistent sample pool can be utilized as well, where samples are not thrown away between iterations. The best sample can come from a previous lucky iteration, where multiple good candidates were generated.

2.6.2 Clustering and local search

Clustering complexity is not decreased by having independent threads, there are still $\mathcal{O}(n*m)$ comparisons and m samples inserted. With cheap evaluations m and n tend to be even larger, however with the looser algorithm the thread interactions can be decreased. In ParallelGlobal there is no central store of clustered samples. Instead, a linked list of clusters is used where the individual clusters are locked on insertion, and not the whole list of samples. The probability of collisions is much lower, given that there are at least a few different clusters. On the rare occasion when only one cluster exists the objective function most likely has one local minimizer, the algorithm will exit without generating a lot of samples.

If clustering of the sample is not successful, it becomes a starting point for a local search. When the local optimum does not fit an existing cluster a new cluster is appended to the list. To reduce the number of locks needed as much as possible, the "store length beforehand" approach is used, where new clusters are ignored in already started clustering runs. When a cluster is inserted it is enough to lock the neighboring items instead of the whole list.

2.6.3 Results and findings on ParallelGlobal

It is clear that ParallelGlobal is a much simpler algorithm than SynchronizedGlobal. Only the container for clusters is shared and there are very few synchronized variables, mainly used in checking termination criteria. The reduced opportunity for thread interactions highly decreases the effects of saturation in ParallelGlobal, algorithmic efficiency is sacrificed for higher utilization of CPU time.

Experiments on ParallelGlobal are conducted similarly to SynchronizedGlobal. Tests are evaluated with 1, 2, 4, 8, and 16 thread settings for ParallelGlobal. The hardness level of functions is also altered to simulate computationally more demanding problems, as described in Section 2.5.5. To gain more stable results every data point is evaluated 100 times. A key difference to evaluations on SynchronizedGlobal is that robustness was not taken into account. Tests on ParallelGlobal focus on the parallel behavior and speed gain instead of comparison in efficiency. To reduce computing cost, a set of 14 functions was chosen with diverse characteristics.

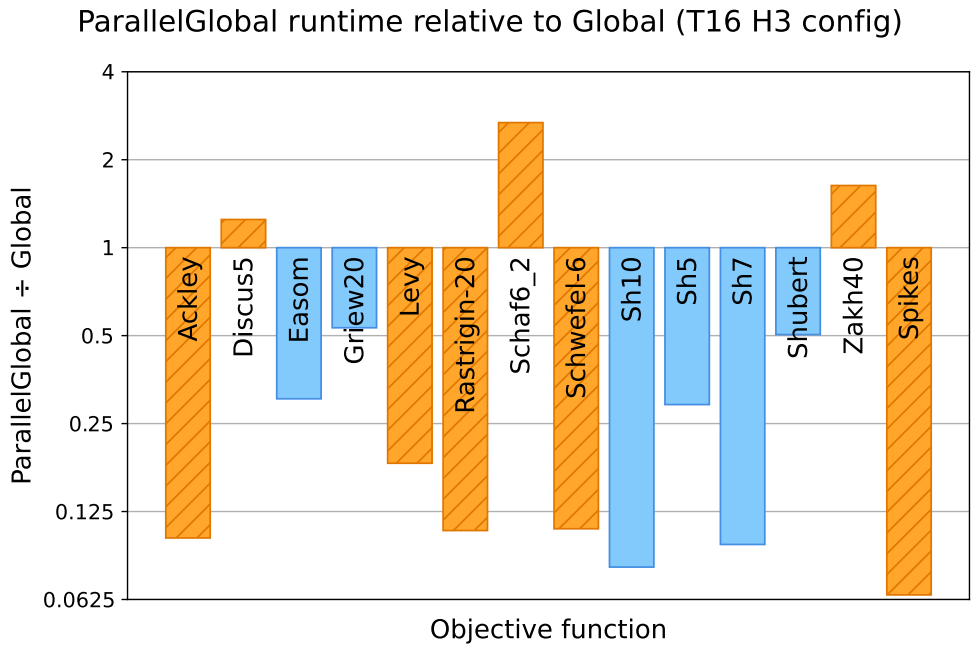

Figure 2.5: *ParallelGlobal runtimes compared to Global, with 16 threads and hardness 3 configuration. Plotted values are calculated by the n_{PG}/n_G expression. Results marked with orange hatch indicate objective functions that reached the 10^5 NFEV soft limit.*

Figure 2.5 shows interesting behaviors caused by parallelization. It shows the ratio of runtime needed by ParallelGlobal compared to Global, with 1000x hardness. ParallelGlobal is configured to run with 16 threads, which provides the most advan-

tage in available resources. Conversely, it also poses the biggest challenge, as the resource utilization must be efficient.

The functions marked with orange hatch are too difficult to solve at the given `T16H3` configuration and the limit of 10^5 maximum function evaluations. The *Spikes* function has large variance in the results, a few evaluations have reached the NFEV limit, however the average number of evaluations is around 30.000. Results fluctuate from less than 10^3 to 10^5 evaluations. Since the same amount of work is expected from ParallelGlobal and Global on functions that are not completely solved, they are also included in the results.

On most test functions ParallelGlobal is faster with less than half runtime compared to Global. On a few functions namely Discus-5, Schaffer, and Zakharov-40 the needed runtime is higher for ParallelGlobal. Considering that ParallelGlobal has the ability to use 16 threads and is still slower the cause has to be investigated.

Table 2.5 shows the numerical results for the three functions in question. It is clear that the number of function evaluations increases with the number of threads. On Schaffer the growth follows the number of threads with a low linear coefficient, on 16 threads the NFEV is doubled. On Discus-5 the growth is greater than linear, from 25% more function evaluations on 2 threads it grows to overall 5x evaluations on 16 threads. ParallelGlobal shows the strongest growth on Zakharov-40. Given N threads, the necessary NFEV is N times the single-thread amount.

With closer examination of the data the reason is clear. The number of function evaluations used in local searches closely correlates with the overall function evaluations for each configuration. The data shows that increasing the number of threads increases the number of local searches. The high number of local searches leads to the high number of function evaluations and in consequence low algorithmic efficiency. Discus-5 and Zakharov-40 is reliably solved using 1 and 2 threads, but at 16 threads all attempts failed to stop before reaching 10^5 evaluations.

In contrast, the Ackley, Rastrigin-20, Schwefel-6, Shekel-7, and Shekel-10 functions show efficiency values near the theoretical limit. On these functions the runtime is less than one eight of Global in the 16 thread case. For Ackley, Rastrigin-20, and Schwefel-6 the reasoning is simple. The number of local searches executed is significantly higher than the number of threads. The increase caused by additional threads is comparably small, it has only a minor effect on algorithmic efficiency. Since ParallelGlobal can execute local searches simultaneously, despite the reduced efficiency it gains a significant speedup.

Hardness	Threads	Discus-5		Schaffer		Zakharov-40	
		NFE	Runtime	NFE	Runtime	NFE	Runtime
1x	1	17,605	295.8	88,397	485.2	20,669	1,575.3
	2	21,331	332.5	90,897	438.2	41,215	1,707.1
	4	34,731	407.9	119,443	474.5	83,306	1,966.6
	8	55,636	488.2	145,501	528.6	163,408	2,336.5
	16	101,020	674.4	199,621	688.7	324,282	3,618.9
10x	1	18,876	328.5	90,423	531.4	20,641	1,603.9
	2	23,075	335.3	96,158	471.2	41,176	1,746.7
	4	33,302	408.5	123,748	524.3	81,237	1,901.0
	8	54,584	503.6	141,006	532.8	163,904	2,369.6
	16	101,805	674.6	196,040	708.4	324,783	3,620.9
100x	1	16,935	305.4	90,245	999.6	20,471	1,858.4
	2	22,386	335.4	97,952	707.2	40,995	2,049.9
	4	35,365	412.7	120,426	670.5	81,536	2,217.5
	8	58,902	520.0	148,541	682.2	162,631	2,670.0
	16	100,114	684.3	218,265	814.0	323,983	3,896.8
1000x	1	17,954	481.7	86,429	6,018.8	20,402	5,062.0
	2	21,057	406.6	98,078	3,691.8	41,762	5,495.4
	4	35,553	496.5	120,928	2,600.3	82,159	5,811.3
	8	59,552	597.6	143,945	2,022.7	163,595	6,453.3
	16	103,038	760.1	201,185	2,143.8	324,343	8,319.5

Table 2.5: *Results obtained by running ParallelGlobal on the Discus-5, Schaffer, and Zakharov-40 test functions.*

The explanation for the Shekel functions is different, similarly to Discus-5 the number of local searches grows linearly with the number of threads. However, compared to Global the number of function evaluations starts from a much lower value at 1 thread, hence the number of function evaluations is still relatively low at 16 threads and a speedup observed. ParallelGlobal and Global has roughly the same number of function evaluations performed in local searches, but Global evaluates much more in the sampling phase. With additional threads ParallelGlobal would reach and surpass the function evaluations of Global causing a slowdown for these functions too.

Figure 2.6 shows the detailed results on the Shubert function. Results are shown as function of the number of threads on the four hardness levels. Every data point is the ratio between ParallelGlobal and Global, where lower values favor ParallelGlobal.

On the left graph the number of function evaluations are shown, they have no difference on the different hardness levels. Due to the lower algorithmic efficiency ParallelGlobal needs twice as many evaluations on 1 thread, which grows to four times as many on 16 threads.

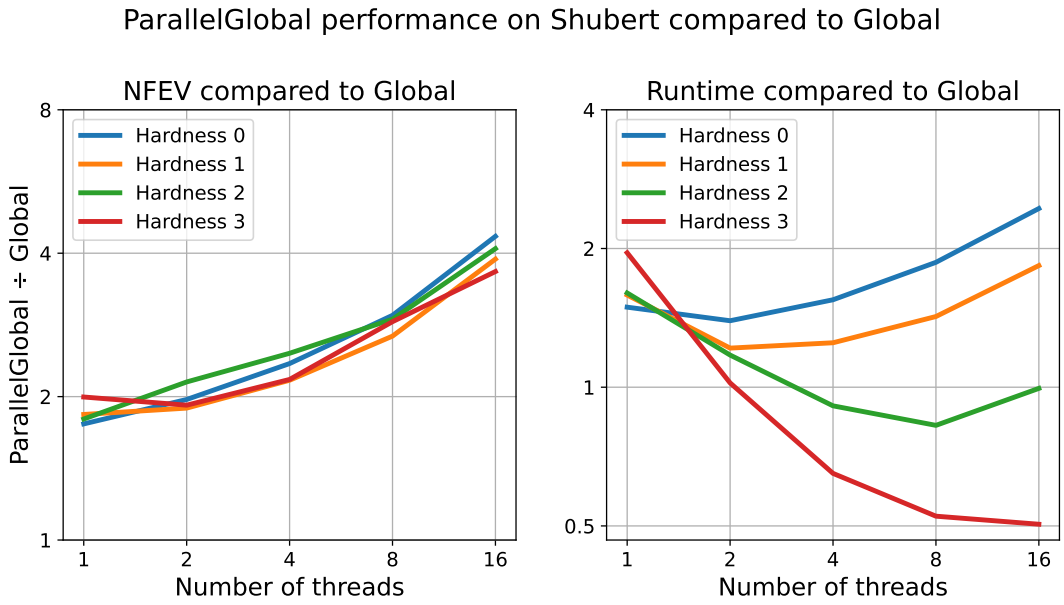

Figure 2.6: *ParallelGlobal results compared to Global on the Shubert function, with different number of threads and hardness values. Plotted values are calculated by the* n_{PG}/n_G *expression.*

Not an ideal situation, but as the graph on the right shows, given a computationally expensive objective function ParallelGlobal gains advantage. At all four hardness levels the 2 thread configuration reduces the runtime. H0 and H1 is immediately saturated, additional threads increase the runtime. H2 is improved up to 8 threads, at the 16 thread configuration the saturation is already in full effect. H3 shows improvement up to 16 threads, however the curve flattens out suggesting the saturation point is around 16 threads. On H3 ParallelGlobal is effective at 2 threads, as the runtime decreases by a factor of 2. At 16 threads despite using 8 times the computing power as with 2 threads, the runtime is only reduced by another factor of 2. In conclusion ParallelGlobal can be effective on Shubert, but it has significant limits.

Figure 2.7 shows the results on the Spikes function. Spikes is defined according to Equation (2.1), and it is specially made to test an extreme case of optimization. It has a flat global trend, meaning that the function has no slope on large scales. Another feature of the function is that the global optimum is not only a point, but a whole area. If a point is evaluated in this area, the optimization problem is solved. The points next to the area do not slope towards the area, it can only be discovered by trial and error. The only effective method is random sampling until the optimum is found.

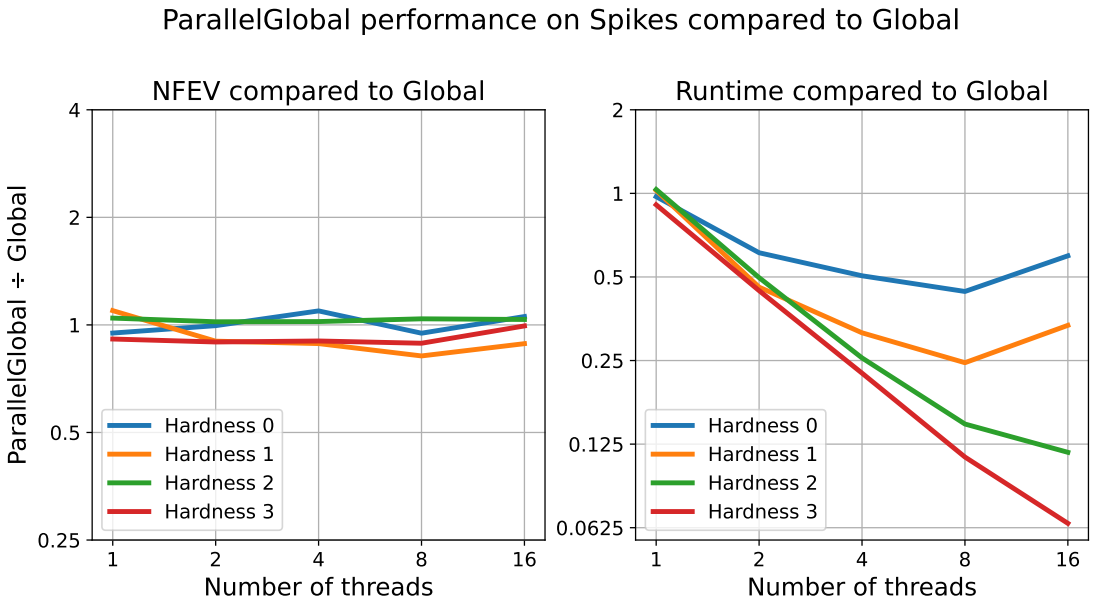

Figure 2.7: *ParallelGlobal results compared to Global on the Spikes function, with different number of threads and hardness values. Plotted values are calculated by the* n_{PG}/n_G *expression.*

The left graph shows that the NFEV ratio is constant for all thread and hardness settings. This is expected, as the chance of generating a point in the optimum area is constant. The number of function evaluations matches the amount needed by Global.

The right graph shows the runtime comparison to Global. ParallelGlobal is very effective in utilizing the additional threads, even for the lower H0 and H1 configurations the saturation only occurs after 8 threads. For H2 only the start of saturation is visible, while for H3 the efficiency is almost the theoretical maximum, 16 threads causes an almost 16-fold decrease in runtime.

$$spikes(x) = \begin{cases} 1000, & \text{if } \|x - (15.25, 15.75)\|_2 \leq \frac{1}{4} \\ 1002 + \Pi_{x_i} sin(2\pi x_i), & \text{otherwise} \end{cases} \tag{2.1}$$

2.7 DistributedGlobal

SynchronizedGlobal and ParallelGlobal are both capable of using a computing cluster by assigning tasks to remote computers and collecting the results. The most obvious remote task is a function evaluation. It is an independent computation, on expen-

sive objective functions it provides sizeable load per assignment. The drawback is that on a cheap objective function the relative cost of communication becomes quite noticeable, it can easily dominate the time cost of an evaluation.

A solution to the problem is the assignment of a larger task per node. Possible larger tasks are the whole sampling phase, a local search optimization problem, or clustering. However, this approach would reintroduce the inter-thread dependency problems experienced in SynchronizedGlobal. Instead of threads, the compute nodes would wait for each other to finish. The logical conclusion is that a version of ParallelGlobal is needed that already solves the issues with SynchronizedGlobal, and handles relatively slow worker-to-worker communication well. This is the basic motivation behind the thought experiment of DistributedGlobal.

Currently DistributedGlobal is not implemented, all insights are gathered from testing ParallelGlobal in a single machine environment. Because of the strong similarities the exercise is not meaningless, however these tests only explore a small area of the topic.

On a distributed network of computers where slow node-to-node communication has to be assumed expensive objective functions can be optimized efficiently, since the relative cost of communication will be low. Running on a computing cluster is very similar, the main difference is the improved communication latency. This improvement enables the effective optimization of much cheaper objective functions with the same algorithm. At the extreme, multiple node instances can run on a single machine. The near instant communication further lowers the limit of function cost at which optimization is effective. Theoretically, approaching instantaneous data sharing with further decrease in latency would eliminate the distinction between DistributedGlobal and ParallelGlobal. The only real difference is locality of containers, which loses its meaning when information is shared instantly. As the computing environment has a great effect on efficiency and drives the available computing power, a balance has to be found. By adjusting the algorithm parameters and the running environment, DistributedGlobal can be tailored toward the task at hand.

2.7.1 Messaging in a distributed system

In the previous two Global algorithms data was shared in memory, there was only one instance of every shared container. Since DistributedGlobal has to handle environments without shared memory, data has to be duplicated between the local containers. There are widely known distributed algorithms to solve these problems, especially for data dissemination and aggregation [39].

When a node locally acquires information through work and it is shared across the distributed system, data is added to the distributed system. Data addition can be achieved through the use of a dissemination algorithm which ensures that eventually all nodes receive the information. With dissemination algorithms the shared data reaches nodes multiple times. Deduplication of this data is easy to solve. If a sample is encountered more than once it is ignored after the first time. When clusters are received the problem is harder since the same shared cluster might arrive with different contained samples. In this case different cluster versions have to be merged. This task is solvable in many ways, however it is significantly more expensive than simple equality checks.

Data deletions in a distributed system are harder, although based on the dissemination algorithms a "mark for deletion" message can be shared to erase data in the whole system. Data is not deleted in any of the Global algorithms, therefore deletions should not be part of the system.

Moving data in a distributed system needs careful consideration to account for communication errors. Node-to-node move of data would be inefficient to solve with the system wide deletion mechanism, only the sender has to perform the delete operation. There are two error scenarios during the move operation, data can either be lost or duplicated. Data loss occurs when a message is lost in transit and the receiver never registers it. Duplication occurs when a message is sent and/or received multiple times, or the sender is not informed that the receiver has registered the data.

To mitigate data loss the *at least once* sending strategy can be used. A message is repeated until reception is confirmed, it is certain that the data is received before transmission is stopped. Obviously this strategy is prone to data duplication, since the message might be received by the recipient multiple times. If the recipient is deemed unreachable the message can be sent to another recipient. This however risks data duplication, the message might be received by multiple recipients.

To mitigate data duplication the *at most once* sending strategy can be used. A message is only sent once and reception is not checked. If the recipient happens to receive it, then it is certain that the data only arrives once at a single recipient. If the data is lost in transit, then it is deleted permanently, but at least no duplication occurs.

When both data loss and duplication has to be mitigated, the *exactly once* sending strategy can be used. By sending a message multiple times and it is checked for duplication, – e.g. with unique message fingerprints – in most cases the data arrives

safely and exactly once. This is however exactly the so called *Two Generals' Problem* that demonstrates the lack of a perfect way to communicate in an environment with possible message loss. In practice well thought out versions of the *exactly once* strategy are already implemented and should be used. For example, the well known *Transmission Control Protocol (TCP)* is available in every computing environment.

Given a network connection with relatively bad reliability at 1% message loss the *exactly once* strategy should have no issues. For a successful reception only one in a 100 thousand messages needs to be repeated more than three times. The usual networks are orders of magnitude more reliable than this, most of the time only a total loss of connection has to be handled.

2.7.2 DistributedGlobal worker algorithm

The DistributedGlobalWorker algorithm described by Figure 2.8 consists of two phases. First, data exchange happens with the distributed network, after that the optimization phase executes the now familiar modules of Global. The exchange phase is not necessarily separated from the optimization phase, reception of new clusters and new global optimum candidates can be done asynchronously, similarly to SynchronizedGlobal and ParallelGlobal. Outgoing communication like sample sharing or sending out new cluster data can happen at convenient times, so that the algorithm is not blocked. For example, when samples are clustered, when a new cluster is created, or when samples can be shared. If iterations are fast data shares might be bundled and sent less frequently to reduce the network load. For now, data exchange is considered as the first step in the algorithm.

The DistributedGlobal algorithm workers lack a centralized data store, therefore generated samples, clustered samples, and the found local minima have to be stored locally. If every node keeps a local record without sharing the data there is no point in having a distributed algorithm. Reducing the amount of duplicate work is important, therefore the available information has to be shared. DistributedGlobal nodes have two local containers, the sample pool and the cluster storage. The sample pool has less urgency to be shared, while replicating the known clusters across nodes is important. With this strategy a significant portion of local searches can be prevented, it has a great impact on efficiency.

Although data containers behave differently, sampling is identical to the solution in ParallelGlobal. The node can create a sample set such that selecting the best sample will result in the required sample reducing ratio. A persistent pool is also

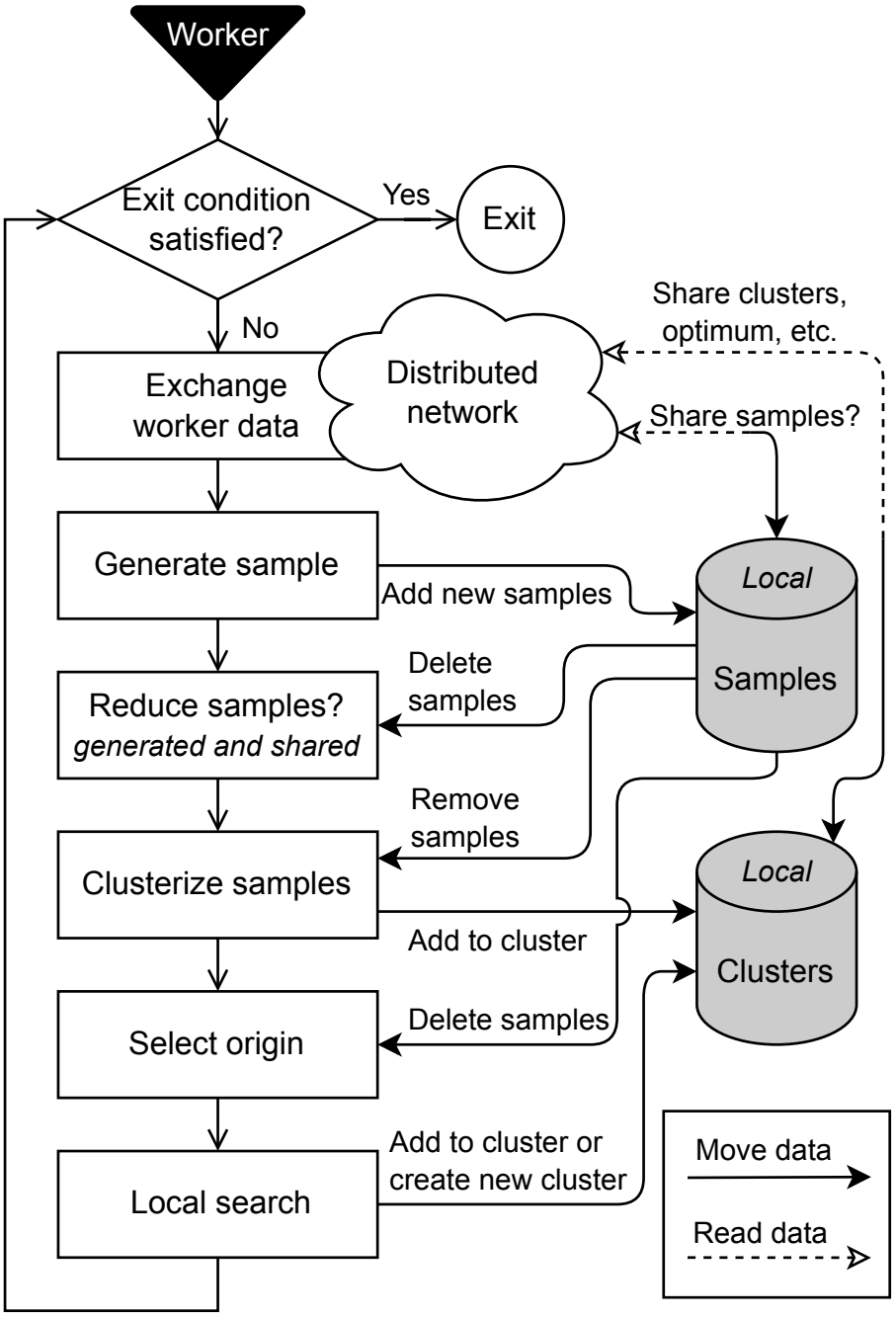

Figure 2.8: *DistributedGlobal worker algorithm*

possible and in most cases it is advised, so that it evens out the fluctuations of sample quality during sample reduction.

Besides every node creating its own sample pool, mixing the sample pools is also a possibility. It is useful to even out sample quality between nodes and reduce long local searches due to a bad quality origin. Sending a sample to another worker risks duplication and could induce a secondary evaluation wasting resources. In practical cases the *exactly once* sending strategy should be used, in most cases communication is much cheaper than evaluating a local search. If the optimization problem is such that good origins are hard to find, we should lean toward enabling duplication, for example using the *exactly once* strategy with "send until confirmation", or "change recipient on fail" policy. If good candidates are relatively easy to find but costly to evaluate, we should lean towards message loss. For example try to send up to a limited time, then consider it sent even if no confirmation happens.

The act of generating a sample pool and selecting a curated set of samples is one form of sample reduction. The goal of sample reduction is to filter samples based on the encountered distribution, such that the workload of local searches is reduced. If evaluation of a single sample is expensive, then local searches will also have a significant cost. Obtaining a good local search origin is imperative, while the overall evaluation costs also have to be kept in balance.

Other sample reduction strategies also exist. For example samples can be rejected if they don't fall under a given percentile of the encountered objective function values. Samples can be rejected based on the spacial distribution. When clustering uses the known samples to estimate the worth of a sample, spacial distribution and function value are both used. Any factor that helps in identifying good candidates for local search origins can be incorporated.

Clustering in DistributedGlobal is simple. Since the containers storing cluster data are local to the nodes, clustering is not affected by concurrency. The new sample is simply compared to the clustered samples and stored without complications, just as in Global.

Finally, execution of local searches is identical to ParallelGlobal. When a node executes a local search, the result is first stored in a cluster, then it can be shared in the distributed system, so every node receives all local minima.

2.8 Discussion and concluding remarks

In this chapter the approaches of SynchronizedGlobal and ParallelGlobal are discussed as viable multi-threaded implementations for Global. The DistributedGlobal concept is also mentioned as a way to extend the algorithm to distributed systems.

SynchronizedGlobal closely resembles the Global algorithm, on a single thread configuration they are identical. SynchronizedGlobal successfully utilizes multiple threads and keeps avoidable function evaluations at a manageable level. Gains are most significant on hard to compute objective functions. Close coupling of workers leads to idling, SynchronizedGlobal never reaches the N times speedup theoretical limit. However, SynchronizedGlobal is usually not wasteful with resources, the usual overshoot encountered in the number of function evaluations is only 15%.

ParallelGlobal only loosely resembles the Global algorithm. While the main components are similar, their interactions are different. ParallelGlobal is characterized by low algorithmic efficiency and high utilization of the available resources. This is quite apparent when 16 workers need 16 times as many function evaluations, while the overall optimization process is slower than using the single thread Global. On some objective functions ParallelGlobal has an advantage, the low coupling permits effective use of the available computing power, 16x speedup is also a possibility. In conclusion, ParallelGlobal can produce much lower as well as much higher runtimes compared to Global, even on hard to compute objective functions.

The charm of Global is its simplicity and effectiveness in incorporating the available information into the optimization process. SynchronizedGlobal builds on this by offering a tradeoff between efficiency and increased resource utilization. ParallelGlobal heavily prioritizes resource utilization, to the point where efficiency can severely suffer. Depending on the task at hand each algorithm can be useful.

As the experiments showed, success of parallelism also depends on the characteristics of the objective function. Flat functions with relatively short local searches are the best to parallelize. Functions with a single or very few optima that can be reached from an easy to guess local search origin are hard to parallelize. As discussed, SynchronizedGlobal suffers much less from these "ill-conditioned" functions.

Another key point that stands out is the inherent inefficiency of parallel execution. Given the choice one should always tend towards a single chain of evaluations rather than simultaneous executions. For optimization faster CPUs are better than many CPUs with the same computing power. Parallelizing the evaluation of single samples is better than evaluating several of them at the same time. Every evaluated

sample provides information about the objective function that can be used to make further decisions. Faster incorporation of this information leads to more efficient optimization, parallel optimizers will always sacrifice a portion of this efficiency.

The primary problem in the current framework is that local searches which are already started cannot accept new information or give way to other local searches with better chances of finding the global minimizer. It is worthwhile to consider introducing a framework that allows prioritization of tasks that are expected to yield good results faster. As an example, consider ParallelGlobal on a function with a single global minimizer point. When the minimizer is first found we already suspect that it is the only minimizer in the vicinity, other areas should be explored before we try to refine very low level details of the function. Therefore, a policy can be set that stops any local search from using resources to find the minimizer that we already know of. Another interesting area is to rank ongoing local searches based on their probability of finding a new minimizer point and prioritize the best of them. These strategies can apply to both single-thread and multi-thread evaluations.

Previous works [28, 34] discussed probabilistic guarantees on the optimum value found by the Global algorithm. Recent works did not examine this aspect of the algorithm, the main goal is to find the supposed global optimum faster. The single thread versions are forced to balance between exploration and refining the promising origins until an optimum is found. Multi-threaded versions inherently do more exploration than necessary for finding the known global minimizers, they are verified with higher confidence. This aspect of optimization is not reflected in the current evaluation framework, especially for functions that can be solved with a single local search starting from any point. Although the multi-threaded versions lose efficiency, they provide higher confidence for knowing the global optimum.

Even with lowered efficiency, parallel execution of the objective function is useful when the goal is to find good function values, rather than solving the function to optimality. A real application of Global is the optimization of physical systems. When finite element methods are used, the models of these systems are numerically examined. Global can use measurements in the simulated systems to optimize the model parameters. A successful application is in the enhancement of plasmonic nanorod fluorescence [40, 41]. Optimization on the computationally expensive simulations using a single machine would lead to high runtimes. Using the integrated JPPF grid system and delegating the function evaluations to a cluster of computers has enabled the effective parallel evaluation of the expensive objective function.

Chapter 3

Problems and solutions in neural network verification

3.1 Introduction

Neural networks are widespread tools for solving a wide range of tasks. Generative AI is often used to create images, text, speech, and music. For most of these tasks the expected behavior is not well defined as there are many solutions perceived to be correct. Rigorous verification in these cases is not really meaningful due to the requirements being very hard or even impossible to define. Another widespread use-case for neural networks is classification, where an input is labeled according to its semantic meaning.

Some lesser known use-cases are compression of complicated logic, and approximation of computationally expensive algorithms. These tasks require much less input and output variables, the corresponding networks have a small(er) size. Expectations are much clearer on these tasks, networks can be examined whether they adhere to the expected functionality.

It is common knowledge, that even well trained neural networks can express peculiar behavior on seemingly innocent inputs. As Szegedy et al. [42] showed, small perturbations can have unexpected large effects on the outcome of classifier outputs. For example single pixel attacks try to find erroneous behavior when the adversary is only allowed to alter the value of a single pixel in the image. As Su et al. in [43] showed, for a lot of neural networks even single pixel attacks can be successful. With special training the sensitivity can be reduced, however sensitive inputs are not fully eliminated [44, 45], attacks are still possible.

Robustness to input perturbation is therefore an important property of neural networks. Formally we call a classifier network robust for an input if labeling is uniform in a given domain around the input. Robustness score of a network is defined over a set of inputs as the ratio of robust inputs. In practice the robustness ratio can be evaluated on the test or validation dataset accompanying the accuracy metric with a robustness score.

To acquire the robustness score for an actual network, the robustness property has to be practically decidable for an input. The task of verification is to solve a defined verification problem, and via the solution prove or disprove the robustness property on networks. Such verification problems and properties have to be formulated mathematically and tested with a suitable algorithm.

When formulated well, detecting sensitivity to perturbation becomes a verification problem. Given a perturbation amplitude, and a perturbation model – for example global random noise, geometric transformations or applying stickers – specifies the domain around an input. The verification problem also needs the property that should hold over the domain. The most common property is uniform classification, which usually translates to an output variable being maximal across the whole domain. If an input exists with a different classification, then the property does not hold and the input-domain pair is not robust. Other properties can be formalized too, for example multiple classes can be accepted in the domain as part of the definition of robustness. More complicated relationships are also expressible. A good example of a complex perturbation model is the verification problem aiming to verify safety of the Aerial Collision Avoidance System for unmanned aircraft [46]. Verification can be used to ensure that the system does not attempt aircraft turns in a trivially unsafe direction. However, the classification does not necessarily specify the strength of a turn and other aspects of the commanded movement, there is no strict classification to verify.

3.2 Contents and contributions

In this chapter I present advances on neural network verification, identify currently unsafe algorithms, and strategies to make them safer. The topic of neural network verification is introduced. Important distinctions are made between different types of verification algorithms, focusing on their ability to prove safety of neural networks. Strengths and weaknesses of bounding algorithms are also discussed.

In the next section I present the advances on neural network verification based on MILP solvers. First, definitions for the verification task are introduced. Then a brief overview is given on previous works regarding the underlying solver algorithm and effectiveness in solving the verification task. I explain the MIPVerify algorithm (by Tjeng et al.) in detail, a basis for showing issues in verification that apply to a wide range of implementations. Rounding errors in floating-point computations are presented, as it is relevant to the verification topic. Based on the modeled verification problems and the floating-point rounding errors the verification misalignment problem is introduced. Combining these topics, I present a trivial adversarial attack on neural network verifiers, with the exploit mechanism explained in detail. Computational results using different configurations of the exploit and the verifier are presented in support of the theory. Positive results in exploiting a state-of-the-art neural network verifier shows, that further research is needed, and authors of verifiers have to be careful about the strength of their statements.

My contribution towards the proof of concept exploit are experimentally finding erroneously evaluated verification problems, and arriving at the rounding error based exploit. Based on the results, I created a neural network implementing a computation with large rounding error. Conducting tests and evaluating results are mainly my work, as well as discovering the exploit mechanism in the examined implementation.

After presenting the minimal successful attack on verification, a setup simulating a realistic attack is introduced and is computationally evaluated. Possible alterations to the attack are mentioned that help evade detection of the exploit. Feasibility of an obfuscated attack is computationally tested by evaluating different obfuscated network configurations with multiple verifier configurations. Based on details of the exploit a defense is introduced. An explanation is given why the defense should be an effective tool, tests with different parameterization of the defense are conducted.

My contributions to the realistic attack are implementation of the exploit in an already existing network, generating minimal obfuscated adversarial networks, performing tests and evaluating the results.

In the final section I present the ongoing research for improved tolerance of inaccuracies in verification. First, the observed error phenomena are discussed with some potential causes. An attempt is made to defend against numeric issues introduced while solving the MILP models. Different methods for propagating rounding error information are considered, but a fully suitable algorithm is not reached. Another aspect of rounding errors in MILP models is also considered, where the model itself is incorrectly formulated for the task of proving statements required by verifica-

tion. An improvement in verification performance was achieved, where runtime was significantly reduced compared to the re-implemented MIPVerify algorithm.

I also show that the naive use of interval arithmetic to derive strong conclusions in verification is dangerous. More research is needed to find suitable algorithms that correctly bound the reachable outputs of neural networks.

My contributions in seeking improvements of verification based on MIPVerify are the following. Discovery of too strict bounds in MIPVerify's network model, leading to false negative results. Creation of a software framework to implement verifiers and evaluate tests, with attention to detail in communicating the strength of provided proofs. Replication of the MIPVerify algorithm and implementation of variants with efficiency and reliability improvements. Conducting tests and evaluating results on a rounding error propagation method. Tests on MIPVerify variations, discovering points of improvement and implementing a more effective version.

3.3 Approaches to verification

Verifiers come in different types, a defining feature is the class of verification problems that the verifier can solve. Linear and convex-nonlinear problems are usually solvable in practice, however problems with non-convex and non-linear constraints are most often infeasible to solve. The piecewise linear problem is a compromise between the two, where the search space is non-convex, but it is composed of many convex-linear problems. For this class there are many optimized and specialized algorithms, given enough computing power sizeable problems are feasible to solve.

Neural networks are by necessity non-convex structures, they necessarily have high expressivity, hence verification problems on neural networks come with non-convex spaces. The implication is that piecewise linear networks are a good compromise. They have the necessary expressive strength, while the generated piecewise linear problems can be solved with verifiers up to useful complexity.

A major strategy for verification is the use of optimizer algorithms. Proving bounds on the reachable set of outputs can be expressed as an optimization problem. As optimization is a very well developed area of research, many algorithms are available to solve piecewise linear optimization problems. They are solvable to optimality in finite time, in practice a lot of resources might be needed depending on the problem size. As a consequence verification problems can be solved in manageable time on smaller neural networks with piecewise linear activations. Verification

on sizeable networks usually requires a lot of resources, large state-of-the-art models are not feasible to solve in practice.

An approach for deciding a verification problem is proof by exclusion. Using interval arithmetic (IA) and given an interval bound for each input variable a region containing all reachable output configurations can be determined. This region is not a strict representation of the possible outputs, it is only a bounding box that is guaranteed to contain but is not limited to the results of the related mathematical expression. If the region does not contain a point that violates the examined property we can be sure that the property holds for the whole input domain. If the bounding space contains examples of violation and non-violation, we cannot decide whether the property holds. Either the actual output space contains violating examples, or due to overestimation the bounding box introduces them, and the property in reality holds. The problem can only be decided when an input space is mapped to a fully non-violating, or a fully violating output space. When the property does not hold on any point of the bounded space, every input is guaranteed to be a counterexample. When the property does hold on the input space, the amount of overestimation controls whether deciding the problem is feasible.

The simple interval arithmetic bounding algorithm suffers from excessive overestimation, in most cases the use of which is hindered due to the interval dependency problem. Combining intervals that are not independent causes overestimation, as it is assumed that they can take any value independent of each other. The typical example is that given $x, y \in [-2, 2]$, and $x = y$, the result is $x * y \in [0, 4]$. However, the $x = y$ constraint is ignored, the computed result is $x * y \in [-4, 4]$. In neural networks the actual dependencies are very complicated, exactly representing result intervals and preventing the dependency problem is not feasible.

A more sophisticated use of interval arithmetic is the branch and bound (B&B) algorithm. The B&B algorithm mitigates overestimations by splitting the undecidable intervals to smaller ones in hope that they become decidable. Smaller input regions lead to smaller overestimations and could produce decidable outputs. If the output of subdivided interval boxes have a uniform classification, they are already known to be decided. To exclude the possibility of counterexamples on the whole domain all boxes have to be proven free of them. Finding a uniform box of counterexamples proves the violation by example immediately. Unfortunately the B&B algorithm also suffers from the interval dependency problem introduced by the use of interval arithmetic. Because of overestimation and combinatorial explosion of the subdivided boxes the Branch and Bound algorithm is not considered feasible to verify neural networks.

3.4 Neural network verification is not solved

Image classification networks are often sensitive to inputs that contain low amplitude global noise. The modifications are usually not detectable by human eye, they are perceived as camera noise, compression noise, or characteristics of the scene itself like fog. Adversaries may tamper inputs and attack networks with this noise model, hindering the classification process. Simple variability in the inputs might also lead to wrong classifications in a non-hostile setting. To detect and train against erroneous classifications, verification using these models has to take place.

Verification problems can be formulated in many ways, we chose to adopt the formulation from Tjeng et al. in [47]. For an input x let $\mathcal{G}(x)$ denote the set of inputs considered similar to x. The $\mathcal{G}(x)$ set is commonly constructed as a sphere around x in a metric space defined by a suitable vector norm. Let $\mathcal{X}_{valid}$ denote the set of valid inputs, for image classification problems it is usually the $\mathcal{X}_{valid} = [0, 1]^m$ box in m dimensions, where m is the number of pixels, and pixels are normalized to the $[0, 1]$ interval. The $\mathcal{D}(x)$ input domain is defined as the $\mathcal{G}(x) \cap \mathcal{X}_{valid}$ non-empty intersection.

The problem definition specifies uniform labeling on $\mathcal{D}(x)$ as the condition for a sample to be robust. Let $f(x) : \mathbb{R}^m \to \mathbb{R}^n$ denote a neural network as a function, and let $\lambda(f(x))$ denote the label of an x input based on the network output. The label is defined as the index of the maximal element in the $y = f(x)$ output vector; $\lambda(y) = \underset{i}{\text{argmax}}\ y_i$. Finally, the requirement of uniform labeling is expressed. Given the x sample and the $\mathcal{D}(x)$ domain, for every $x' \in \mathcal{D}(x)$ input the $\lambda(f(x)) = \lambda(f(x'))$ equality should hold.

Conversely, if an $x' \in \mathcal{D}(x)$ input exists such that $\lambda(f(x)) \neq \lambda(f(x'))$, then the uniform labeling property is proven to be violated on the domain. The given formulations can be composed into a constraint satisfaction problem (CSP) or a global optimization problem. If a problem formulation can be solved the existence of x' can be decided, which either proves or disproves robustness of $f(x)$ for the x input and the $\mathcal{D}(x)$ domain.

Given the formulated problem there are many algorithms that can tackle it in various ways, and to different extent. It is easy to see that half deciding methods can disprove robustness by example. The simplest one is random sampling, where by trial and error finding a sample provides proof by example. A bit more sophisticated method is to search for a feasible x' using a swarm of local searches. Using interval arithmetic disproving the existence of an x' counterexample can be attempted, but

naive approaches fail because of the interval dependency problem, and B&B fails because of combinatorial explosion in the number of sub-problems.

A promising approach is the use of mixed integer linear programming (MILP) solvers in verification. These model based approaches do not suffer from overestimation, given some restrictions the neural network can be modeled exactly. The Reluplex method by Katz et al. [48] describes a ReLU formulation based on linear constraints and a splitting rule to exchange the convex overestimation of ReLU units for multiple sub-problems. Reluplex utilizes a SAT solver to generate and evaluate the feasibility of logical expressions that describe the verification problem. The solver can also help with branching and eliminating branches of the evaluation tree. The Reluplex calculus is supplemented by an LP solver to introduce tighter bounds on variables, greatly reducing the search space.

To find the adversarial example with the smallest perturbation Carlini, Katz, Barrett, and Dill [49] proposed a naive logarithmic search based on satisfiability analysis by Reluplex. The runtime of Reluplex is acceptable but already significant, increasing it by another order of magnitude would severely limit its usability.

Based on the results of Reluplex in [50] Lomuscio and Maganti introduced a MILP formulation that models the verification problem exactly. The two approaches treat continuous variables in the same way. While the SAT based model does not encode binary variables directly, the MILP model includes them as part of the problem model explicitly. As MILP solvers are quite mature, deciding the verification problem and finding the closest possible adversarial example at the same time is feasible. The runtime of MILP based verification is in the same order of magnitude as the Reluplex algorithm, much better performance than the Reluplex based binary search.

In [47] Tjeng et al. based on the MILP model described in [50] developed an iterative approach, where the MILP model is built up layer by layer. Using various tools the ReLU units are filtered reducing the amount of modeled neurons, hence reducing the runtime of the verification process.

As the mentioned works show, effective modeling of verification problems reduces the evaluation time to manageable levels even on networks large enough to be useful in practice. Completeness of MILP solvers implies that the verification problems are solved to optimality and with a valid solution, hence verification problems encoded in the given mathematical formulation can practically be decided. In reality, details of executing a verification task conceal the real challenges, naive use of MILP solvers does not yield the confidence levels required for verification.

Neural network verification is not a solved problem.

In strict mathematical sense most verifiers are not sound, nor complete. Usually the verification problem is not well defined. As it will be discussed, floating-point operations are not linear even though they are usually modeled as such. A sound algorithm either has to model the performed computations exactly matching how they are executed in practice, or it has to overestimate the possible outputs to cover any possible evaluation scenario. When overestimation happens the algorithm cannot be complete, some verification problems simply cannot be decided with a given level of overestimation. No algorithm is known that is capable of exactly modeling floating-point computations and is feasible to execute, hence verification of neural networks is still an open problem.

Nonlinearity of floating-point operations gives an unexpected focus to the order of operations. Widely used neural network definitions do not specify the order of operations, but verifiers usually implicitly assume one. If this differs from the actual order of operations during execution, then the used system slightly differs from the verified one. Similarly to how a small perturbation on the network inputs can have large implications on the output, slight differences in the computations during the verification process can lead to different results.

The rest of this chapter discusses issues in modeling and numerics in verification.

3.4.1 MIPVerify

MIPVerify by Tjeng et al. [47] is a state-of-the-art neural network verifier based on mixed integer linear programming (MILP) solvers. Unlike earlier algorithms, MILP based verifiers are capable of evaluating sizeable verification problems with runtimes in the order of seconds to minutes, it is a great accomplishment in verification technology. Solving MILP problems is a widely studied subject, it is already optimized for fast and large scale computations. These solvers are a reasonable choice to be the basis for verification.

A limitation of MILP based algorithms comes from the linear MILP constraints. Models with linear constraints and integer variables define a set of linearly bounded convex subspaces. Solving a subspace usually scales linearly with problem complexity, but depending on the verification problem the number of subspaces can be exponential. MIPVerify models contain linear and binary variables, reducing the number of binary variables is therefore an important aspect of practical verification.

Let $x_j^{(i)}$ denote the output of the jth neuron in the ith layer, and for $i = 0$ the jth value of the input vector. Neuron outputs are defined as the linear combination of values from the previous layer and the ReLU activation on the resulting scalar value. Weighted mixing of values with a bias is equivalent to the simple linear expression $z_j^{(i)} = b_j^{(i)} + \sum_k w_{j,k}^{(i)} \cdot x_k^{(i-1)}$, which is directly representable as a MILP constraint. The ReLU activation produces the output of a neuron $x_j^{(i)} = relu(z_j^{(i)}) = max(z_j^{(i)}, 0)$. This expression cannot be modeled directly.

The following constraints are needed for a neuron to express the ReLU activation, where a is a binary variable, l and u are the lower and upper bounds of the z input variable, and x is the ReLU output. For a verification problem the $l_j^{(i)}$ and $u_j^{(i)}$ bounds are constant, they can be calculated or approximated before formulating the ReLU unit.

$$z \leq x \tag{3.1}$$

$$0 \leq x \tag{3.2}$$

$$x \leq z - l \cdot (1 - a) \tag{3.3}$$

$$x \leq u \cdot a \tag{3.4}$$

To see how these constraints function, let's consider the following scenarios. Given a negative z input the first constraint becomes redundant, as the second constraint already forces the output to be non-negative. If a is 1, the third constraint reduces to $x \leq z$. Combined with $z \leq x$ the $x = z$ equality is implied, a contradiction with the $0 \leq x$ constraint, as z is negative. Therefore, given a negative z input $a = 0$ is the only possibility, where the fourth constraint becomes $x \leq 0$, and in combination with $0 \leq x$ it reduces to $x = 0$. We have established, that a negative z input results in an $x = 0$ output, which is required by the ReLU function. The $a = 0$ value renders the third constraint ineffective, it becomes $x \leq z - l$. As l is a lower bound for the value of z, $z \geq l$ is satisfied, hence $z - l$ is not negative. Since we already established that $x = 0$, the $x \leq z - l$ constraint where $z - l$ is not negative has no effect.

Now let's consider a positive z input. The second constraint becomes redundant, as the first already forces x to be positive. If a is 0, the $x \leq u \cdot a$ constraint becomes $x \leq 0$, which is in contradiction with $z \leq x$ and z being positive. Therefore, $a = 1$ is the only possible value, which reduces the third constraint to $x \leq z$. Combined with $z \leq x$ we have established that a positive z input results in the $x = z$ output, which is required by the ReLU function. The fourth constraint is ineffective, as u is an upper bound of z, and by $x = z$ it is an upper bound of x, therefore $x \leq u$ is satisfied.

Using the $x_j^{(i)}$, $z_j^{(i)}$, and $a_j^{(i)}$ variables and the four constraints to represent ReLU units, we can represent the full neuron and create a MILP model of all neurons in a neural network. Note that the l lower bound and u upper bound is needed to model a neuron. Also note that finding the exact bounds is not necessary, validity of the bounds is enough to construct a correct model. However, an overestimation that is orders of magnitude larger than the real bound can cause numeric issues, we discuss this in the next section.

The MILP model of a neural network contains a binary variable for every neuron. In sizeable networks the number of neurons is at minimum in the thousands, more likely in the tens of thousands. As mentioned above, the amount of subspaces created by the combinations of binary variables significantly slows the optimization process.

It is easy to see that some ReLU units can be simplified in the model. A $u_j^{(i)}$ upper bound that is negative implies that the output is always zero, the neuron is inactive. Similarly, an $l_j^{(i)}$ lower bound that is positive implies that the output is always equal to the input, the neuron is active. In both cases the ReLU activation acts only on one side of the piecewise linear function, we call these neurons *stable*. When $l_j^{(i)} \leq 0$ and $u_j^{(i)} \geq 0$ are both satisfied, the neuron is *unstable*, therefore modeling of the ReLU unit is necessary. If the bounds are estimated instead of a tight calculation, stability of the ReLU might not be unknown. For the purpose of efficient modeling of a network the only interesting question is whether stability can be proved.

Stable neurons can be simplified in the MILP model. When $z_j^{(i)}$ is always negative in the verified domain the $x_j^{(i)}$ output is known to be zero. Instead of introducing a model variable the *zero* constant can be used. When $z_j^{(i)}$ is positive in the domain the output is known to be equal to the input, we can use the $x_j^{(i)} = z_j^{(i)}$ equality or simply use $z_j^{(i)}$ in place of $x_j^{(i)}$. In both stable cases the $a_j^{(i)}$ binary variable is unnecessary, and can be omitted to reduce the problem complexity.

There are multiple strategies to obtain the necessary $l_j^{(i)}$ and $u_j^{(i)}$ bounds. The simplest approach is using interval arithmetic to approximate the reachable intervals. This comes with several problems. Interval arithmetic might need to overestimate the $\mathcal{D}(x)$ domain, as only independent variables can be represented accurately. In further layers the dependency problem will cause large overestimations, which severely reduces the efficiency of filtering. The same large overestimated bounds can cause numeric issues for the MILP solver. Interval arithmetic can be effective when the dependency problem is limited by independent input intervals, or when dependent calculations have a relatively small effect.

Another possibility to obtain the bounds is the use of MILP solvers. When the neural network is modeled, we can use an iterative algorithm that always calculates the bounds in the next layer. The approach utilizes the already modeled previous layers. If the model is built up to and including the $(i-1)$th layer, then solving for the $l_j^{(i)} \rightarrow \texttt{Min}$ and $u_j^{(i)} \rightarrow \texttt{Max}$ objective functions will provide sharp bounds.

Although these smaller MILP problems are much easier to solve, they still represent a large portion of the verification problem. As a compromise the LP relaxation of MILP problems can be used. By relaxing the integer constraint on the binary variables the MILP model is transformed into a single linearly bounded subspace, an LP problem. This wider LP model can be solved in much shorter time and provides an overestimated but still decent solution.

With another use-case for interval arithmetic a further improvement on the efficiency of model optimization can be made. As discussed, using it on the whole network would quickly lead to problems, however in layer-to-layer propagation it is still a valuable tool. Calculating an estimate for the next layer has a cost on the same order of magnitude as simply evaluating a layer. The calculated bounds will be able to decide stability for most neurons at a low cost, the more expensive methods only have to be used for a fraction of the layer bounds.

As MIPVerify only creates a representation for the network part that is necessary to solve the given verification problem, it can be evaluated significantly faster. The MILP model containing the network representation starting from the $\mathcal{D}(x)$ domain is further extended with tooling to evaluate the verification problem. To verify the existence or the absence of an adversarial example a last optimization problem has to be formulated that encodes the verified property. MIPVerify offers multiple verification targets. The standard problem formulation is "using the ∞-norm find the input closest to the perturbed input that differs in classification".

As an $x^{(0)}$ perturbed input vector closest to the x original input has to be found, the objective function must be $||x^{(0)} - x||_\infty \rightarrow \texttt{Min}$. The ∞-norm difference of the two vectors is equivalent to $\underset{j}{max}|x_j^{(0)} - x_j|$, an expression that we can formulate. The absolute value function could be expressed with a binary variable, however in combination with the objective function an additional ϵ continuous variable is enough. For every element of the input vector the $\epsilon \geq x_j^{(0)} - x_j$ and $\epsilon \geq x_j - x_j^{(0)}$ constraints are added, hence ϵ is not less than the maximal difference. It provides an upper bound for the objective function. The objective function of the final MILP problem is set to $\epsilon \rightarrow \texttt{Min}$, which forces ϵ to take its minimum possible value, therefore ϵ has to be

equal to the maximal element in the $|x^{(0)} - x|$ difference vector. Because the objective function is minimizing $\epsilon = ||x^{(0)} - x||_\infty$, it formalizes the required objective.

Since $x^{(0)}$ is not constrained and $|x^{(0)} - x|$ has the trivial minimum value of 0, evaluating the model with the given objective would always find this unsatisfactory solution. To find the closest counterexample to x, MIPVerify constrains $x^{(0)}$ to be an adversarial input.

Let's consider the ℓ label of the original x input, and the $x_j^{(n)}$ output variable corresponding to the ℓ label. When $x^{(0)}$ is adversarial, the $x_j^{(n)}$ output variable is not the maximal in the output vector. A larger $x_k^{(n)}$ output exists corresponding to the $\ell' \neq \ell$ label of the perturbed input. Let's model an extra layer of ReLU units as $y_k = relu(x_k^{(n)} - x_j^{(n)})$ for each k, where $k \neq j$. The y_k variables indicate whether an output exceeds the one corresponding to ℓ. Any greater than zero y_k value proves the existence of an adversarial example. To constrain the model to only contain examples where a positive y_k value exists, a $y = \sum_k y_k$ variable is introduced. If any y_k value is positive, y will also be positive. If y is zero, then all y_k values have to be zero. Therefore, a positive y value exactly corresponds to the input being adversarial.

To constrain the MILP model so that all feasible solutions are adversarial, the $y \geq y_{min}$ constraint is added, where y_{min} is a small positive constant. If the MILP problem has no feasible solution with this constraint, then an adversarial example cannot exist, therefore the original input is proven safe. If there are feasible solutions, they have to be adversarial and the objective function ensures that the optimum corresponds to the adversarial input closest to x. In this setup feasibility indicates whether the domain contains an adversarial input. The setup ensures that if a solution exists, it will coincide with the closest possible adversarial input to the original input, measured with ∞-norm.

With the final MILP model in hand, all models are defined for a verification problem. MIPVerify can generate solvable models that ensure complete verification for ReLU – or in general piecewise linear – networks. To solve the MILP problems, MIPVerify uses Gurobi [8] by default, a state-of-the-art closed source optimizer. In our experience it is currently the best MILP solver for neural network verification.

3.4.2 Issues with floating-point computations

Floating-point number representation is the single most widespread tool for representing and executing arithmetic operations with real numbers. Most hardware supports some version of the IEEE 754 standard [27], that specifies the representation

and operations on binary floating-point values. The most commonly implemented word sizes are 32 and 64 bits. For an example let's consider the 64-bit representation. Floating-point values are defined by the generic formula $s \cdot b^e$, where s is the signed significand, b is the base and e is the exponent. Computers use binary numbers, hence $b = 2$, s and e are stored in the available 64 bits as two separate binary numbers. By the standard, s is 52+1 bits wide, where a bit explicitly encodes the sign, and e is stored on the remaining 11 bits. The IEEE 754 floating-point representation contains further optimizations. If we consider the unsigned binary values of the s_{sign}, s and e bit-fields, then the encoded f_{64} value is computed as

$$f_{64} = (1 - 2 \cdot s_{sign}) \cdot \left(1 + \frac{s}{2^{52}}\right) \cdot 2^{e - e_{bias}}, \tag{3.5}$$

where $e_{bias} = 2^{11-1} - 1 = 1023$ is a constant specific to the number of bits that represent e. The resolution[1] – the space between neighboring representable values – from $f_{64} = 1.0$ to the next number is $\epsilon = \epsilon_{64}(1.0) = 2^{-52} \approx 2.22 \cdot 10^{-16}$. This value is also referred to as the "machine epsilon". When representing $f_{64} = 1.0$, the positive value implies $s_{sign} = 0$. Given the 2, 0.5, or other exponent multipliers 1.0 is not representable, therefore $e = 1023$. This simplifies the equation to

$$\begin{aligned} f_{64} &= (1 - 0) \cdot \left(1 + \frac{s}{2^{52}}\right) \cdot 2^{1023-1023} \\ &= 1 + s \cdot 2^{-52} \\ &= 1.0\ . \end{aligned}$$

Let's take $1 + s \cdot 2^{-52} = 1.0$ and rearrange it for s, the result is $s = 0$. The smallest value that is representable and greater than 1.0 comes from substituting $s = 1$ to Equation (3.5), with the other values unchanged the result is $f_{64} = 1 + 2^{-52} = 1 + \epsilon_{64}$.

The value of s due to its binary representation is integer. If representing a number would require a non-integer s, rounding has to take place. The standard requires correct rounding, therefore any number between 1 and $1 + \epsilon/2$ is rounded down to 1, and values above are rounded up to $1 + \epsilon$.

Now consider the value of the $10^{20} + 2025 - 10^{20}$ expression. Using simple algebra we can quickly arrive at the solution of 2025. However, f_{64} operations produce a different result as the expression is translated to $(10^{20} +_{f64} 2025) +_{f64} -10^{20}$, where $a +_{f64} b$ includes rounding to a representable f_{64} number. Although both 2025 and 10^{20} are possible to represent in f_{64}, the $10^{20} + 2025$ sub-result is not. In fact, the sum

[1] A similar and often confused concept is the maximum roundoff error around $f_{64} = 1.0$, which is $\epsilon/2 = 2^{-53} \approx 1.11 \cdot 10^{-16}$.

is closer to 10^{20} than the next representable number, as $\epsilon_{64}(10^{20})$ is 16384. Therefore, the next expression in the evaluation chain is $(10^{20}) +_{f64} -10^{20}$. The final result is 0, and not 2025.

As we can see, roundoff errors can limit the accuracy of our computations. Now let's consider the $(10^{20} +_{f64} -10^{20}) +_{f64} 2025$ expression. The first addition evaluates to 0, and the remaining expression is $(0) +_{f64} 2025$, which evaluates to 2025. Roundoff errors are not only limiting the accuracy of computations, the introduced nonlinearity yields different results based on the order of operations.

In general, this phenomenon occurs with addition when the ratio between the addends is larger than $2/\epsilon$, which is the reciprocal of the relative roundoff error. With an a/b ratio lower than $2/\epsilon_{64}$ the summation does satisfy $(a +_{f64} b) > a$, but the result can still experience a large rounding error.

3.4.3 The verification misalignment problem

State-of-the-art verifiers are capable of verifying sizeable networks with the help of mixed integer linear programming solvers. MILP problems describe an algebraic construct where perfect precision variables and computations are assumed. Almost all implementations of MILP solvers use floating-point arithmetic to represent and solve problems. Given the nonlinearities introduced by floating-point arithmetic the MILP solvers operate on different constructs than the verification algorithms assume. The unmodeled nonlinearities lead to small errors in the optimization process which can blow up in size and produce large errors.

The verification misalignment problem occurs when a verification algorithm assumes a mathematical description of the neural network or the MILP solver that is not true. In practice both are affected by the false assumption that the calculations operate on accurate arithmetic over real/rational numbers. Calculations are modeled as exact algebraic expressions, while in reality the computations happen on floating-point arithmetic. When a network model is evaluated the specific floating-point representation is often implied from the model parameters. Optimizations on the lower execution levels can also distort computations. If parallelization is involved the execution order usually becomes undefined or changes easily depending on the exact execution environment. Other tools for acceleration involve the use of lower precision algorithms to compute complicated mathematical functions. Again, accuracy of computations is not defined by the model, only in combination with the computing algorithm and environment.

Some implementations of MILP solvers utilize varying precision floating-point representations [12], usually they have a maximum precision of 256 bits. Although this reduces the rounding error – to be exact $\epsilon_{256}(1) = 2^{-236}$ – the problem is not solved, at best postponed. In reality any discrepancy can be used to hinder verification. Also, 256-bit floating-point numbers have no widespread hardware support, therefore using them comes with a large performance cost.

Some MILP solvers are implemented on rational arithmetic [51]. This implementation is exceedingly slow as operations with rational arithmetic are not supported by hardware. Surprisingly, naive use of the rational arithmetic MILP solver also presents a verification misalignment problem. Since rational arithmetic has infinite precision, every computation result is exact. This can cause smaller output envelopes than it is possible with floating-point arithmetic because of the rounding behavior. Unless the model accounts for the rounding errors in the floating-point computations, rational arithmetic can easily miss outputs that are possible to reach during normal use of the network.

3.4.4 A trivial adversarial attack

Let's see a demonstration of the discussed numeric issues in form of an attack on numerically vulnerable methods. A small neural network is enough to induce the explained numeric error and demonstrate the exploit as part of a classification task. On Figure 3.1 the trivial adversarial network is shown, where neuron activations are in the $x \in [0, 1]$ input range. It performs the classification task where the input range is categorized as below and above 0.5. As the standard setup for classifiers has an indicator output for every class, the y_1, y_2 output layer is introduced, where a maximal y_1 signifies $x < 0.5$ and a maximal y_2 means $x > 0.5$.

The network first separates the classified regions in value, then introduces the $\omega + 1 - \omega$ computation. Neuron A performs the separation, its output is $relu(x - 0.5)$, the ReLU function with the breakpoint at 0.5. The next layer consists of neuron $B_1(x) = relu(\sigma \cdot A(x) + 1)$, and $B_2 = relu(1) = 1$. The negative σ is a tuning constant which ensures that the levelset of $B_1(x)$ covers the entire $[0, 1]$ interval. Since $A(x)$ is at maximum 0.5, σ has to be below -2.

Neuron C is the key point, where the $\omega + 1 - \omega$ calculation is enforced. The activation is $C(x) = relu(\omega \cdot B_1(x) - \omega \cdot B_2(x) + 1)$. With the usual definition of neural networks the sum is calculated in the $\omega - \omega + 1$ order, resulting in no rounding error. This setup not only introduces the large rounding error, but in fact it produces

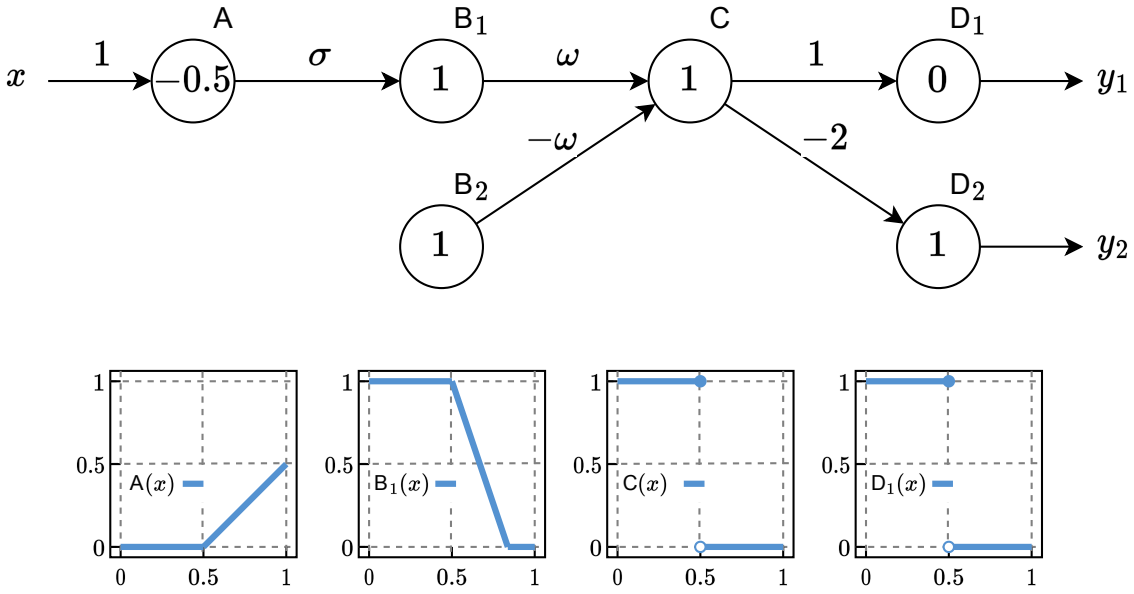

Figure 3.1: *Trivial adversarial network exploiting the $\omega + 1 - \omega$ type computation. Circles represent neurons with their bias inside the circle and ReLU activation. The bottom graphs show the output of neurons over the $x \in [0, 1]$ input range.*

a binary output on neuron C. The two possible values are 0 and 1, given a 64-bit or smaller floating-point representation. Since the contribution of B_2 to the sum is always $-\omega$, we only have to discuss the different values of B_1 visualized on the bottom graph of Figure 3.1.

If the value of B_1 is lower than 1, the expression $\omega \cdot B_1(x) - \omega \cdot B_2(x)$ results in a large negative value that always dominates the smaller bias of C. Hence, the output of C has to be zero due to the ReLU activation. If the value of B_1 is equal to 1, the expression $\omega \cdot B_1(x) - \omega \cdot B_2(x)$ evaluates to exactly zero. Since the activations of the previous layer cancel out, only the bias of neuron C remains, which is 1. The value of $C(x)$ is therefore binary.

The neurons D_1 and D_2 provide the y_1 and y_2 outputs for classification. With an easy transformation from the binary output of neuron C the label $y_1(x) = C(x)$ is 1 if $x < 0.5$, when $x > 0.5$ the value of $y_2(x) = 1 - C(x)$ becomes 1. Due to the binary nature of $C(x)$ the non-active output always becomes 0.

Other neural network implementations might use the $1 + \omega - \omega$ or $\omega + 1 - \omega$ computation order, which renders the presented network ineffective. For these environments a slightly larger network can ensure that the order of computations is fixed, regardless of evaluation order within a layer.

To verify the problematic behavior induced by the trivial adversarial network the MIPVerify algorithm was tested, using the Gurobi [8] and CPLEX [9] proprietary MILP solvers, and the GLPK [11] open-source MILP solver. Evaluations on different combinations of $\sigma \in [-15, -2]$ and $\omega \in 2^{54}, 2^{55}, .., 2^{70}$ parameterizations were conducted, with 500 random values of σ. MIPVerify was tasked with verifying whether the $x = 0.75$ input had a counterexample in a radius of 1, which covers the whole input range. Clearly, $x = 0$ and $x = 1$ prove the existence of two different classifications, hence the answer should always be "yes, there exists an input with different classification in the $\mathcal{D}(x)$ domain". Yet, MIPVerify for all parameter combinations using any of the three solvers have reported the problem as "infeasible, there are no other classifications on the input range".

We showed a successful attack on the MIPVerify algorithm, or more generally on search algorithms naively using MILP solvers for verification.

At this stage the finding can come from two different sources. MIPVerify applies a filtering step to reduce the problem size presented to the MILP solver. In our experiments we noticed that on many occasions the output of neuron C was simply bound to the zero constant, due to the filtering step. This explains why our attack is so reliable, the derived numeric models simply don't allow for a solution where C has a non-zero output. In later experiments we see that even when neuron C evades purging the MILP solver does not find a feasible solution to the numeric model. Hence, it fails to prove the existence of an adversarial example, and reports the verification task to have no adversarial solution.

3.4.5 New backdoors in existing networks

Findings on the trivial adversarial network prove that verification is vulnerable. However, when the attack is scaled up from the 6 neuron sterile environment, results could differ significantly. Hence, tests are performed on a network that resembles real use-cases. An existing network with useful functionality is utilized, that would be too big to be verified with methods prone to combinatorial explosion. Based on the trivial adversarial network a backdoor is created in the host network.

As basis for testing we chose *WK17a* [52], a trained MNIST network that was already evaluated in the paper by Tjeng et al., where it is called LP_d-CNN_A[47]. The MNIST dataset contains images of handwritten digits. The task is to classify images based on which digit from 0 to 9 appears on them [53]. The network has 784 inputs for the 28x28 pixel gray scale images, with the value of each pixel being in the

$x_i \in [0, 1]$ range, and it has 10 outputs for the digits 0-9. The *WK17a* network has the ConvNet architecture which consists of four layers. The first two are convolutional layers with 16 and 32 filters respectively, and stride 2. Then comes a fully connected layer of 100 neurons and lastly the fully connected output layer with 10 neurons. The hidden layers use ReLU activation, while the output layer uses the Sigmoid activation. Since the Sigmoid function is a strictly increasing monotone function, omitting the activation does not change the order of outputs when sorted by value – which is the property used for determining the result label. Verification that is only concerned about the assigned labels can ignore the Sigmoid activation in the output layer.

To obtain a network that contains the trivial adversarial network and uses it as a backdoor, the architecture of *WK17a* is extended as shown on Figure 3.2. By embedding the adversarial network into the original structure, the backdoor behavior can be controlled separately from the original functionality. The network is extended such that different inputs to the backdoor either result in the original *WK17a* behavior, or an alternate mode. The goal of the alternate mode is to always provide close-by inputs that have alternative classifications, so that the network always contains an adversarial example closer than a fixed distance.

Based on the paper of Tjeng et al. the verification search radius of 0.1 was selected with the ∞-norm[2] distance measure. A condition is needed to activate the backdoor behavior, the $x_1 = x_{topleft} >= 0.05$ condition is chosen. The first pixel – which is in the top left corner when displayed – has 0 value on all the test inputs, it is an ideal target for a backdoor trigger. By setting the activation threshold to half of the 0.1 search radius, it can be ensured that none of the tested inputs fall into the activated region, yet all of them are close to the unmistakable adversarial region. In fact, for every test sample half of the search space falls into the active backdoor region.

For the two states of the adversarial switch two very distinct outputs are needed that always differ in classification, so that the resulting network has an adversarial example for every input. To achieve this, the *rotate* operation is implemented on the output vector. Given an x input where the adversarial switch is off and a close by input x' where the switch is on, $\lambda(x') = \lambda(x) + 1 \bmod 10$. In simple terms, for every $digit$ label originally assigned by the network, the activated backdoor returns $digit + 1$, in case of 9 it returns 0. This behavior ensures that a close by input always exists that has a classification different from the original.

[2]Also called max-norm, the vector norm results in the vector element with the maximal absolute value; $||v||_\infty = \max_i |v_i|$.

[3]*WK17a* is also known as LP_d-CNN_A in [47].

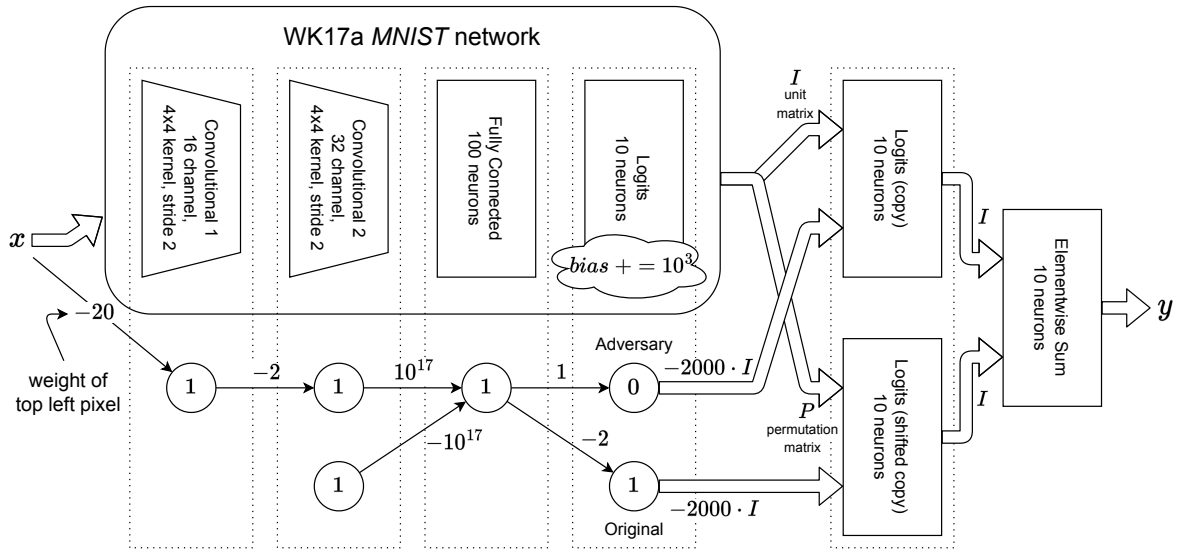

Figure 3.2: *WK17a-adv adversarial network, based on WK17a*[3]*, integrated with the network shown on Figure 3.1. Dotted areas denote neurons encoded in the same layer.*

To achieve the described behavior, the small adversarial network is embedded in *WK17a* as shown on Figure 3.2. The first convolutional layer is extended with a filter to encode the $relu(-20 \cdot x_1 + 1)$ calculation. The second convolutional layer is extended with two filters for the B_1 and B_2 neurons of the trivial adversarial network on Figure 3.1. The B_2 neuron is encoded with a constant zero filter and a bias of 1. The B_1 neuron is encoded with strategically placed filter weights, such that one of the output values on the output channel yields the correct result. The third fully connected layer is extended with neuron C, all new weights are zero except the two that connect the outputs of B_1 and B_2 to C. Weights are placed such that only the correct calculations are picked out from the convolution output. Connections introduced to already existing network parts have zero weight, implementation of the original behavior should not be modified, only extended.

The logits layer outputs the result vector in the original network, in *WK17a-adv* it is extended with the D_1 and D_2 neurons. For the adversarial network additional layers are needed with ReLU activation to perform the shifting and switching behavior. As the logits activation is not supported by MIPVerify and the D_i neurons require ReLU activation, the layer must be changed to use ReLU activation. Naively switching to ReLU activation would map the negative Sigmoid outputs to zero. To prevent the information loss the occurring outputs are measured, it is concluded that raising values by 1000 prevents the cutoff. Increasing the bias value does not change classification of the output vector, for verification it has no significant effect.

The first extra layer encodes two versions of the original output vector. The first half is simply a one-to-one copy, the second half is the rotated copy produced by a permutation matrix. The switch neuron selecting the original behavior is connected to all neurons of the rotated copy with a large negative weight. The $1 \cdot -2000$ value forces the unwanted values in the layer to be negative, which are mapped to zero by the ReLU activation. The neuron selecting the adversarial behavior eliminates the original copy and preserves the rotated copy.

As either the neuron labeled with *Original* or *Adversary* can be active, exactly one copy in the first extra layer is the zero vector. The second extra layer performs an element-wise sum of the two vector copies. As exactly one of them is zero, the sum simply yields the non-zero vector. In effect, either the original logits output, or the rotated logits output is yielded as the network result.

Validation on the *WK17a-adv* network was performed to ensure the so far described behavior. Evaluation of test set samples resulted in the exact same output vectors as *WK17a* produced. Test set samples have the $x_{topleft} = 0$ pixel, therefore the backdoor does not activate. The test set with $x_{topleft} > 0.5$ modified samples yielded the rotated output in every case, the backdoor was successfully activated.

After validating the adversarial functionality the main test was executed, evaluation of the *WK17a-adv* network with a state-of-the-art verifier. The verifier of choice is *MIPVerify*, configured with *Gurobi*. The evaluated verification problem – apart from the network – replicates the results on LP_d-CNN_A found in [47]. All test set samples are evaluated for adversarial behavior with 0.1 search radius using the ∞-norm. Out of 10000 samples 438 were found to be vulnerable, which yields 4.38% adversarial error. The exact same robustness is reported for LP_d-CNN_A, the extended network was perceived as if no backdoor was present. Validation of the network proved that the backdoor is indeed present, the problem lies in the verification process.

We also tried other configurations and verifiers with similar results, more details can be found in [54]. Tests use the simple one pixel switch backdoor activation, as it is an easy demonstration of the adversarial behavior. Since 50% of the search space is adversarial, it is easy to find by heuristic searches based on random sampling. To defeat these algorithms and to properly hide the adversarial activation, a more complex switch can be used. A switch consisting of the top left 3x3 area activated in a checkerboard pattern was also tested. The resulting adversarial space is only $1/512$ times the search volume, which has proved to be too difficult for the random methods to guess. More complicated switches are also possible that further hide the

backdoor. By feeding the output of a neural network tasked with detecting a pattern or symbol into the switch, the backdoor activation can be arbitrarily diverse.

3.4.6 Obfuscation

A trivial adversarial network crafted into a universal adversarial switch showcased on Figure 3.2 is a powerful tool in the hands of attackers, naturally a good defense sought after to mitigate exploits. In its current form the switch structure is easy to detect by observing the large weights and the characteristic subgraph that coordinates the computation. As a first step, a real adversary would probably try to disguise the switch, which is achievable in several ways. The easiest approach is to apply the large weights in several smaller steps, by implementing a series of multiplications. If more neurons are used in consecutive layers with the $\omega_1 \cdot \omega_2 \cdot \ldots \cdot \omega_n = \omega$ effective computation instead of a singular ω, we can spread the large weights. The ω_i weights can have any value except for zero, only the end result matters. The $\omega = 10^{17}$ value in the example on Figure 3.2 is only by choice, in fact a large range of values result in the "desired" numeric errors. The important factor is the equality of the ω values coming from the B_1 and B_2 neurons. Other characteristics of the switch can be obfuscated as well. Neurons with a constant output can be achieved trivially using the ReLU cutoff and the desired bias value. Perfectly canceling variables coming from multiple paths in the graph can also lead to an effectively zero input neuron. For better obfuscation a deep and/or wide network is needed, which is common in neural network technology, the network sizes that can be verified are still limited in comparison. Increasing the graph complexity by adding connections that cancel out should also be possible to further obfuscate the switch network. A simple measure of graph topology might not be enough to detect adversarial sub-networks.

This implies the beginning of an arms race between detectors and obfuscators. In Section 3.5 we showcase interesting results about the viability of hiding backdoors in networks that can be verified, a heuristic but effective defense appears to be viable.

To test obfuscation of the switch network shown on Figure 3.1, the ReLU cutoff and multi-stage ω weight is implemented, which produce the constant ω input for neuron C. The calculations producing the two ω values are broken up into separate multiplication chains, as shown on Figure 3.3. The ω_{1i} and ω_{2i} values come from the same random distribution, which can be engineered to mimic the distribution of networks weights. The result of the two chains might not be perfectly equal due to rounding errors. In these cases the switch will not function correctly, the chains

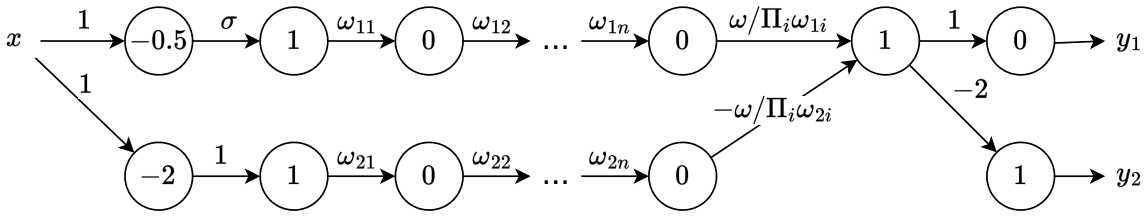

Figure 3.3: *Obfuscated version of the trivial adversarial network, where $\sigma < -2$ and both the ω_{1i} and ω_{2i} series multiply to exactly ω. Magnitude of the ω_{ji} values can be controlled by the number of multiplication layers.*

have to be adjusted. In our experience 70% of the trials yielded the symmetric chains needed for a working switch. The neuron with constant zero output is also hidden. Given the $x \in [0, 1]$ input variable, the first neuron in the lower branch on Figure 3.3 will always evaluate to zero. The input interval of its ReLU activation is $[-2, -1] < 0$, hence the output is mapped to $[0, 0]$.

The obfuscated network switch was tested with 20 and 50 multiplication stages, and different σ and ω parameters. The networks were evaluated with MIPVerify using the Gurobi, CPLEX, and GLPK solvers.

When ω is above 2^{54} MIPVerify is defeated with all configurations and networks. At an ω value whose rounding error as a 64-bit floating-point number just exceeds the "small value" constant in the $\omega + 1 - \omega$ computation the results are mixed, they are shown in Table 3.1. We only consider a result successful if MIPVerify using all three solvers comes to the wrong conclusion that the network is safe. Results show that the n and σ parameters have no effect on the verification outcomes.

$\sigma \in$	$[-15, -10]$	$[-10, -6]$	$[-6, -4]$	$[-4, -3]$	$[-3, -2]$
$n = 20$	97.4%	97.5%	98.6%	92.3%	96.0%
$n = 50$	97.4%	97.3%	95.9%	97.3%	98.7%

Table 3.1: *Percentage of successful attacks by the network shown on Figure 3.3, with n, σ, and $\omega = 2^{54}$ parameters. An attack is successful if MIPVerify considers the network safe using any of the Gurobi, CPLEX, and GLPK solvers.*

3.4.7 Defense

Although obfuscation techniques can make detection of the switch significantly harder, it cannot prevent anyone in the possession of the network to disrupt the adversarial behavior. As mentioned before, the functioning of the switch depends on the ω

constants on the two paths being equal, down to the last bit. As an idea for defense, the network weights and biases can be perturbed to disrupt any careful alignment that could hide adversarial behavior. This defense is cheap, and as a precaution can be performed without any prior knowledge whether a network is clean of backdoors. Of course the defense does not guarantee that an ordinary backdoor or defect does not exist in the network, verification is still an important step.

Let's see the case when multiplicative perturbation is applied to every constant in the network in the $1 \pm \delta$ range, where δ is calibrated to the floating-point resolution. In the computation of neuron C, the ω^+ positive ω coefficient and ω^- negative ω coefficient are part of a summation. When the $\omega^+ \cdot (1 + \delta^+)$ perturbed coefficient is smaller than $\omega^- \cdot (1+\delta^-)$, the input of neuron C permanently becomes a large negative number. As the rounding error of numbers on the same order of magnitude as ω is very large, the miniscule difference between the two branches easily outweighs the 1 constant. The result is that C has a constant 0 output. The D_1 neuron cannot become active, therefore the backdoor behavior does not affect the network output. The backdoor is disabled and the network becomes safe.

If the perturbation caused the $\omega^+ \cdot (1 + \delta^+)$ branch to be larger in value, the situation is not so clear. Although a bit modified, the backdoor functionality still remains, especially if the backdoor implements a switch output limit in the $[0, 1]$ interval, with the help of a C' neuron. The small relative difference between the ω values will be large compared to the bias of C, the backdoor however still functions. The slightly smaller $\omega^- \cdot (1 + \delta^-)$ value still allows deactivation of the backdoor behavior, with the active domain being slightly larger. The main advantage for verification is that the special canceling of the ω^+, ω^- pair is no longer working perfectly. It is much easier for MILP solvers to pick up on the backdoor behavior, and explore the "on state" of the switch when searching for feasible solutions.

The backdoor either becomes a larger target for MILP solvers, or it becomes completely deactivated – with about 50% probability. The simple perturbation provides a strong defense against backdoors that utilize rounding errors and the perfect cancellation of large numbers. However, a detector is only useful when it is used in practice. The existence of a simple and strong defense is not enough to dismiss the potential of backdoors based on rounding errors.

Tests were evaluated on the *WK17a* and *WK17a-adv* networks. A series of δ perturbations are tested, shown in Table 3.2 alongside the averaged *test accuracy* [4] of

[4]The ratio of correctly classified samples in the test set.

10 independent perturbed networks. As expected, larger noise results in lower test accuracy, a balance has to be found. For $\delta = 10^{-6}$ and smaller relative perturbations the networks preserve all classifications on the test set. This is in alignment with the number representation of network parameters, as the relative rounding error for 32-bit floating-point values is around $\epsilon_{32}(1.0) = 10^{-6}$.

Interestingly, the network with a switch has much higher sensitivity to the perturbations. The explanation is that noise in the logits layer bias leads to false classification. Notice that the absolute value of every neuron in the logits layer is increased by 1000, the stored information only amounts to 1% to 10% of the absolute value. This answers why test accuracy falls sharply from 10^{-2} and above. After the fact compensation of the constant is not possible, as the noise cannot be separated from the activation values. Other structures that can keep noise and data separate can be used to eliminate this sensitivity.

δ	10^{-1}	10^{-2}	10^{-3}	10^{-4}	10^{-5}	10^{-6}	10^{-9}	0
WK17a	0.9788	0.9811	0.9810	0.9811	0.9811	0.9811	0.9811	0.9811
WK17a-adv	0.1118	0.3744	0.9725	0.98105	0.98105	0.9811	0.9811	0.9811

Table 3.2: *Test accuracy of the WK17a and WK17a-adv networks with perturbed parameters (average of 10 independent perturbed networks).*

Verification using MIPVerify+Gurobi shows improvement on the perturbed adversarial networks. The switch network and the obfuscated switch network perturbed with 10^{-3}, 10^{-6}, and 10^{-9} values of δ are identified as adversarial in all cases.

The *WK17a-adv* network was evaluated using MIPVerify with different δ perturbation levels, as shown on Table 3.3. As expected, the backdoor is disabled in about 50% of the cases. To test the verification process, the networks with a still functioning backdoor are selected. Tests showed 75.85% to 98.3% success rate for finding the backdoor. Curiously, smaller scale perturbations seem to result in higher success rates of verification. We currently do not have an explanation for this behavior. One speculation is that larger perturbations create a larger $\omega^+ - \omega^-$ difference, between 10^8 and 10^{14} order of magnitude. These values will enter the model through constraints of the C neuron and cause issues during the MILP solving process.

Verification was also performed on the original *WK17a* network with the same perturbation levels. As Table 3.3 shows, the 10^{-6} and 10^{-9} perturbation level results matched the zero perturbation baseline of 4.38%, while the 10^{-3} perturbation level even made a small accidental improvement in adversarial sensitivity to 4.37%.

δ	10^{-3}	10^{-6}	10^{-9}	0
WK17a	4.37%	4.38%	4.38%	4.38%
WK17a-adv	75.85%	91.03%	98.3%	4.38%

Table 3.3: *Adversarial sensitivity of the WK17a and WK17a-adv networks with perturbed parameters, measured using MIPVerify+Gurobi.*

In conclusion, the results show that for a network using 64-bit floating-point representation $\delta = 10^{-9}$ perturbation is a good choice. In theory even smaller perturbations are enough, as long as the delicate balance of the ω weights is disturbed, and the MILP solver is able to operate on model variables with large scale differences.

3.5 Modeling floating-point computations

In this section the ongoing research topics are discussed, that naturally follow after showing problems with MILP based neural network formulations. It is important to acknowledge that these preliminary findings are not always refined enough to support definitive or robust statements. I present some ideas and findings to give a basic understanding on what is possible with the current technology using mixed integer linear programming solvers and interval arithmetic.

As it was shown in the previous section, neural network verification based on MILP solvers is an approach plagued by numeric errors. Errors in the MILP constraints and optimum values are carried over and amplified in the iterative model construction. Inaccuracies in the model can cascade to large errors, in badly aligned cases the errors can lead to wrong modeling of the network connectivity. In the worst case, model errors and consequently errors in the verification process can grow to an arbitrary amount. Numeric issues have to be accounted for to avoid misleading the verification process.

3.5.1 Is interval arithmetic always applicable?

It is a widely applied fact that interval arithmetic is a tool capable of producing valid bounds for floating-point arithmetic computations [21]. While this statement is true, some nuances exist that need attention. Using pure interval arithmetic guarantees the analytical result to fall within bounds computed for an expression.

Analytical results have infinite precision, they do not suffer from the nonlinear behavior caused by numeric errors. In contrast, the rounding behavior of floating-

point arithmetic introduces nonlinearity, floating-point addition and multiplication are both nonlinear. As a consequence, the $a +_{f64} (b +_{f64} c)$ expression might not evaluate to the same value as $(a +_{f64} b) +_{f64} c$, where $+_{f64}$ is 64-bit floating-point addition. The bounds produced by interval arithmetic are intended to encapsulate the analytical result of expressions [19]. Floating-point results may also be correctly bounded, but new theory is needed to have conclusive statements.

Definition 3.1. Let $\mathcal{E}$ denote the set of finite numeric expressions that consist of the $\circ \in \{+, *, /\}$ operations, the $(...)$ precedence operator, symbols, and signed numbers. Symbols – for example x_1, or δ – can be replaced with numbers or expressions in $\mathcal{E}$. For technical reasons the subtraction operator is not allowed in $E \in \mathcal{E}$, but it can be expressed, as $a - b \equiv a + (-1 * b)$.

Definition 3.2. Let $\mathcal{R}$ denote the set of rigid numeric expressions. In all $R \in \mathcal{R}$ expressions a precedence operator encapsulates each operation, R is built recursively from expressions of the form $(a \circ b)$, where a and b are replaced by other expressions in $\mathcal{R}$. $\mathcal{R} \subset \mathcal{E}$ is easy to see.

Corollary 3.1. Due to the construction of $R \in \mathcal{R}$, it does not contain an associative sub-expression, the order of operations is therefore defined.

Corollary 3.2. Given a $P \in \mathcal{E}$ expression where the precedence rules already define the order of operations, the P' equivalent expression can be constructed, where the order of operations is made explicit by repeated application of the precedence operator. For example the $P : a + b * c$ expression is transformed into $P' : (a + (b * c))$. As $P' \in \mathcal{E}$, and all operators are encapsulated with the precedence operator, P' has a form that implies $P' \in \mathcal{R}$.

Definition 3.3. Let $eval(E, A, I)$ denote the value of an $E \in \mathcal{E}$ expression, where first the $I = \{s \to v\}$ set of symbol replacements is applied, then the operations in E are evaluated as defined by the A arithmetic. The evaluation can only happen if following the $E \mapsto_I E'$ substitution E' contains no symbols, the result of operations acting on symbols is not defined. Evaluating the contents of $(...)$ always has the highest precedence in $eval(E, A, I)$, then $\{/\}$ has the second highest, then $\{*\}$, and lastly $\{+\}$ has the lowest precedence. On the $\{+, *\}$ associative operations there is no predefined evaluation order, the evaluation may take any order, hence a unique result is not guaranteed. Consecutive $\{/\}$ operations are always performed left to right. It is assumed that A has a symmetry on positive and negative numbers, if x and c are representable numbers, then $-x$ can also be represented, and $-(x + c) = -x - c$.

From Corollary 3.1 and Definition 3.3 it follows, that due to the explicit precedence in $R \in \mathcal{R}$, the $eval(R, A, I)$ evaluation has no ambiguity. The order of operations is always defined, and a single result exists.

Lemma 3.1. Given an $E \in \mathcal{E}$ expression, the $eval(E, \mathbb{R}, I)$ analytical result exists and is unique, where $\mathbb{R}$ denotes the arithmetic on real numbers.

Proof. In $\mathbb{R}$, the $\{+, *\}$ operations are associative, the order of same precedence operations does not matter. A sub-expression consisting of consecutive $\{+\}$ or $\{*\}$ operations always yields the same result. The order of consecutive $\{/\}$ operations is defined in $eval(E, \mathbb{R}, I)$, therefore the result of this sub-expression is also unique. Precedence between the $\{+, *, /\}$ operations is also defined. Since all valid evaluation orders yield the same result, the result of $eval(E, \mathbb{R}, I)$ is unique. □

Digitally, for example with $\mathbb{F}$ (floating-point arithmetic) E might be evaluated in multiple ways, yielding different results. As Moore has stated [19], the floating-point result of E is not fixed under associative and distributive transformations.

The $\{+, *, /\}$ operations are monotone in $\mathbb{F}$. Given a δ positive number, $a \circ b \leq (a + \delta) \circ b \leq (a + 2\delta) \circ b$. A similar inequality also holds for a negative δ, and δ applied to the b number. When b is amended, the inequality changes in direction for the $\{/\}$ operation. This is a simple consequence of the monotonicity in $\mathbb{R}$, the correct rounding in $\mathbb{F}$, and the representation of floating-point numbers.

Definition 3.4. $\mathbb{I}_\mathbb{F}$ denotes interval arithmetic (IA), where $\mathbb{F}$ is the floating-point arithmetic used by $\mathbb{I}$. Given an $E \in \mathcal{E}$ expression, the v number in E that is representable by $\mathbb{F}$ is implicitly handled by $\mathbb{I}_\mathbb{F}$ as the $[v, v]$ zero width interval. Crucially, in $\mathbb{I}_\mathbb{F}$ no refinement takes place after the bounds have been obtained with rounding.

From the IEEE 754 standard for floating-point computations [27] and the results on interval arithmetic by Moore [19], $\mathbb{I}_\mathbb{F}$ can find reliable bounds for the operation performed in $\mathbb{R}$. By choosing round to $-\infty$ mode (round down mode), a floating-point operation results in the largest representable number, that is not greater than the result from the same operation performed in $\mathbb{R}$. This is also true for round to $+\infty$ (round up mode), where a reliable upper bound can be obtained for the result.

Lemma 3.2. Given an $X_1 \circ X_2, \circ \in \{+, *, /\}$ expression, where X_1, X_2 are intervals in $\mathbb{I}_\mathbb{F}$, and given the $x_1 \in X_1$, $x_2 \in X_2$ numbers representable by $\mathbb{F}$, it follows that $eval(x_1 \circ x_2, \mathbb{F}, \emptyset) \in eval(X_1 \circ X_2, \mathbb{I}_\mathbb{F}, \emptyset)$.

Proof. Interval arithmetic operations are constructed to bound the analytical result, $eval(x_1 \circ x_2, \mathbb{R}, \emptyset) \in eval(X_1 \circ X_2, \mathbb{I}_\mathbb{F}, \emptyset)$ holds. Given the $y_\mathbb{R} = eval(b_1 \circ b_2, \mathbb{R}, \emptyset)$, and $Y_\mathbb{R} = eval(X_1 \circ X_2, \mathbb{I}_\mathbb{R}, \emptyset)$ evaluations, where b_i is an extreme value of the corresponding X_i interval, and given the $\circ$ monotone operation, $y_\mathbb{R} \in \{\max(Y_\mathbb{R}), \min(Y_\mathbb{R})\}$ holds. If $Y_\mathbb{R} \subseteq Y_\mathbb{F}$ holds[5], the $Y_\mathbb{F} = eval(X_1 \circ X_2, \mathbb{I}_\mathbb{F}, \emptyset)$ bounding is correct. Rounding errors must be accounted for when evaluations in $\mathbb{F}$ are performed, when calculating the bounds of $Y_\mathbb{F}$, the round down and round up mode is used. As $\circ$ is monotone, the bounds of $Y_\mathbb{F} = eval(X_1 \circ X_2, \mathbb{I}_\mathbb{F}, \emptyset)$ can be obtained in a way, where extreme points of the input intervals are enough to consider. The bounds of $Y_\mathbb{F}$ are directly computed using evaluations like $eval(b_1 \circ b_2, \mathbb{F}, I)$, where I substitutes b_i for the necessarily evaluated bounds. Therefore, the result of $eval(x_1 \circ x_2, \mathbb{F}, \emptyset)$ must be contained in $Y_\mathbb{F}$, where instead of the b_i bounds the $x_i \in X_i$ contained points are substituted by I. □

Theorem 3.1. Given an $R \in \mathcal{R}$ expression, $eval(R, \mathbb{F}, I) \in eval(R, \mathbb{I}_\mathbb{F}, I)$.

Proof. Let $R^{(i)}$ denote a series of expressions, where $R = R^{(0)}$, and let $I_\mathbb{F}^{(i)}$ and $I_\mathbb{I}^{(i)}$ denote the series of substitution sets for the evaluations by the $\mathbb{F}$ and $\mathbb{I}_\mathbb{F}$ arithmetic, where $I = I_\mathbb{F}^{(0)} = I_\mathbb{I}^{(0)}$. Let a, b be values or symbols (not expressions), and let $R^{(i)}$ be the expression, where in $R^{(i-1)}$ the first occurrence of an $(a \circ b)$ shaped sub-expression is replaced by the $s^{(i)}$ symbol. Let $I_\mathbb{F}^{(i)} = I_\mathbb{F}^{(i-1)} \cup \{s^{(i)} \to eval(a \circ b, \mathbb{F}, I_\mathbb{F}^{(i-1)})\}$ and $I_\mathbb{I}^{(i)} = I_\mathbb{I}^{(i-1)} \cup \{s^{(i)} \to eval(a \circ b, \mathbb{I}_\mathbb{F}, I_\mathbb{I}^{(i-1)})\}$ be the symbol replacements for evaluations on the $R^{(i)}$ expression.

As an induction step let's assume that the $I_\mathbb{F}^{(i-1)}$ substitution set maps to the x_i numbers, and $I_\mathbb{I}^{(i-1)}$ maps to the same x_i numbers, or alternatively it maps to the corresponding X_i interval, such that $x_i \in X_i$. By Definition 3.4, $\mathbb{I}_\mathbb{F}$ handles values in $\mathbb{F}$, such that $v = eval(v, \mathbb{F}, \emptyset) \in eval(v, \mathbb{I}_\mathbb{F}, \emptyset) = [v, v]$. Lemma 3.2 applies, and establishes that $eval(a \circ b, \mathbb{F}, I_\mathbb{F}^{(i-1)}) \in eval(a \circ b, \mathbb{I}_\mathbb{F}, I_\mathbb{I}^{(i-1)})$. Consequently, the $I_\mathbb{F}^{(i)}$ and $I_\mathbb{I}^{(i)}$ substitution sets retain the property assumed by the induction step.

Let's provide an induction base case on the $R^{(0)}$ expression. Due to Definition 3.3, I must replace all symbols in $R^{(0)}$. For the $eval(R, \mathbb{F}, I)$ evaluation to be valid, I can only map to numbers that are representable by $\mathbb{F}$. By Definition 3.4, $\mathbb{I}_\mathbb{F}$ can handle values in $\mathbb{F}$, therefore $eval(R, \mathbb{I}_\mathbb{F}, I)$ has to be valid. As it was pointed out in the induction step, $\mathbb{I}_\mathbb{F}$ handles numbers correctly for the induction to work. Therefore, the base case also satisfies the assumption of the induction step.

By induction, $eval(R, \mathbb{F}, I) \in eval(R, \mathbb{I}_\mathbb{F}, I)$ holds. □

[5]The $F \subseteq G$ operator applied on intervals signifies that the bounds of F are between the bounds of G. Mathematically this implies that the G interval contains the F interval.

Definition 3.5. Let $d(E, A, I)$ denote a valid bounding function, the result of which contains all possible results of $eval(E, A, I)$. More concretely, given the $\mathcal{L}(E) \subset \mathcal{R}$ set of all possible rigid expressions derived from E, for each $L \in \mathcal{L}(E)$ expression $eval(L, A, I) \in d(E, A, I)$ holds. Let $\mathcal{D}(E, A, I)$ denote the set intervals that correctly bound all $eval(L, A, I)$ evaluations.

Lemma 3.3. If $\mathcal{L}(G) = \{G'\}$, where $G' \in \mathcal{R}$, then a $d(G, \mathbb{F}, I)$ interval exists, such that $d(G, \mathbb{F}, I) \subseteq eval(G, \mathbb{I}_{\mathbb{F}}, I)$. Interval arithmetic gives a correct bound for $eval(G, \mathbb{F}, I)$.

Proof. By definition, G has a unique order of operations. Applying Corollary 3.2, the G' equivalent expression exists, such that $G' \in \mathcal{R}$, and the $v = eval(G, \mathbb{F}, I) = eval(G', \mathbb{F}, I)$ value is unique. The $[v, v]$ zero width interval correctly bounds the v value, therefore $[v, v] \in \mathcal{D}(G, \mathbb{F}, I)$. From Theorem 3.1, we know that $eval(G, \mathbb{F}, I) \in eval(G, \mathbb{I}_{\mathbb{F}}, I)$. Since $[v, v] \subseteq eval(G, \mathbb{I}_{\mathbb{F}}, I)$, and $d(G, \mathbb{F}, I) = [v, v]$ is valid, a $d(G, \mathbb{F}, I)$ interval exists, such that $d(G, \mathbb{F}, I) \subseteq eval(G, \mathbb{I}_{\mathbb{F}}, I)$. □

The result of an expression calculated with infinite precision is not sensitive to the varying evaluation order of associative expressions. Calculating the interval arithmetic bounds of an expression in $\mathcal{E}$ will encapsulate the true value. However, the floating-point value also depends on the evaluation order. As it is demonstrated below, computing the interval arithmetic bounds for an expression does not guarantee to encapsulate the floating-point value of associative variants. When interval arithmetic bounding is used in verification algorithms, the provided bounds are implicitly assumed to encapsulate the floating-point value of the evaluated expression. This assumption is only valid in cases where the expression has no associative ambiguity, $|\mathcal{L}(E)| = 1$ therefore $E' \in \mathcal{R}$ exists. Verification tasks do not satisfy this assumption. Neural network definitions usually have a lot of associative elements, at minimum the performed vector multiplications involve associative addition.

Let's explore an example of the problematic behavior. Let $N = [n_1, n_2, \cdots, n_i]$ be a list of floating-point numbers that are at an ordinary scale (for example in the same order of magnitude as 1), and an ω large number that has a comparably significant rounding error. Including ω and $-\omega$ in the N' list does not change the overall $S' = \sum n_i$ sum, assuming infinite precision. However, an arithmetic with rounding errors produces very different results based on the summation order. In the $S' = \omega - \omega + \sum n_i$ form, where the large numbers can cancel out first, the result is just the floating-point $\sum n_i$ sum. If the summation is in the $\sum n_i + \omega - \omega$ form where the large numbers cancel out last, the result will be the multiple of $eps(\omega)$ closest to S, the resolution

becomes degraded. The most dramatic rounding errors appear if the summation is in the $\omega + \sum n_i - \omega$ form where the canceling of the large numbers surrounds the sum. Every $\omega + \sum_1^j n_i$ sub-result will be rounded to a multiple of $eps(\omega)$. In drastic cases the rounding always happens in the same direction. Due to the rules of the "round to nearest" rounding applied in every summation step, the worst case scenario can have an error equal to the S sum.

Given that all n_i numbers have the same sign, and their absolute value is lower than $\epsilon(\omega)$, the rounding errors can add up to the value of S. Given any evaluation order $\sum_{N'} n_i \in S \pm S = [0, 2S]$, where both the 0 and $2S$ extremes are reachable with the correct summation order.

The expressions and their results in Table 3.4 demonstrate this behavior, where $eps(\omega) = 2$, and $1 \pm \epsilon$ are the closest values[6] to 1.0 in f_{64}. Results show that the sums have a similar effect on interval arithmetic evaluation. This is expected, as $\mathbb{I}_{\mathbb{F}}$ is based on the $\mathbb{F}$ arithmetic, and rounding errors are always taken into account by interval arithmetic sums. However, in case of the $\omega - \omega$ difference there is no rounding error, resulting in the $[0, 0]$ interval[7]. In accordance with floating-point computations, when the $\omega - \omega$ term is located at the end, the bounds will be a multiple of $eps(\omega)$ close to the original bound. With ω at the start and $-\omega$ at the end, the resulting interval grows fast, with every additional n_i term it is widened by $eps(\omega)$.

Expression	Result with f_{64}	Result with IA
$(1-\epsilon)+\omega-\omega$	0	$[0, 2]$
$\omega-\omega+(1-\epsilon)$	$1-\epsilon$	$[1-\epsilon, 1]$
$\omega-\omega+(1+\epsilon)$	$1+\epsilon$	$[1, 1+\epsilon]$
$(1+\epsilon)+\omega-\omega$	2	$[0, 2]$
$\omega+\sum_{1..M}(1-\epsilon)-\omega$	0	$[0, 2 \cdot M]$

Table 3.4: *Differing results achieved by different ordering of an expression using floating point arithmetic and interval arithmetic.*

With a large enough rounding error of the ω number, the floating-point sum can fall in the $[0, 2S]$ interval depending on the evaluation order. Meanwhile, when ω values are at the beginning, interval arithmetic produces the same result as for the $\sum n_i$ sum, the resulting interval will be quite narrow compared to $eps(\omega)$.

[6]The ϵ is added to avoid ambiguity in the rounding direction.

[7]Given that ω is exactly representable in the used floating-point arithmetic, the interval representation of ω is zero width. Since the result of the subtraction is also representable, there is no widening applied, the resulting interval has zero width.

Table 3.4 demonstrates that in a computation where the order of associative operations is not fixed, interval arithmetic does not necessarily bound the floating-point value of the expression, clearly the $f_{64} = 0$ result is not in the $\mathbb{I}_{f64} = [1, 1 + \epsilon]$ interval. This is problematic, as the order of operations in MILP solvers and neural network implementations can vary.

Floating-point evaluations are not always contained in interval arithmetic bounds.

A notable safe case is provided by Theorem 3.1. Given an expression with fixed evaluation order, an $\mathbb{I}_{\mathbb{F}}$ arithmetic satisfying Definition 3.4 is guaranteed to contain the expression value computed by the $\mathbb{F}$ arithmetic.

A surprising consequence is that interval arithmetic cannot be trusted to evaluate reachability of a single neural network layer, due to the possible associative expression variants. Evaluation of a network layer with $\varphi(A \cdot x + b)$ model is ambiguous. Unless we can guarantee that the result of all evaluation orders are bounded – either by restricting the evaluation order, or by correct bounding –, the resulting floating-point vector might not be contained in the output interval bounding box.

The seemingly trivial solution to the problem is to strictly define the order of operations in the neural network model. This however leads to performance loss, high-throughput computations benefit from a flexible evaluation order. For example matrix multiplication is highly optimized for the trio of a given task, the running environment and the momentarily available resources. Results may vary with GPU model, tuning of the GPU model, current load on the system, or the input batch size. While it would be hard to create a reliable exploit for these volatile factors, a generally unstable neural network could be created that behaves erratically on specific inputs.

Generally, this problem can rarely manifest on neural networks that have not been tampered with. Overestimation in the interval arithmetic bounding typically covers the small errors that could result from the evaluation order. However, targeted attacks could exploit the inconsistency.

Despite the issues, interval arithmetic is still a strong tool with good applicability in verification. Naturally, the limitations must be accounted for, or acknowledged.

3.5.2 Combined power of interval arithmetic and MILP solvers

As we have seen, verification based on MILP solvers can evaluate network models very efficiently, however a known exploit exists, therefore the results cannot be fully

trusted. It is also known that verification based on interval arithmetic can be sound, but solvable verification problems are very limited in complexity. MILP solvers are equipped with algorithms to track and simplify logical connections in a complex problem, meanwhile interval arithmetic is good at accounting for numeric issues during a computation.

Existing algorithms – including MIPVerify – are utilizing the combination of interval arithmetic and MILP solvers, however these implementations ignore the potential rounding errors in the modeling and model solving process. In the iterative approach of MIPVerify we clearly see problems that occur due to the ill-formed model. An adversarial network can choose the value of some parameters in the constructed MILP model. Finding the optimum in such an ill-formed model can be difficult, purposefully introduced numeric issues can trick the solver into finding a sub-optimal solution. We demonstrated this behavior with the forced incorrect modeling of our trivial adversarial network (Figure 3.1). Besides the ill-formed model, model scaling is another attack surface, where an adversary can purposefully upscale the MILP model parameters by a constant. Even though proportions of the model geometry are unchanged, not every solver parameter is scaled accordingly. An adversary can forcefully reduce the solution accuracy by a geometrically equivalent model. Slightly tighter than necessary constraints will result in more pessimistic optimum values, and consequently a model that omits reachable network output regions. An attacker can amplify the effect and hide adversarial behavior in these regions.

The *Gurobi Guidelines for Numerical Issues* excessively writes about problems caused by ill-conditioned models. An adversary could design networks that exploit these issues, therefore verifiers must have defenses that solve these issues. One such vulnerability that surfaced frequently during our research is that numeric instabilities have led to unsatisfiable constraints in a model that modeled the adversarial space. We had example inputs that simply proved this result wrong.

In cases where the optimum value is conservative, for example due to already modeled constraints being too strict, the built MILP model will impose constraints that are too narrow to correctly encapsulate all possible activations. Consequently, in iterative construction of the network model use of the too narrow intermediate models leads to unsatisfactory bounding of layer outputs. These bounds are used to determine ReLU stability, some might be erroneously classified as stable. Completely modeling stable ReLU units would introduce unnecessary binary variables reduce solver performance, therefore only a simplified version is included in the model. This problem allowed the trivial adversarial network to became so powerful. After the

verifier has decided to model the most important neuron in the network as the *zero* constant, any adversarial behavior was necessarily and completely missed.

If the effect of these modeling errors could be estimated, then solutions from the MILP solver could be amended with the known uncertainties. For now, we have to accept the fundamentally unreliable verifiers, with added protection in detecting numeric issues and trusting solver correctness in average cases. When any issue is detected that could lead to sub-optimal MILP solutions, the verifier must not give out safety certificates. The optimized MILP model could be ill-formed, and the result is unreliable. Solvers usually have built-in mechanisms to detect such issues, during model building the constraint parameters can also be sanity checked. For example the trivial adversarial network has parameters in the 10^{17} order of magnitude which is a clear red flag. By acknowledging currently unfixable vulnerabilities and sacrificing completeness, a heuristic, performant, and closer to sound algorithm is achievable. Not accepting the result of MILP solvers at face value is an important step, that must be taken to avoid the false sense of security.

The best MILP solvers are complicated closed source algorithms. It would be hard to bound rounding errors that affect a concrete model solution even when the source code is available. Without knowledge on the internals, we cannot provide reliable bounds. On the other hand, some assumptions can be made that point in the right direction, so that creation of a less vulnerable heuristic system can be attempted.

If the MILP model and optimum values are extended with heuristic error bounds, some hard to overcome protections can be built in the verifier algorithm. Granted that numerically unstable verification tasks are rejected, we only have to focus on bounding the average case rounding errors. In the average case, MILP solver operation is stable, optimum values can be accepted as correct and without much error. Error bounds for the average case computation can be relatively tight.

In verification the conservative estimation[8] of reachable states is vital in every computation step. If necessary overestimations are kept small, the completeness of a method will only weaken a little, while correctness can be ensured. Small overestimations can be indistinguishable from normal execution with rounding errors. When large overestimations have to be used, the underlying numeric problems already make usability of MILP solvers questionable.

[8]Conservative in this case means, that the estimation is wider, such that all possible states are encapsulated. In verification false positives are tolerated, false negatives are not.

3.5.3 Rounding errors in the MILP model optimum

When the MILP solver provides the optimum solution for a problem, the corresponding value of all model variables can be retrieved. Based on the input vector, the optimum value can be re-calculated with improved reliability, and an error estimate can also be obtained. Based on the assumptions on how rounding errors are introduced, specific bounding algorithms are required for the error estimate. The simplest of all is to use the unaltered standard tool, interval arithmetic, with the limitations established in Section 3.5.1 kept in mind. Given an algorithm that can bound associatively ambiguous expressions, it can be a drop-in replacement for interval arithmetic to get safer bounds.

While an *any-order bounding algorithm* is not developed, we still want to use our "verifiers" aided by an educated guess on error bounds. Simply applying interval arithmetic does not take into account that an ω number with large rounding error has a strong effect on our results. To mitigate this issue and lessen the attack surface, a simple workaround can be applied. When evaluation of an expression is started, constants in the expression can be widened by adding a minimal non-zero width interval. For example the $[-\epsilon(0), \epsilon(0)]$ or $[0, \epsilon(0)]$ interval can be used, where $\epsilon(0)$ is the smallest representable number. Even when terms of the expression would perfectly cancel out, at least one of the operands is a non-zero width interval, which ensures the expression result to also have non-zero width. If the expression contains ω, the result is guaranteed to include the effects of degraded precision caused by $\epsilon(\omega)$. Therefore, the $\omega +_{\mathbb{I}} -\omega +_{\mathbb{I}} 1$ expression will result in the $[1-2\cdot\epsilon(\omega), 1+2\cdot\epsilon(\omega)]$ interval. If ω is on an average scale, the introduced error term is around 10^{-4} for 32-bit floating point representation, and only 10^{-13} for 64-bits – a tolerable overestimation. While this does not eliminate all problems, the attack surface is significantly reduced by the "epsilon widening" of expressions.

3.5.4 Include rounding errors of modeling constraints

Sadly, even if a good any-order bounding algorithm is available, rigorous bounding of the optimum value is not guaranteed. On pathological MILP models generated by a malicious network, MILP solvers vulnerable to numeric issues do not guarantee optimality of the solution. When solvers determine possible solutions and evaluate their feasibility, the model constraints directly affect the result. Since the constraints are driven by the network structure, an adversary can introduce unstable regions.

Small discrepancies can be introduced in modeling of the linear expressions and the ReLU units, where the constraints are stricter than necessary by just a few ϵ values. The attack does not require large rounding errors that the optimum bounding would account for, the exploit is encoded in the MILP model structure. Therefore, the real optimum can still get out of bounds. If this reliably happens by even a small amount, the exploit is feasible, and verification results cannot be trusted.

When constructing the MILP model, too strict constraints can amplify numeric errors. A constraint can behave strict due to a rounding error changing it slightly, two constraints in a scissor geometry are particularly sensitive to such small numeric issues. In case of 32-bit models this effect can be quite noticeable, as the relative error in the number representation itself is quite large. Gurobi uses 64-bit representation and by default applies 10^{-6} slack for the constraints and optimality condition to reduce numeric issues. It also applies 10^{-5} slack for integer constraints, which means that integer values can be off by this amount. The extra tolerances help with satisfying the strict equality and integer constraints. However, according to the "Gurobi Guidelines for Numerical Issues" in [8] and our empirical results, these tolerances do not provide a complete solution to strictness issues.

As mentioned in Section 3.5.2, there are different ways of introducing rounding errors in the MILP solving process. Geometry scaling is an interesting one. While scaled up models should have equivalent satisfiability, implementation details of the solver can be used to induce errors. Most importantly, the tolerances mentioned above are absolute, scaling of the optimization problem matters. Therefore, geometry scaling can control the effective model strictness. By upscaling, or on a model with values on a very wide scale spectrum (like the trivial adversarial network), adversaries can force an overly narrow search space.

The geometry of the network activation pattern has a free scaling parameter. Values in hidden layers can be scaled up and down, while the same network outputs are maintained as without scaling. With the naive modeling currently used by MIPVerify, an adversary can choose the scaling parameter, and consequently can exploit the well-known numeric issues. It should be noted that these numeric issues are not bugs in the MILP solvers, they come from the intrinsic behavior of floating point computations. The currently widespread 64-bit representation gives a hard limit on solver accuracy.

Calculating whether a solution satisfies a constraint suffers from rounding errors, this phenomenon enables numerous attacks. Since the error occurs in the MILP solving process, it has to be accounted for either before, or during the solving process.

An after the fact relaxation is not suitable, as an overly strict constraint could render a real optimum infeasible, which in turn can lead to errors of arbitrary magnitude.

Constraints can be widened by an error bound to mitigate issues in the MILP solving process. A heuristic error profile and magnitude can be assumed that make attacks harder to execute. One idea is to approximate accumulated rounding errors with the $\epsilon(x)$ function and a magnitude value. However, the $\epsilon(x)$ function only bounds a floating point value, and not the result of an expression. When a constraint calculation has rounding errors, a resulting 0 value does not mean that the calculation has $\epsilon(0)$ error. The error profile and magnitude should both be deduced from the constraint expression, and from the bounds of input variables. The constraint can be widened with this amount.

The true error profile of linear expressions is complicated. Inputs can take any value from the determined input intervals, which yields an error function with $\mathbb{F}^N \to 1$ dimensionality, where N is the number of inputs. It is not feasible to exactly determine and model the error function, however a simple bounding is possible given that a $d(E, \mathbb{F}, I)$ any-order bounding function exists. From $[l, u] = d(E, \mathbb{F}, I)$, we know that $eval(E, \mathbb{F}, I) \in [l, u]$, and $eval(E, \mathbb{R}, I) \in eval(E, \mathbb{I}_\mathbb{F}, I)$, also it is easy to see that $eval(E, \mathbb{I}_\mathbb{F}, I) \cap d(E, \mathbb{F}, I) \neq \emptyset$. Therefore, $|eval(E, \mathbb{R}, I) - eval(E, \mathbb{F}, I)| \leq |[l, u]| + |eval(E, \mathbb{I}_\mathbb{F}, I)|$, the rounding error on an E expression can be bounded.

Definition 3.6. Let $\mathcal{I} = \{s \to [l_s, u_s]\}$ denote the set of symbol to interval mappings. Given the $\mathcal{I}$ set, $I_\mathcal{I}$ is a symbol substitution set, that has a substitution for each $s \in$ key($\mathcal{I}$). Given a $(s \to [l, u]) \in \mathcal{I}$ mapping, the $(s \to v) \in I_\mathcal{I}$ substitution satisfies $v \in [l, u]$.

Definition 3.7. Let $q(E, \mathbb{F}, \mathcal{I})$ denote a valid bounding function for the maximum rounding error that can occur when $eval(E, \mathbb{F}, I_\mathcal{I})$ is calculated. Formally, given the $\mathcal{L}(E)$ set of all possible expressions derived from E, such that $\mathcal{L}(E) \subset \mathcal{R}$, let $q(E, \mathbb{F}, \mathcal{I}) \geq \underset{L_1, L_2 \in \mathcal{L}(E)}{argmax}\ eval(L_1, \mathbb{F}, I) - eval(L_2, \mathbb{F}, I)$.

Currently, a proven algorithm for calculating a valid $q(E, \mathbb{F}, \mathcal{I})$ bound is not known. With a similar construct to interval arithmetic, where both the interval value and the rounding error is tracked, an algorithm might be feasible.

Given the network up to the $i-1$th layer and the possible intervals for all $x_j^{(i-1)}$ variables, an error bound can be calculated for the $z_j^{(i)} = b_j^{(i)} + \sum_k w_{j,k}^{(i)} \cdot x_k^{(i-1)}$ expression. An assumption has to be made on how the MILP solver determines whether a constraint is satisfied. For an $A \cdot x = b$ model, the $A \cdot x - b = 0$ constraint test is

assumed. A constraint takes the $-b_j + \sum_k A_{j,k} \cdot x_k = 0$ form, the solver is assumed to literally compute the left-hand side of this expression and check whether the equality holds within the slack parameters.

Given that the MILP solver does not have a larger error in constraint calculation than the rounding errors in this expression, the determined error bounds can be applied as the $\delta = q(E, \mathbb{F}, I)$ slack to the constraint. In practice, if a $\delta_j^{(i-1)}$ error bound is calculated, the equality constraint has to be modeled as the $z_j^{(i)} \geq -\delta_j^{(i-1)} + b_j^{(i)} + \sum_k w_{j,k}^{(i)} \cdot x_k^{(i-1)}$ and $z_j^{(i)} \leq \delta_j^{(i-1)} + b_j^{(i)} + \sum_k w_{j,k}^{(i)} \cdot x_k^{(i-1)}$ constraint pair. On constraints modeled with $\leq$ and $\geq$, the $\delta_j^{(i-1)}$ value can be directly applied with the correct sign, no re-modeling is required.

With the corrected constraints for $z_j^{(i)}$, we can proceed to model the ReLU activation. As $relu(z) = max(0, z)$, there are no rounding errors introduced in direct calculation of the function. However, we need to take into account that modeling of the ReLU function requires the 3.1-3.4 constraints, rounding errors might still be introduced.

For ease of notation, let's make $z = z_j^{(i)}$ and $x = x_j^{(i)}$, therefore $x = relu(z)$. The simplest of the four constraints is $0 \leq x$. As x is just a value, no calculation is needed to determine it, $q(x, \mathbb{F}, \mathcal{I}) = 0$. The constraint is not modified.

The $z \leq x$ constraint will be rearranged to $0 \leq x - z$ (or equivalently $0 \geq z - x$). In this case $\delta = q(x - z, \mathbb{F}, \mathcal{I})$, which depends on the intervals z and x can fall into. The constraint becomes $0 \leq x - z + \delta$.

The $x \leq u \cdot a$ constraint will be rearranged to $0 \leq u \cdot a - x$, where u is a constant, therefore the expression is still linear. In this case $\delta = q(u \cdot a - x, \mathbb{F}, \mathcal{I})$. In multiplication with the 0 and 1 theoretical values of a there is no additional rounding error. In reality, the MILP solver introduces 10^{-5} slack for integer variables, therefore $a \to [0 - 10^{-5}, 1 + 10^{-5}]$ is the correct mapping in $\mathcal{I}$. The constraint becomes $0 \leq u \cdot a - x + \delta$.

The $x \leq z - l \cdot (1 - a)$ constraint has one key difference to the previous ones, the expression is not in the "sum of products" form. The $A \cdot x \leq b$ encoded expression is $x \leq z - l + l \cdot a$, which is rearranged to $0 \leq z - l + l \cdot a - x$, therefore $\delta = q(z - l + l \cdot a - x, \mathbb{F}, \mathcal{I})$. The constraint becomes $0 \leq z - l + l \cdot a - x + \delta$.

With this δ-widened model, the possibility of exploiting rounding errors in the MILP solver is reduced, the problem however is most likely not eliminated.

3.5.5 Improved ReLU stability check

Besides the question of reliability, verification speed is also an important quality. While we examined MIPVerify, an easy to achieve performance improvement was noticed. In MIPVerify the bounds for the $z_j^{(i)}$ variables do not have to be strict. Within reasonable scales the values of $l_j^{(i)}$ and $u_j^{(i)}$ do not matter, only correctness of the bounds is necessary. With interval arithmetic we can easily obtain most likely overestimated bounds[9], based on the bounds of $z_j^{(i-1)}$. If the ReLU is found to be stable by interval arithmetic, no further refinement is needed, modeling of the stable ReLU unit is trivial. When the overestimated bounds indicate an unstable ReLU unit, a choice can be made. Either the addition of a potentially unnecessary binary variable is tolerated, or by evaluating the so far built MILP model stricter bounds can be obtained, and only necessary binary variables are added. MIPVerify chooses the latter option in every case, therefore evaluates the MILP model numerous times.

MIPVerify first evaluates the $u_j^{(i)}$ upper bound. If the upper bound is negative, the ReLU unit is proven to be inactive. If the upper bound is positive, the $l_j^{(i)}$ lower bound also has to be sharpened. If the lower bound is positive, then the ReLU is known to be always active, otherwise it is potentially unstable and introduction of the $a_j^{(i)}$ binary variable is necessary.

This behavior can significantly be improved. The neural network can be evaluated at an arbitrary point and the value of all $z_j^{(i)}$ variables for that single evaluation can be saved. This provides a $z_j'^{(i)}$ example point for all ReLU units, and the points are known to lie between the activation bounds of the corresponding ReLU unit. When building the network, we can have a look at the $z_j'^{(i)}$ point for every ReLU that is deemed to be unstable using interval arithmetic bounds. If $z_j'^{(i)}$ is negative, we can prioritize solving the MILP model for the upper bound. If the upper bound is negative, the ReLU is proven to be inactive, hence introducing a binary variable is not needed. If the upper bound is positive, the ReLU is proven to be unstable with a sharp upper bound and an overestimated lower bound, based on which modeling of the unstable ReLU is possible. This logic can be symmetrically applied to the case where the $z_j'^{(i)}$ point is positive. In the unstable cases only a single MILP optimization round is needed, in some stable cases MILP optimization can be completely skipped.

In cases where the ReLU is permanently active, the original MIPVerify implementation requires two MILP evaluations. First the upper bound, which yields a positive value, then the lower bound which also yields a positive value. Our solution alre-

[9]While repeating the mantra “see the limitations mentioned in Section 3.5.1”.

ady knows that the ReLU can have a positive value, therefore it eliminates the MILP evaluation that sharpens the upper bound. When the first MILP evaluation yields a positive lower bound, the ReLU unit is proven to be stable.

If it turns out that the sharp bounds of the $z_j^{(i)}$ variables are important, a reduction in MILP solves is still possible. For stable ReLUs we can still ensure that only a single MILP solve is needed, as opposed to the either one or two MILP solves required by MIPVerify. For unstable ReLUs instead of having one sharp and one loose bound, both are sharpened, two MILP solves are required by the updated algorithm. Depending on the ratio of unstable ReLU units, the speedup from the network pre-evaluation can have a very high, or almost no impact on verification performance.

By evaluating the network once, and storing the $z_j'^{(i)}$ values in all layers, a significant reduction is achievable in the number of MILP evaluations. Also, the approach points to the possibility of a compromise between the number of MILP evaluations and the number of superfluous binary variables. In later stages of the model building MILP evaluations become more expensive, while the introduced binary variables take part in fewer evaluations. It is an interesting task to find the sweet spot for the best overall performance.

3.5.6 Runtime and reliability comparison

To gain data on effectiveness of the heuristic findings, different implementations derived from MIPVerify were created. We found that extending the MIPVerify algorithm using the published codebase by Tjeng et al. is not ideal for prototyping. Moreover, the answers returned by MIPVerify are not nuanced enough to clearly indicate its implications. For example a “not feasible” answer could indicate a strong guarantee on safety, or a heuristic result based on the raw output of the MILP solver. In case of MIPVerify it is in fact just a raw output from the MILP solver, which clearly needs further processing to indicate a clear conclusion.

To combat these problems a new toolbox was created, that is easy to extend with new algorithms and provides a cleaner interface for the user. Namely, we distinguish the answers `PROVEN_ADVERSARIAL`, `POSSIBLY_ADVERSARIAL`, `POSSIBLY_SAFE`, `PROVEN_SAFE`, and `UNKNOWN`. The answers containing the word `PROVEN` should only be used when there is evidence for the result excluding the possibility of the other answer being correct. For example the MILP solver reporting “not feasible” can at most be categorized as `POSSIBLY_SAFE`, because we know that MILP solvers are fallable. If the solver returns “feasible” the categorization should be `POSSIBLY_ADVERSARIAL`,

as the MILP solver might have made a mistake. Taking the model variables corresponding to the solution input and evaluating the input with the real neural network implementation, we might get an output that is categorized as acceptable for the safety requirements. If the tested input returns an adversarial example, then we have proof by example and can rightfully return `PROVEN_ADVERSARIAL`.

The `PROVEN_SAFE` answer needs the most care. An algorithm has to be proven correct before this answer is considered. If algorithms use it without consideration, trust in their correctness can be damaged. By default, the `POSSIBLY_SAFE` answer should be used, it highlights that the answer is not guaranteed to be error-free. Our algorithms never use `PROVEN_SAFE` as we are not comfortable issuing a strong safety certificate when the algorithms are suspected to be vulnerable.

To evaluate the proposed improvements to MIPVerify, several algorithm versions were created. The original MIPVerify algorithm implementation by Tjeng et al. is referred to as `mip`. To gain a baseline comparison, MIPVerify was reimplemented with a reduced feature set that is essential for the conducted tests, it serves as a comparison baseline. We will refer to the re-implemented algorithm as `miprep`. MIPVerify (specifically `miprep`) was extended with the improved bounds check mentioned in Section 3.5.5, this version is referred to as `mipplus`.

The `miprep` algorithm was also extended with a naive version of rounding error compensation and is referred to as `recrep`. The currently implemented rounding error compensation applies a fixed relative widening of constraints, as a first test for viability of improved modeling. It also extends the $l_j^{(i)}$ and $u_j^{(i)}$ bounds using a heuristic algorithm to estimate the rounding errors committed when calculating the network layers. A version called `recmip` was also created, that besides the rounding error compensation also implements the speed improvements described in Section 3.5.5.

Figure 3.4 shows runtime results of the MIPVerify variants. The *WK17a* network (without the adversarial backdoor) and the first 100 samples of the MNIST test set were used to evaluate the verifiers. Each run of 100 evaluations was ordered in ascending order of the runtime and is shown on the plot. As it is seen on Figure 3.4, `mip` is about twice as fast as the `miprep` re-implemented algorithm. Mostly the same packages and tools are used in the implementation of `miprep` and `mip`, the most important of which are `IntervalArithmetic`, `JuMP`, `MathOptInterface`. Theoretically it should not matter, but in practice it is important that the `Gurobi` optimizer was used with the `Gurobi.jl` package for interfacing. An important difference is that `miprep` uses `Flux` and `NNlib` to evaluate neural networks, while `mip` has a custom

implementation as part of the package. The observed slowdown and the difference in the calculation of model variables establishes that `mip` is better optimized than `recmip` in how or which model variables are created.

There is a noticeable grouping in the results, `mip` is the fastest, `miprep` and `recrep` are the slowest, `mipplus` and `recmip` stand in the middle. The difference between `miprep` and `recrep` is the added rounding error compensation in the latter. As it appears, computing the rounding error compensation is negligible compared to the verification process. The `recmip` and `mipplus` algorithms are consistently faster, as they implement the improved ReLU stability check. The difference between these algorithms is also the rounding error compensation, which is only implemented by `recmip`. Again, there is no significant difference in runtime, the heuristic rounding error compensation is very cheap to compute.

The results suggest that in average cases the improved bound check yields a consistent speedup. This can most likely be transferred to any algorithm with a similar architecture to MIPVerify. Also, the heuristic rounding error compensation – which requires interval arithmetic evaluation of complete network layers – adds only a negligible overhead to the total computation cost. The additional workload is on par

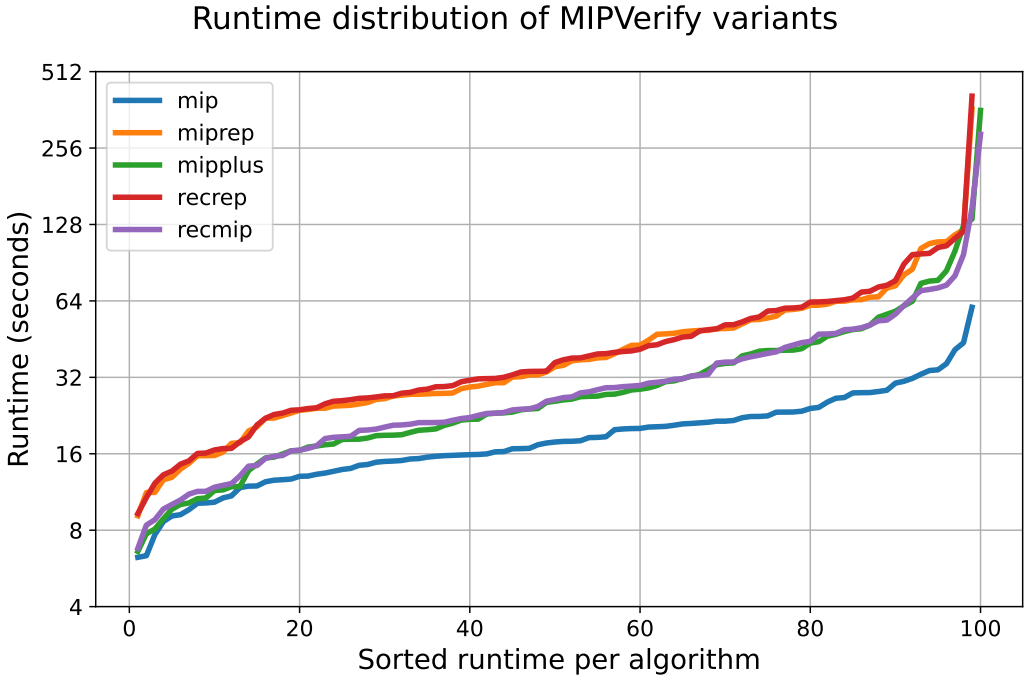

Figure 3.4: *Runtime distribution of different MIPVerify versions evaluating the first 100 MNIST test set samples on the WK17a neural network.*

with the simplified version of the more robust rounding error compensations described in Section 3.5.3 and Section 3.5.4. Hopefully, the described incomplete but strong rounding error compensation will also prove to be low cost, providing an effective heuristic defense against attacks based on rounding errors.

3.5.7 Results on reliability

The `recmip` algorithm does not implement the comprehensive rounding error compensations described, only a much simpler heuristic version. It is still not reliable in finding the backdoor region in the *WK17a-adv* network. However, on the first 100 MNIST test set samples it consistently reports the `POSSIBLY_ADVERSARIAL` verification result. This is likely caused by numeric errors in the MILP solution process. The verification outputs show that the final network model solving step found a feasible adversarial solution, however when verifying its classification, the input was not found to be adversarial. This is consistent with the distance to the adversarial candidate, only 9 out of 100 results had a distance larger than 0.05, which is the distance to the active backdoor region. This result is promising regarding the search for backdoors based on rounding errors, even a simple heuristic was able to reliably hint at the existence of a backdoor.

3.6 Conclusions

The chapter presented findings and thoughts on neural network verification. As we have seen, verification is a complicated topic where algorithms with strong guarantees are hard to create. Errors in modeling are non-trivial to recognize, and are even harder to mitigate.

Performant algorithms strongly depend on MILP solvers, leading to severe issues in reliability, as MILP solvers are not designed to be fully robust against numeric issues. A trivial size and potent adversarial network was presented (Section 3.4.4), which demonstrated that errors of arbitrary magnitude are possible. A real-world adversarial network was also presented, that successfully uses the exploit to hide some of its behavior from verification (Section 3.4.5).

In Section 3.4.7 a trivial to execute defense was introduced that is capable of defeating the exploit in its current form, by either eliminating the backdoor or making it visible to the previously fooled verification algorithm.

Even though the ideas and results presented in Section 3.5 are part of currently unfinished research, they offer an outlook on what is possible to achieve in the next iteration of interval arithmetic and MILP solver based verifier algorithms.

These preliminary results shed light on issues in the currently used verification algorithms. The usability of interval arithmetic for rigorous verification was discussed in Section 3.5.1, where previously undocumented limitations are highlighted on naively bounding neural network outputs with interval arithmetic.

In Section 3.5.2, Section 3.5.3, and Section 3.5.4 different failure modes of MILP solver based verification were discussed. A cheap heuristic defense is introduced against underestimating the rounding errors when a MILP optimum value is calculated in the network model. Another heuristic defense is introduced against rounding errors in the MILP constraint calculation process. These improvements aim to eliminate the underestimation of reachable model states, which can easily lead to suboptimal points being reported by the MILP solver as valid solutions. Consequently, the chance to exploit the erroneous network model should be significantly lowered, alongside errors of arbitrary magnitude caused by malicious ReLU networks.

In Section 3.5.5 an efficiency improvement is presented for algorithms with similar architecture to MIPVerify. A more efficient ReLU stability check is discussed, that can significantly reduce the number of necessary MILP optimization processes. Section 3.5.6 and Section 3.5.7 discuss the results obtained by evaluating different versions of MIPVerify on a sizeable network. While the original algorithm version is the fastest, there are promising signs for the version with improved reliability to operate with a negligible slowdown. Besides the additional cost of stronger bound computations, the more efficient ReLU modeling could even lead to both a more robust and faster algorithm.

A key takeaway of the presented research is that verification algorithms can be misleading. We often believe them to be more secure and reliable than they actually are. This emphasizes the need for thorough evaluation of their capacity to prove statements definitively.

Even though the partial results of Section 3.5 do not have the capacity for guaranteed definitive proofs, they show a promising path forward. The lack of an any-order expression bounding algorithm does not mean that we cannot improve the current best verification algorithms. While they still have no proving strength in all cases, reliability of these methods should be significantly improved as many error sources are eliminated by the corrections.

Publications

Journal publications

A. Mester, D. Zombori, L. Pál, and B. Bánhelyi. Efficiency improvement of the GLOBAL optimization method by local search changes. *Acta Polytechnica Hungarica*, 19(2):29–42, 2022

D. Zombori and B. Bánhelyi. Effects of pooling in ParallelGlobal with low thread interactions. *Informatica*, 45(2):191–196, 2021

Full papers in conference proceedings

B. Bánhelyi, T. Csendes, B. Lévai, D. Zombori, and L. Pál. Improved versions of the GLOBAL optimization algorithm and the GlobalJ modularized toolbox. In *AIP Conference Proceedings*, volume 2070, no. 020022, 2019

D. Zombori and B. Bánhelyi. ParallelGlobal with low thread interactions. In *Proceedings of the 22nd International Multiconference Information Society*, volume I, pages 83–86, 2019

D. Zombori, B. Bánhelyi, T. Csendes, I. Megyeri, and M. Jelasity. Fooling a complete neural network verifier. In *9th International Conference on Learning Representations*. `https://openreview.net/pdf?id=4IwieFS44l`, 2021

Further related publications

B. Bánhelyi, T. Csendes, B. Lévai, L. Pál, and D. Zombori. *The GLOBAL Optimization Algorithm*. Springer Briefs in Optimization, 2018

Summary

Optimization tasks have many faces. As we have seen, interactions between the optimizer and the optimized model can be non-intuitive and lead to problems. An abnormally high number of function evaluations, and underutilized computing threads hint at harmful interplay between the optimizer and the optimized function. Depending on the requirements for an acceptable solution, a number of different challenges can arise. The amount of tolerated error in the optimum and the acceptable computational cost will limit the choice in the available optimizers.

In my dissertation I discussed the multi-threaded implementations of a global optimizer, where the main challenges concerned balancing between algorithmic efficiency and performance, and interplay of the multi-threaded algorithm and objective function characteristics. Regarding the usage of MILP optimizers to perform verification tasks, I discussed the mismatch between an optimization model and the real system, the reliability of optimal solutions, alongside the applicability of interval arithmetic in verification.

Parallel global optimization

The Global algorithm originates from research on stochastic multi-start optimization. As it is an effective derivative-free method to solve various black-box problems, research was also performed on it in the past few years, improving the algorithm. To ease the usage and development of Global, an optimizer toolbox was created capable of configuring and assembling algorithm components. Easy reusability enabled the development of algorithm variants that better suit specific optimization tasks. One area where the toolbox and specifically the Global algorithm had room for improvement was the utilization of a multi-threaded computing environment. Creating an effective multi-thread Global variant is not a trivial task, it has lead to the research presented in this dissertation.

In Chapter 2 three approaches are discussed to create multi-threaded algorithm versions of Global, which in general are applicable to similar multi-start optimization methods.

The SynchronizedGlobal version has a close resemblance to the original Global algorithm, on a single thread the two algorithms closely match. While execution happens on multiple threads, SynchronizedGlobal tries to preserve the main characteristics of Global. In effect SynchronizedGlobal tries to utilize the available threads, but it prioritizes the reduction in the number of function evaluations when the two are in direct conflict.

The ParallelGlobal version aims to maximize the utilization of computing power while trying to incorporate efficiency improvements applied in the single thread algorithm. Rather than waiting for other threads, the evaluation does not stop, at any point the best available information is used to guide the algorithm.

The idea of a fully distributed version of Global is captured in DistributedGlobal, providing the means to run Global threads in a loosely coupled computing environment. When a single computer does not have enough resources to perform an optimization task in acceptable time, the computations can be executed on a distributed system. Communication times in these systems can easily dominate the evaluation time of cheaper objective functions, therefore simply delegating the evaluations would not always be efficient. Instead, the workers can run on compute nodes with loose coupling, where information is disseminated asynchronously, without blocking other workers. In addition, DistributedGlobal does not necessarily need a central information authority, results can be shared using distributed algorithms.

The algorithms were tested to assess how successful they are at decreasing optimization time, increasing the algorithm performance, and how closely they match the efficiency of the single thread version.

Tests on SynchronizedGlobal showed that the algorithm can be effective at reducing the necessary runtime. Given a large enough task per thread, the multi-threaded implementation can be more effective, while an overabundance of threads will cause slowdown. Using SynchronizedGlobal with a multi-threaded configuration was found to increase the number of function evaluations. This is countered with a significant decrease in runtime, when the objective function is computationally expensive.

Tests on ParallelGlobal showed a much more varied result. While objective functions favoring the algorithm are solved quickly and efficiently, some objective functions

posed a serious issue. The reduced algorithmic efficiency of ParallelGlobal – depending on the objective function – has lead from 0% to 1500% increase in the number of function evaluations, heavily affecting the runtime. For some objective functions the single thread Global algorithm was faster, even on the most expensive evaluation setting.

In conclusion, multi-thread implementation of multi-start optimizers is viable and yields useful algorithms. Optimizers are sensitive to the characteristics of the objective function, in multi-thread algorithms these effects are even stronger. For some problems they provide a valuable increase in performance, for others they can perform worse than the single-thread version.

Problems and solutions in neural network verification

Neural networks are versatile tools to solve complex problems, where requirements are rather fuzzy in nature than precise. While large models are out of reach for useful verification, the small to mid-sized networks are on the edge of possible verification. These models are large enough to perform single cutout tasks, like character recognition or collision avoidance control. Since these networks can be used as building blocks in more complex algorithms, it is important to acquire reliable implementations. Recently, numerous verification algorithms were proposed to ensure correctness of these networks. As we found, there are two categories of verification algorithms, direct solver algorithms that are not fast enough to verify 1000 neuron networks, and fast algorithms that use MILP solvers to at least aid, but more often to solve verification tasks. It is well known that numeric algorithms are prone to numeric errors, MILP solvers also suffer from this issue. Directly using results from a MILP solver to decide reachability of adversarial inputs is therefore dangerous.

In Chapter 3 MIPVerify – a verifier relying on MILP solvers – is examined in detail, to provide a showcase of exploiting the known issues. Numeric issues are uncovered in its implementation regarding modeling of the network and the output reachability. An exploit for MIPVerify+Gurobi is presented, that an attacker could use to evade detection of a backdoor during verification, while freely choosing the network output in a subspace. The exploit can be fine-tuned for other verifier+solver combinations, as long as they are sensitive to the numeric issues affecting the MILP solver. A two part neural network is presented, that performs classification on the MNIST dataset, and has an attached backdoor. Tests showed that MIPVerify reliably misses the ad-

versarial behavior in the tampered network, and incorrectly yields the same result as for the clean version.

To counter the attack, a defense is introduced that is trivial to execute, cheap to apply, and can disrupt exploits based on perfectly canceling numbers. It however does not defend against classic problems in neural network verification.

In addition to discovering potential issues with using MILP solvers in verification, the usability of interval arithmetic is also discussed. The findings uncover a limitation of interval arithmetic, that affects reliability of verifiers, and which the verifier implementations consistently ignore.

Potential ways to improve robustness of MILP models in verification is discussed. The main focus is on countering numeric issues that can affect the calculation of an optimum point, and issues that can lead to numerically unstable models by affecting constraints. Heuristics are introduced that can render MILP models more robust against numeric errors.

A simple and widely applicable improvement was found in verification of ReLU networks using MILP models. By better utilization of the available information, a significant reduction in MILP model solves is achieved, without compromising reliability. In verification the majority of computing power is used by the MILP model solves, the speedup is therefore significant.

In conclusion, verification of neural networks is still not a solved problem. Reliable solvers can only handle very small networks, while verifiers claiming to tackle larger networks have serious flaws that prevent reliable verification. As MILP solvers are unlikely to improve in their sensitivity to numeric issues, other approaches are needed to speed up reliable verification, or harden fast and currently unreliable verifiers against exploits.

Összefoglalás

Az optimalizálási feladatok sokrétűek. Ahogy az előző fejezetekben láttuk, az optimalizáló algoritmus és az optimalizált probléma közötti bonyolult kölcsönhatások problémákhoz vezethetnek. A függvénykiértékelések kiugróan magas száma és a kihasználatlan processzorszálak ilyen problémákra utalnak. Attól függően, hogy egy potenciális megoldásnak milyen követelményeket kell teljesítenie, számos kihívás adódhat. Az optimális megoldásban megtűrt hiba mértéke és a rendelkezésre álló számítási teljesítmény behatárolja a felhasználható optimalizáló algoritmusokat.

Disszertációmban tárgyaltam egy globális optimalizáló többszálas implementációját, ahol legfőbb kihívásként az algoritmus teljesítmény és hatékonyság közötti egyensúly, valamint a többszálas algoritmus és a célfüggvény közötti kölcsönhatás jelent meg. A vegyes egészértékű lineáris optimalizálókon alapuló neuronháló verifikálás témakörben tárgyaltam az optimalizált modell és a valós rendszer közötti különbségek problémáját, valamint az optimálisnak gondolt megoldások megbízhatóságát. Az intervallum aritmetika felhasználásával kapcsolatban is történt egy kitekintés.

Párhuzamos globális optimalizálás

A Global algoritmus a sztohasztikus multi-start optimalizálók kutatásából származik. Mivel egy hatékony algoritmusról van szó, ami az ismeretlen célfüggvények optimalizálásához nem igényli a derivált függvény meglétét, az utóbbi években is születtek eredmények az algoritmussal kapcsolatban. Hogy a Global könnyebben fejleszthető és felhasználható legyen, egy olyan optimalizáló toolbox lett létrehozva, ami konfigurálható komponensekből építi fel az algoritmust. A könnyű újra-felhasználhatóság lehetővé tette a konkrét optimalizálási feladatokra szabott algoritmus létrehozását. A toolbox és a Global algoritmus továbbfejlesztésére a többszálas számítási környezet kihasználásában nyílt lehetőség. Egy többszálas Global verzió megalkotása nem triviális feladat, eredménye a disszertációban bemutatott kutatás.

A 2. fejezet a többszálas Global algoritmus megalkotására három megközelítést tárgyal, amik alkalmazhatók más, hasonló felépítésű optimalizálóknál.

A SynchronizedGlobal verzió az eredeti Global algoritmushoz nagyon hasonló, egy szálon futtatva a kettő megegyezik. A SynchronizedGlobal algoritmus többszálas futtatás esetén igyekszik megőrizni a Global lehető legtöbb tulajdonságát. A teljesítmény növeléséért a processzorszálak legnagyobb kihasználtságára törekszik, azonban a célfüggvény kiértékelések számának csökkentése prioritást élvez. Ha ez a két hatás konfliktusban áll, utóbbit részesíti előnyben.

A ParallelGlobal többszálas verzió a processzorszálak legnagyobb kihasználtságát részesíti előnyben, miközben az egyszálas algoritmus verzió hatékonyságot növelő megoldásait is igyekszik integrálni. Ahelyett hogy a szálak egymásra várakoznának, a pillanatnyilag elérhető információk alapján hoznak döntést.

A teljesen elosztott rendszeren futtatható Global ötletét a DistributedGlobal algoritmus írja le, lehetővé téve a Global algoritmus szálak gyengén összefüggő rendszereken történő futtatását. Amennyiben egy számítógép nem rendelkezik elég számítási kapacitással egy optimalizálási feladat kivárható elvégzéséhez, a számítások több, hálózatba kötött számítógépen is elvégezhetők. Ilyen hálózatokban a kommunikációra szánt idő könnyű célfüggvényeken meghaladhatja a célfüggvény kiértékelésekre szánt időt, ezért a kiértékelések delegálása nem feltétlenül hatékony. Ehelyett a szálak között laza a kapcsolat, külön számítógépeken futnak. A szálak közötti kommunikációt elosztott információmegosztó algoritmusok végzik, az optimalizáló szálak akadályozása nélkül. Az elosztott információmegosztó algoritmusok miatt a DistributedGlobal-nak nincs szüksége központi szerverre.

Az többszálas algoritmus tesztek vizsgálták az optimalizáláshoz szükséges idő változását, az algoritmus teljesítményének növekedését és az algoritmus hatékonyságát az egyszálas teljesítményhez képest.

A SynchronizedGlobal algoritmuson futtatott tesztek megmutatták, hogy az képes hatékonyan csökkenteni a szükséges futási időt. Amennyiben a szálakra megfelelő méretű feladat jut, a többszálas algoritmus növeli a teljesítményt, miközben túl kis részproblémák az egyszálas verzióhoz képest is rosszabb eredményt érhetnek el. A SynchronizedGlobal algoritmus növeli a szükséges függvénykiértékelések számát, ennek ellenére nagyobb számításigényű célfüggvényeken a teljesítmény növekedése csökkentett futásidőt eredményez.

A ParallelGlobal algoritmuson végzett tesztek nagyobb változatosságot mutattak. Miközben az algoritmusnak kedvező célfüggvények kiértékelése gyors és ha-

tékony, néhány célfüggvény jelentős problémát okoz. Az algoritmus hatékonysága célfüggvénytől függően csökkent, 0% és 1500% közötti növekedést okozva a kiértékelések számában, ami jelentősen növelte a futási időt is. Néhány célfüggvényen még a legnagyobb számítási igény esetén is az eredeti Global verzió volt gyorsabb.

Levonható konklúzió, hogy a multi-start optimalizálók többszálas implementációja kivitelezhető és hasznos algoritmusokat eredményez. Az optimalizálók érzékenyek a célfüggvény tulajdonságaira, többszálas algoritmusok esetén ez az érzékenység még erősebben fennáll. Bizonyos problémák esetén a többszálas algoritmusok képesek növelni a hasznos teljesítményt, más problémákon még az egyszálas verziónál is rosszabb eredményt érnek el.

Problémák és megoldások a neurális hálók verifikációjában

A neurális hálók sokrétű eszközök bonyolult problémák megoldására, amik inkább zavaros mint egyértelmű követelményrendszerrel rendelkeznek. A nagy modellek elérhetetlen távolságban vannak verifikáció szempontjából, azonban a kis és közepes modellek az értelmes számítási kapacitás határán vannak. Ezek a modellek képesek egy behatárolt feladat elvégzésére, mint a karakterfelismerés, vagy az ütközéselkerülési vezérlés. Mivel ezek a neurális hálók nagyobb rendszerek építőelemeiként funkcionálnak, fontos hogy a megbízhatóságuk ellenőrizhető legyen. A közelmúltban számos verifikáló algoritmus született az ilyen hálók helyességének ellenőrzésére. Mint kiderült, ezek az algoritmusok két csoportra oszthatók. A direkt algoritmusok, amik potenciálisan megbízhatóak, de nincs elég teljesítményük 1000 neuronból álló hálózatok ellenőrzésére sem, valamint a vegyes egészértékű lineáris modelleken alapuló algoritmusok, amik nagy teljesítményű, azonban nem megbízható algoritmusok. Közismert tény, hogy a numerikus módszerek alkalmazásánál a numerikus hibák elkerülhetetlenek, a vegyes egészértékű megoldók szintén kitettek a problémának. Emiatt direktben egy megoldó által szolgáltatott eredményre alapozva eldönteni az adverzális halmaz elérhetőségét veszélyes.

A 3. fejezet a vegyes egészértékű (MILP) modelleken alapuló MIPVerify algoritmust részletesen vizsgálja, hogy az említett támadási felületek kihasználására egy mintapéldát adjon. Az algoritmus implementációjában a numerikus problémák nem megfelelő kezelésére derült fény, ami befolyásolja a neurális háló modellezését és az elérhető kimeneti halmaz helyességét. Egy sebezhetőség kerül bemutatásra a

MIPVerify+Gurobi verifikálón, amit kihasználva egy támadó elkerülheti a beépített hátsó ajtók felfedezését, miközben az adverzális altérben a neurális háló kimenetét szabadon megválaszthatja. A sebezhetőség más – akár több – verifikáló és MILP megoldó kombinációra is hangolható. Bemutatásra kerül egy két részből álló neuronháló, ami az MNIST adathalmazon végez osztályozást és rendelkezik egy beépített hátsó ajtóval. Tesztekkel igazoltuk, hogy a MIPVerify nem veszi észre a hátsó ajtót és hibásan ugyanazt a választ adja mint a hátsó ajtóval nem rendelkező neuronháló esetén.

A sebezhetőség javítására egy védekezés is bemutatásra kerül, ami könnyen alkalmazható, minimális számításigényű és képes megzavarni az egymással tökéletesen megegyező számpáron alapuló támadásokat. A neuronhálók viselkedésében fellépő klasszikus problémákkal szemben azonban nem alkalmazható.

A MILP megoldók verifikációban történő problémás felhasználásán túl az intervallum aritmetika alkalmazhatóságával kapcsolatban is kérdések merültek fel. Az eredmények egy fontos korlátozást mutatnak, ami befolyásolja a verifikálók megbízhatóságát és amit a vizsgált verifikálók rendre megszegnek.

A MILP megoldók megbízhatóságának javítására több lehetőség is felmerült. Főbb pontok a numerikus problémák kezelése a MILP optimum kiszámításánál és a numerikus problémák miatt instabil modellekhez vezető MILP korlátok javítása. A felmerült problémák enyhítésére heurisztikus algoritmusok lettek javasolva.

Egy egyszerű, széles körben alkalmazható algoritmus fejlesztést is bemutatunk, ami a ReLU hálózatok MILP modellen alapuló verifikációjában jelent előrelépést. A rendelkezésre álló információ hatékonyabb kihasználásával a MILP modell megoldások számában sikerült jelentős csökkenést elérni, a megbízhatóságot nem csökkentve. Mivel a verifikáció számítási igényének túlnyomó részét a MILP megoldások adják, a sebességnövekedés szignifikáns.

Levonható konklúzió, hogy a neurális hálók verifikációja nem megoldott probléma. A megbízható verifikálók csak kis méretű neuronhálókat képesek kezelni, miközben a nagyobb hálók kezelésére is alkalmas verifikálók jelentős sebezhetőségekkel rendelkeznek, így az általuk adott válaszok nem megbízhatóak. Mivel a MILP megoldók nem várható hogy jelentős javulást érnek el a numerikus hibák elleni küzdelemben, más megközelítés szükséges hogy felgyorsítsuk a megbízható verifikálókat, vagy megbízhatóbbá tegyük a gyors, de nem megbízható verifikálókat.

Bibliography

[1] José-Oscar H Sendín, Julio R Banga, and Tibor Csendes. Extensions of a multistart clustering algorithm for constrained global optimization problems. *Industrial & Engineering Chemistry Research*, 48(6):3014–3023, 2009.

[2] George B Dantzig. Maximization of a linear function of variables subject to linear inequalities. *Activity analysis of production and allocation*, 13:339–347, 1951.

[3] Narendra Karmarkar. A new polynomial-time algorithm for linear programming. In *Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 302–311, 1984.

[4] Victor Klee and George J Minty. How good is the simplex algorithm. *Inequalities*, 3(3):159–175, 1972.

[5] Daniel A Spielman and Shang-Hua Teng. Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time. *Journal of the ACM (JACM)*, 51(3):385–463, 2004.

[6] Antoine Deza, Eissa Nematollahi, and Tamás Terlaky. How good are interior point methods? Klee–Minty cubes tighten iteration-complexity bounds. *Mathematical Programming*, 113(1):1–14, 2008.

[7] Jeffrey T Linderoth and Andrea Lodi. MILP software. *Wiley encyclopedia of operations research and management science*, 5:3239–3248, 2010.

[8] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual. `https://www.gurobi.com`, 2023.

[9] IBM. CPLEX User's Manual. `https://www.ibm.com/docs/en/icos/22.1.1`, 2023.

[10] John Forrest, Ted Ralphs, Stefan Vigerske, Haroldo Gambini Santos, John Forrest, Lou Hafer, Bjarni Kristjansson, jpfasano, EdwinStraver, Miles Lubin, Jan-Willem, rlougee, jpgoncal1, Samuel Brito, h-i-gassmann, Cristina, Matthew Saltzman, tosttost, Bruno Pitrus, Fumiaki Matsushima, and to-st. coin-or/Cbc: Release releases/2.10.11. `https://doi.org/10.5281/zenodo.10041724`, 2023.

[11] GNU project. GNU Linear Programming Kit, Version 5.0. `http://www.gnu.org/software/glpk/glpk.html`, 2023.

[12] Ksenia Bestuzheva, Mathieu Besançon, Wei-Kun Chen, Antonia Chmiela, Tim Donkiewicz, Jasper van Doornmalen, Leon Eifler, Oliver Gaul, Gerald Gamrath, Ambros Gleixner, Leona Gottwald, Christoph Graczyk, Katrin Halbig, Alexander Hoen, Christopher Hojny, Rolf van der Hulst, Thorsten Koch, Marco Lübbecke, Stephen J. Maher, Frederic Matter, Erik Mühmer, Benjamin Müller, Marc E. Pfetsch, Daniel Rehfeldt, Steffan Schlein, Franziska Schlösser, Felipe Serrano, Yuji Shinano, Boro Sofranac, Mark Turner, Stefan Vigerske, Fabian Wegscheider, Philipp Wellner, Dieter Weninger, and Jakob Witzig. Enabling research through the SCIP Optimization Suite 8.0. *ACM Trans. Math. Softw.*, 49(2), 2023.

[13] Timothy J Callahan and John Wawrzynek. Instruction-level parallelism for reconfigurable computing. In *FPL*, volume 98, pages 248–257, 1998.

[14] Bumyong Choi, Leo Porter, and Dean M Tullsen. Accurate branch prediction for short threads. In *Proceedings of the 13th International conference on Architectural support for programming languages and operating systems*, pages 125–134, 2008.

[15] Mageda Sharafeddine, Komal Jothi, and Haitham Akkary. Disjoint out-of-order execution processor. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(3):1–32, 2012.

[16] Paul MB Vitányi. Locality, communication, and interconnect length in multicomputers. *SIAM Journal on Computing*, 17(4):659–672, 1988.

[17] Igor L Markov. Limits on fundamental limits to computation. *Nature*, 512(7513):147–154, 2014.

[18] Gabriel H Loh, Natalie Enright Jerger, Ajaykumar Kannan, and Yasuko Eckert. Interconnect-memory challenges for multi-chip, silicon interposer systems. In

Proceedings of the 2015 international symposium on Memory Systems, pages 3–10, 2015.

[19] Ramon E Moore. *Interval arithmetic and automatic error analysis in digital computing*. PhD thesis, Department of Mathematics, Stanford University, 1962.

[20] G William Walster. Philosophy and practicalities of interval arithmetic. In *Reliability in Computing*, pages 309–323. Elsevier, 1988.

[21] Ramon E Moore, R Baker Kearfott, and Michael J Cloud. *Introduction to interval analysis*. SIAM, 2009.

[22] Arnold Neumaier. *Introduction to numerical analysis*. Cambridge University Press, 2001.

[23] Siegfried M Rump. Intlab—interval laboratory. In *Developments in reliable computing*, pages 77–104. Springer, 1999.

[24] Stefano Taschini. Interval arithmetic: Python implementation and applications. In Gaël Varoquaux, Travis Vaught, and Jarrod Millman, editors, *Proceedings of the 7th Python in Science Conference*, pages 16 – 21, Pasadena, CA USA, 2008.

[25] David P. Sanders, Benoît Richard, Olivier Hénot, Luis Benet, Luca Ferranti, Krish Agarwal, Petr Vana, Josua Grawitter, Michael F. Herbst, Marcelo Forets, Eeshan Gupta, yashrajgupta, Eric Hanson, Braam van Dyk, Christopher Rackauckas, Rushabh Vasani, Sebastian Micluța-Câmpeanu, Sheehan Olver, Twan Koolen, Vincent Tjeng, Caroline Wormell, Daniel Karrasch, David Widmann, Favio André Vázquez, Guillaume Dalle, Jeffrey Sarnoff, Julia TagBot, Kevin O'Bryant, Kristoffer Carlsson, and Morten Piibeleht. JuliaIntervals/IntervalArithmetic.jl: v0.22.2. `https://doi.org/10.5281/zenodo.10417993`, 2023.

[26] IEEE Standard for Interval Arithmetic. *IEEE Std 1788-2015*, pages 1–97, 2015.

[27] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, 2019.

[28] Aimo Törn and Antanas Žilinskas. *Global optimization*, volume 350 of *Lecture Notes in Computer Science*. Springer, 1989.

[29] L. C. W. Dixon and M. Jha. Parallel algorithms for global optimization. *Journal of Optimization Theory and Applications*, 79:385–395, 1993.

[30] Sudhir Ahuja, Nicholas Carriero, and David Gelernter. Linda and Friends. *Computer*, 19(08):26–34, 1986.

[31] A. J. G. Hey and T. D. J. Pritchard. Parallelism in scientific programming and its efficient implementation on transputer arrays. *ESPRIT P1085 Report MO*, 1987.

[32] Laurent Cohen. Java Parallel Processing Framework. `https://github.com/jppf-grid/JPPF`, 2020.

[33] Tibor Csendes. Nonlinear parameter estimation by global optimization – Efficiency and reliability. *Acta Cybernetica*, 8(4):361–370, 1988.

[34] C. G. E. Boender, A. H. G. Rinnooy Kan, G. T. Timmer, and L. Stougie. A stochastic method for global optimization. *Mathematical Programming*, 22:125–140, 1982.

[35] B. Bánhelyi, T. Csendes, B. Lévai, L. Pál, and D. Zombori. *The GLOBAL Optimization Algorithm*. Springer Briefs in Optimization, 2018.

[36] B. Bánhelyi, T. Csendes, B. Lévai, D. Zombori, and L. Pál. Improved versions of the GLOBAL optimization algorithm and the GlobalJ modularized toolbox. In *AIP Conference Proceedings*, volume 2070, no. 020022, 2019.

[37] D. Zombori and B. Bánhelyi. ParallelGlobal with low thread interactions. In *Proceedings of the 22nd International Multiconference Information Society*, volume I, pages 83–86, 2019.

[38] D. Zombori and B. Bánhelyi. Effects of pooling in ParallelGlobal with low thread interactions. *Informatica*, 45(2):191–196, 2021.

[39] Paulo Jesus, Carlos Baquero, and Paulo Sergio Almeida. A survey of distributed data aggregation algorithms. *IEEE Communications Surveys & Tutorials*, 17(1):381–404, 2015.

[40] Mária Csete, D Vass, A Szenes, B Bánhelyi, T Csendes, and G Szabó. Plasmon enhanced fluorescence characteristics government by selecting the right objective function. In *Proceedings of the COMSOL Conference*, 2018.

[41] Emese Tóth, Ditta Ungor, Tibor Novák, Györgyi Ferenc, Balázs Bánhelyi, Edit Csapó, Miklós Erdélyi, and Mária Csete. Mapping fluorescence enhancement of plasmonic nanorod coupled dye molecules. *Nanomaterials*, 10(6):1048, 2020.

[42] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint https://arxiv.org/abs/1312.6199*, 2014.

[43] Jiawei Su, Danilo Vasconcellos Vargas, and Kouichi Sakurai. One pixel attack for fooling deep neural networks. *IEEE Transactions on Evolutionary Computation*, 23(5):828–841, 2019.

[44] Nicolas Papernot, Patrick McDaniel, Xi Wu, Somesh Jha, and Ananthram Swami. Distillation as a defense to adversarial perturbations against deep neural networks. In *2016 IEEE symposium on security and privacy (SP)*, pages 582–597. IEEE, 2016.

[45] Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *2017 IEEE symposium on security and privacy (sp)*, pages 39–57. IEEE, 2017.

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

[47] Vincent Tjeng, Kai Xiao, and Russ Tedrake. Evaluating robustness of neural networks with mixed integer programming. In *7th International Conference on Learning Representations*. https://arxiv.org/abs/1711.07356, 2019.

[48] Guy Katz, Clark Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. Reluplex: An efficient SMT solver for verifying deep neural networks. In Rupak Majumdar and Viktor Kunčak, editors, *Computer Aided Verification*, pages 97–117. Springer International Publishing, 2017.

[49] Nicholas Carlini, Guy Katz, Clark Barrett, and David L Dill. Ground-truth adversarial examples. https://openreview.net/forum?id=Hki-ZlbA-, 2018.

[50] Alessio Lomuscio and Lalit Maganti. An approach to reachability analysis for feed-forward ReLU neural networks. *arXiv e-prints https://arxiv.org/abs/1706.07351*, 2017.

[51] William Cook, Thorsten Koch, Daniel E Steffy, and Kati Wolter. An exact rational mixed-integer programming solver. In *Integer Programming and Combinatoral*

Optimization: 15th International Conference, IPCO 2011, New York, NY, USA, June 15-17, 2011. Proceedings 15, pages 104–116. Springer, 2011.

[52] Eric Wong and Zico Kolter. Provable defenses against adversarial examples via the convex outer adversarial polytope. In *International conference on machine learning*, pages 5286–5295. PMLR, 2018.

[53] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[54] D. Zombori, B. Bánhelyi, T. Csendes, I. Megyeri, and M. Jelasity. Fooling a complete neural network verifier. In *9th International Conference on Learning Representations*. `https://openreview.net/pdf?id=4IwieFS44l`, 2021.

[55] A. Mester, D. Zombori, L. Pál, and B. Bánhelyi. Efficiency improvement of the GLOBAL optimization method by local search changes. *Acta Polytechnica Hungarica*, 19(2):29–42, 2022.

List of Figures

List of Tables