BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FORMAL ANALYSIS OF NEURAL NETWORKS

FORMÁLNÍ METODY PRO ANALÝZU NEURONOVÝCH SÍTÍ

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR DAVID HUDÁK
AUTOR PRÁCE

SUPERVISOR Doc. RNDr. MILAN ČEŠKA, Ph.D.
VEDOUCÍ PRÁCE

BRNO 2021

Brno University of Technology
Faculty of Information Technology

Department of Intelligent Systems (DITS) Academic year 2021/2022

Bachelor's Thesis Specification

24620

Student:	**Hudák David**
Programme:	Information Technology
Title:	**Formal Analysis of Neural Networks**
Category:	Formal Verification

Assignment:

1. Study the existing methods for formal analysis of neural networks (NNs). Focus on methods for symbolic analysis of local robustness of NNs.
2. Evaluate these methods in the context of scalable analysis of practically relevant NNs.
3. Design improvements and extensions of these methods. Focus on different approaches for symbolic representation of NNs and generating adversary inputs.
4. Implement the proposed improvements and extensions on top of an existing tool (e.g. VeriNet or CROWN)
5. Perform a detailed experimental evaluation of the proposed methods on a suitable benchmark.

Recommended literature:

- Liu, C., Arnon, T., Lazarus, C., Strong, C., Barrett, C. and Kochenderfer, M.J. Algorithms for Verifying Deep Neural Networks. *Foundations and Trends in Optimization*. 2020.
- Henriksen, P. and Lomuscio, A. Efficient neural network verification via adaptive refinement and adversarial search. In *ECAI 2020*.
- Henriksen, P. and Lomuscio, A. DEEPSPLIT: An Efficient Splitting Method for Neural Network Verification via Indirect Effect Analysis. In *IJCAI 2021*.
- Xu, K., Zhang, H., Wang, S., Wang, Y., Jana, S., Lin, X. and Hsieh, C.J. Fast and Complete: Enabling Complete Neural Network Verification with Rapid and Massively Parallel Incomplete Verifiers. In *ICLR 2020*.

Requirements for the first semester:

- Items 1, 2, and partially 3.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Češka Milan, doc. RNDr., Ph.D.**
Head of Department:	Hanáček Petr, doc. Dr. Ing.
Beginning of work:	November 1, 2021
Submission deadline:	May 11, 2022
Approval date:	November 3, 2021

Abstract

Today, the area where we can use deep learning is becoming broader. It includes safety-critical domains such as traffic or healthcare, and the need for its verification grows. However, sufficient verification toolkits for neural networks, the leading deep learning approach, are still in development. State-of-the-art algorithms now can not verify commonly used deep networks. In this paper, we focus on one of the state-of-the-art solutions, VeriNet. More generally, we focused on the symbolic approach of local robustness analysis. This approach usually relies on creating, processing, and refining the neural network representation, and we focused on the refinement phase. We primarily dealt with the branch and bound algorithm, which in this toolkit splits node inputs in a network to create smaller sub-problems. For this algorithm, we proposed and implemented new split node selection strategies. Specifically, we designed memory-based, alternating, and semi-hierarchical strategies. We achieved significant improvements in the scalability of the VeriNet toolkit. One of our approaches can solve more complex cases and significantly improve already solved cases' performance. Moreover, we discovered an anomaly in the behavior of the verification algorithm we named branch implosions, which led to extreme speed up for some cases. In addition, we extended the set of performed network benchmarks with models from the Marabou package.

Abstrakt

Škála oblastí, ve kterých se dnes můžeme setkat s hlubokým učením, se velmi rychle rozrůstá. Zasahuje už dokonce i mezi bezpečnostně kritické oblasti jako doprava či lékařství, a tak narůstá nutnost takové systémy verifikovat. Nicméně, dostatečně škálovatelné nástroje pro verifikaci neuronových sítí, které tvoří hlavní přístup k hlubokému učení, jsou stále ve vývoji. Dnešní řešení tak nejsou schopny verifikovat dostatečně hluboké sítě. Z toho důvodu jsme se zaměřili na jeden ze současných nástrojů, VeriNet, a pokusili jsme jej vylepšit. Obecněji jsme se zaměřili na symbolický přístup k analýze lokální robustnosti. Tento přístup běžně spočívá na vytvoření, zpracování a přepracování reprezentace neuronové sítě, přičemž my jsme se zaměřili na fázi přepracování. Primárně jsme se zabývali algoritmem větví a mezí, který spočívá v rozdělování vstupů dílčích síťových uzlů k vytváření menších podproblémů. Specificky jsme navrhli nové paměťové, alternující a semi-hierarchické strategie. Při experimentování jsme dosáhli výrazných vylepšení nástroje VeriNet. Jeden z našich přístupů je tak schopen řešit více komplexních případů a také vylepšuje zpracování již řešitelných případů. K tomu jsme navíc narazili na anomálie pracovně nazvané jako imploze větví, které vedou k extrémnímu urychlení některých případů. V rámci této práce jsme také rozšířili set síťových benchmarků s modely z balíku nástroje Marabou.

Keywords

Neural network, ReLU, VeriNet, ESIP, branch and bound, splitting strategies, branch implosions, semi-hierarchical strategy, formal verification

Klíčová slova

Neuronová síť, ReLU, VeriNet, ESIP, metoda větví a mezí, strategie dělení, imploze větví, semi-hierarchická strategie, formální verifikace

Reference

HUDÁK, David. *Formal Analysis of Neural Networks.* Brno, 2021. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Doc. RNDr. Milan Češka, Ph.D.

Rozšířený abstrakt

Využití hlubokého učení, primárně pak neuronových sítí, se dnes dostává do čím dál více rutinních činnosti lidských životů. Na jedné straně to jsou neškodné oblasti jako fotografování, detekce obličejů, překladače či jiné pomocné nástroje, které usnadňují život. Na straně druhé jsou to i bezpečnostně kritické oblasti jako autonomní řízení vozidel, medicína, letecké kolizní modely či vojenství, kde naopak lze i drobnou chybou životy ztratit. S těmito disciplínami tak vzniká i potřeba je verifikovat, tedy nutnost formálně potvrdit jejich správné usuzování. To s ohledem k faktu, že neuronové sítě pracují v podstatě jako černá skříňka, je poměrně obtížné. V současné době je tak disciplína formální verifikace neuronových velmi dynamická, kdy každým rokem vychází řada nových a lépe škálovatelných nástrojů. Tato práce se pak zabývá konkrétně přístupem symbolické analýzy lokální robustnosti, jejímž výrazným představitelem je nástroj VeriNet, který je detailně popsán v této práci, a jeho rozšíření.

Zmíněná lokální robustnost je jednou z tradičně verifikovaných vlastností neuronových sítí, která je inspirována biologickými procesy. V případě živých organismů robustnost popisuje jako schopnost odolávat neideálním podmínkám. U neuronových sítí se jedná o vlastnost, že přijatelně velké odchylky na vstupech by neměly a nesmí změnit korektní klasifikaci na jinou. Například v případu autonomního řízení je naprosto nutné, aby zpracovávaný záznam z kamery, kde je například dopravní značka stop, nezměnil kvůli nějaké drobné odchylce výslednou klasifikaci na značku dálnice. Taková odchylka může být způsobená různými běžnými jevy – od mírných šumů způsobených kamerou, mírně vybledlé barvě červené na značce či změně světlosti kvůli různé denní době. Vůči těmto všem ochylkám by měla být ideálně robustní neuronová síť odolná.

K tomu, abychom tuto lokální robustnost mohli zkoumat, je nutné vytvořit vhodnou reprezentaci sítě, kterou se snažíme zkoumat. To je poměrně náročná disciplína, jelikož neuronová síť je sama o sobě v podsatě silně nelineární funkce. V případě snah o vytvoření přesné reprezentace tak dochází k problémům s příliš náročným modelem, které pak lze verifikovat jen ve velmi omezené míře. Proto začaly vznikat postupy, které nemají za cíl popsat naprosto přesně výstup neuronové sítě, nýbrž vytváří jeho vhodně nadhodnocenou a dobře zpracovatelnou verzi.

Jeden z těchto přístupů funguje skrze symbolickou reprezentaci sítí, která se řadí mezi metody propagace hranic. Ta spočívá k přiřazení konkrétních číselných hranic a dvou lineárních rovnic (horní a dolní hranice) ke každému uzlu. Jejich kombinace pak popisuje chování konkrétního uzlu, přičemž vytvářeny jsou skrze proces postupného propagování vstupních hodnot s povolenou odchylkou skrze síť přes jednotlivé uzly a vrstvy až po vrstvu výstupní. Díky tomu lze na výstupní vrstvě sítě určit, zdali uzel reprezentující korektní výstup má za každých podmínek ty nejvyšší hodnoty a je tak v rámci odchylek stanovených lokální robustností bezpečný. K vyřešení tohoto problému slouží vybraný vhodný solver (v případě VeriNetu Gurobi LP-solver), který zjišťuje, zda korektní uzel na výstupní vrstvě má pro všechny možnosti tu nejvyšší hodnotu.

V rámci VeriNetu pak bylo implementováno vylepšení ve formě symbolické intervalové propagace založené na chybě[1], jež reprezentuje dané uzly s použitím pouze jedné dolní lineární rovnice, oproti původním dvěma. Horní hranici reprezentuje s pomocí konkrétních chyb definujících vzdálenost horní relaxace.

Jak již ale bylo zmíněno, tato metoda se řadí mezi ty, jež nadhodnocují chování sítí. To což je primárně způsobeno použitím lineárních relaxací, které jsou použity při propagaci

[1] Volný překlad error-based interval propagation.

skrze uzly místo reálných aktivačních funkcí. Je tomu tak z toho důvodu, že běžné aktivační funkce jsou nelineární a v případě postupné propagace jejich nelineárního chování by se jen obtížně dalo použít nějaký solver. Vedlejším produktem použití lineárních relaxací je tedy nemalé nadhodnocení, které vede k situacím, kdy LP-solver najde potenciální průnik, o kterém ale nemůže říct, zdali je reálný.

Z toho důvodu se používá ještě další vhodný solver či algoritmus (v tomto případě lokální gradientní hledání), který se snaží najít konkrétní reálný protipříklad, který by narušoval definici lokální robustnosti. Ten nemusí být právě kvůli nadhodnocení nalezen, a proto je někdy nutné dosavadní popsaný proces opakovat na přesnějších přepracovaných reprezentacích.

Dnešní nástroje jako jsou Crown, Active Sets či již zmíněný VeriNet k tomu využívají metodu větví a mezí. Ta spočívá v dělení vybraného vstupního problému na menší podproblémy (v případě VeriNetu dělením vstupů uzlů). Tímto postupem pak vzniká strom, ve kterém je buď nutné prokázat, že všechny podstromy (větve) jsou bezpečné, nebo nalézt aspoň jednu větev, která bezpečná není.

Výběr uzlů, na kterých se dělí vstupy, je jádrem této práce. Jednou z původních strategií byl hierarchický výběr – nejprve se vzal první uzel v první vrstvě, pak druhý, pak třetí atd. To nebylo efektivní z hlediska toho, že v rámci verifikace se za vhodnou dobu nedostalo na všechny uzly a u některých se dělilo zbytečně. VeriNet pak přinesl adaptivní strategii, která vždy, na základě již zmíněných chyb, bere uzel s nejvyšší zpropagovanou chybou. Problémem této metody je, že jednak nerespektuje pořadí dělených uzlů, kdy se obecně vyplácí rozdělit nejdřív uzly v dřívějších vrstvách a pak v pozdějších, jednak má tendence upřednostňovat uzly spíše v pozdějších vrstvách. S ohledem k tomu, že s každou propagovanou vrstvou se chyba zvětšuje, je vhodné dělit i na dřívějších vrstvách.

Jednou z navržených a vyzkoušených strategií je semi-hierarchická strategie výberu uzlů. Ta si bere část své funkcionality z obou metod. Její základní princip je, že se postupně prochází vrstva za vrstvou, přičemž dělený uzel se vybírá zrovna z aktuální postupně vybrané vrstvy. Díky tomu se šetří čas, který by způsobilo opačné dělení uzlů, a současně se dříve eliminují chyby vyplývající z postupného průchodu relaxacmi. Následkem tohoto přístupu také nedochází k přílišnému dělení na jedné konkrétní vrstvě, které by vedlo k pomalému odstraňování chyb z reprezentace.

Tuto a další experimentální metody jsme implementovali do aktuálního nástroje VeriNet, kde jsme prováděli experimenty s modely sítí natrénovaných na standardním trénovacím datasetu MNIST, primárně pak na sítích s ReLU aktivačními funkcemi a okrajově i s aktivačními funkcemi tanh a sigmoid. U prvních zmíněných jsme dosáhli někdy i několikanásobného urychlení již řešitelných problémů a současně jsme dosáhli redukce neřešitelných problémů, které současné nástroje nebyly schopny verifikovat. Vedlejším produktem těchto experimentů pak bylo odhalení implozivních případů, kdy se daří daný vstupní problém vyřešit za více než tisícinásobně kratší dobu.

S ohledem k tomu, že se jedná o velmi aktivní a dynamicky se rozvíjející se disciplínu, v rámci závěru bylo navrženo několik směrů, kterými se lze vydat dál. Mezi ty se primárně řadí implementace této strategie do dalších verifikačních nástrojů, detailnější srovnání s ostatními nástroji či hlubší zkoumání příčin existence implozivních případů.

Formal Analysis of Neural Networks

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of docent Milan Češka. The supplementary information was provided by Mr. Patrick Henriksen from Imperial College London. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

. . . .
David Hudák
June 25, 2022

Acknowledgements

I would like to thank my supervisor, doc. RNDr. Milan Češka, Ph.D., for his guidance and suggestions. I would like to thank Mr. Patrik Henriksen for his support while working with the VeriNet toolkit. And I would like to thank my family and friends for continuous support throughout my studies.

Contents

Chapter 1

Introduction

Over the past decade, deep learning has become part of many applications. We can find its use in many ordinary disciplines, such as healthcare, traffic, aviation, or photography. Generally, the area where we can use deep learning with deep neural networks is unlimited. However, it has one significant hitch [11].

Many areas where we want to use deep learning methods are safety-critical so that the end customer may require system features such as safety or correctness. For example, when we use an autonomous car, we would like to know whether the car's system reacts (artificial driving system) to various situations correctly. For example, the car does not increase the velocity against pedestrians at the crossing [16]. It is unacceptable to sell something that can be potentially dangerous in the real world.

In contrast to some decision trees or expert systems, neural networks are problematic to understand. Today, we do not have good tools for their verification and their use, especially in safety-critical areas, is still at some risk. Risk because neural networks work as a black box. We can design different architectures and use many methods and examples to train them [10, 18]. However, as a result, we do not know how they work. We can deduce it for small networks but not for the networks we commonly use [16].

The main problem of neural networks revealed by Szegedy [20] is small perturbations that cause misclassification. These perturbations can be so small that they are not recognizable to the human eye [1]. Thus, for example, some hackers[1] or some standard deviations of recording devices (some artifacts, noises, brightness) may trigger potentially dangerous misclassifications [20].

Aircraft Collision Avoidance System X (ACAS Xu) is an example of a deep neural network that has found its use in safety-critical area. This system aims to anticipate collision situations and suggest the correct behavior. Suppose this system made the wrong decision based on intentional or unintentional perturbations. It could easily mean billions of dollars in damage and the loss of hundreds of lives [23].

1.1 State-of-the-art approaches

Neural network verification is a dynamic, evolving topic today, and there are many different approaches. These approaches differ in how they prove the correctness of the network. For example, work [18] notes three elementary approaches – reachability, optimisation and search. Reachability constructs potentially reachable classifications at the output layer [25].

[1] For example https://www.tensorflow.org/tutorials/generative/adversarial_fgsm

Optimization seeks to create a set of variables that are subsequently optimized. For example, MIPVerify [21] describes optimization by finding the minimum possible distance to the closest adversarial result (see 2.2). Today's state-of-the-art solutions use the third approach, search methods.

We combine search methods with the previous two approaches. Instead of a complete proof, they focus on finding concrete examples and counter-examples. Thus, in the context of reachability, they do not try to reconstruct some complete reachable area. They would instead find a counter-example outside the permitted area [13]. As part of optimization, they can look for an example beyond the limit determined by the adversarial result [7]. This thesis focuses on one of these state of the art solutions called VeriNet. As a part of this tool, we design and implement extensions that we compare with the initial results.

The main inspiration came from The Second Verification of Neural Networks Competition[2]. The winner solution was α,β-CROWN toolkit[3], boosted by GPU accelerated algorithms. The second was a VeriNet toolkit[4] (with an improvement named DEEPSPLIT) developed at the Verification of Autonomous Systems (VAS) group at Imperial College London. The main advantage of this toolkit is that there was no GPU acceleration, and the algorithm got an excellent placement even with the CPU-only algorithm.

Both mentioned toolkits implement actual state-of-the-art algorithms for local robustness neural network analysis. However, while having excellent results, these toolkits still face difficulties. They still can not solve more complex cases regarding inadequate training, network size, or different activation layers [11, 27, 14].

These toolkits are focused on searching counter-examples in some neighborhoods of inputs of testing images (or other general inputs). The central premise is simple – if the algorithm proves that there are no possible counter-examples in some ϵ neighborhood, the neural network is safe for that case. If they find at least one, the network is not safe for a particular case. Formally, this work deals with the analysis of local robustness [28, 18, 13].

1.2 Contribution

Most search-based verification algorithms consist of three parts (see Figure 1.1): propagation of input constraints to the output layer, some solvers that will try to find some counter-example to a given representation, or prove that no such case exists. The third phase helps the previous phases with the input problem's refinement (branch and bound) if the solvers mark the problem undecidable.

The VeriNet toolkit achieves it by adding split constraints to node inputs across the network to create smaller sub-problems – branches. Problematic is the choice of split node, so this thesis hypothesizes that creating different splitting strategies should accelerate decisions over more complex cases. We proposed and evaluated three new types of strategies extending the original branch and bound functionality with this idea in mind. Moreover, we proved that we could significantly speed up the current state-of-the-art solution with different splitting strategies. That is not only beneficial for the discussed VeriNet toolkit but also for any verification toolkit that uses branch and bound phase.

With our experiments, we show some undocumented behavior of verification toolkits. For example, we encountered an anomaly which we call branch implosions. We also expanded the set of experimented networks from a package of Marabou toolkit. Moreover, the

[2] https://s.google.com/view/vnn2021

[3] https://github.com/huanzhang12/alpha-beta-CROWN

[4] https://github.com/vas-group-imperial/VeriNet

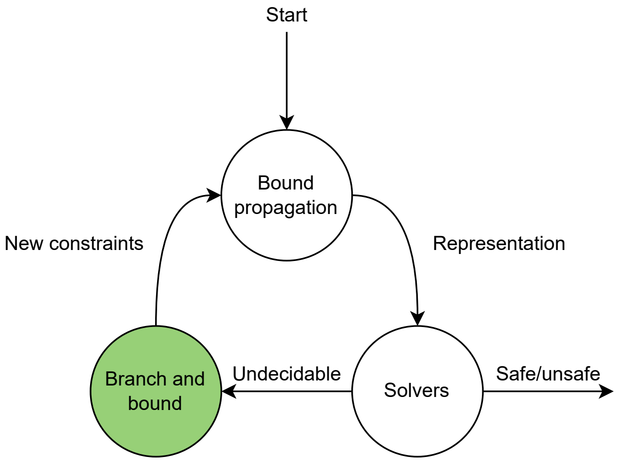


Figure 1.1: Simplified cycle of algorithms with branch and bound phase. We focus on the green one in this thesis. See Section 3.1 for more details.

contribution of this paper is a deeper description of the behavior of the VeriNet verification toolkit.

When reading this work, it is also necessary to keep in mind that the verification of neural networks is a very active and dynamic topic and has taken a considerable step forward in the last six years [29]. Scientific groups worldwide are working even during the writing of this work and are inventing many new algorithms and tools. These take the scalability of verification algorithms to new levels, and the knowledge speculated here can already be surpassed.

Chapter 2

Preliminaries

This chapter describes the main underlying concepts we need for deep learning verification, such as neural network, activation function, local robustness, completeness, or fundamental directions to verification. This chapter also mentions some safety-critical areas where we can use deep learning algorithms.

2.1 Neural networks

Neural networks can be used for various purposes [2]:

- Regression or function approximation – neural network computes from input values output of the simulated nonlinear function.
- Data analysis – neural network sorts some data into some categories by similarities.
- Classification – neural network decides from input values some output classification. The network chooses the classification from the highest value of outputs values.

In this thesis, we focus on the classification of neural networks.

2.1.1 Deep feedforward neural network

Deep feedforward neural networks are feedforward neural networks (**FFNN**) that are composed of a high amount of nodes in many layers.

A feedforward neural network contains one input layer, which accepts all inputs of the given problem, many hidden layers, and one output layer from which we derive the classification. Each layer is composed of nodes (neurons). Each neuron has n inputs and m outputs. Each input is composed of a matrix of learned values (weights), output values from the previous layer (or network inputs), and bias. The neuron multiplies the weights with values from previous layers and then sums it all together with bias. Every neural network node has an activation function: $\sigma : \mathbb{R}^n \to \mathbb{R}^m$, which converts a node input values to matrix output values [11]. Same as in the quoted thesis, we ignore skip-connections in this thesis.

An essential part of neural networks is activation functions, which affect the completeness/soundness of verification and significantly impact the scalability of verified networks. The most important [2] are these:

$$ReLU(x) = max(0, x)$$

$$Sigmoid(x) = \frac{1}{1+e^{-x}}$$

$$Tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$$

VeriNet toolkit also includes support for neural networks with batch layers. These networks have layers that do not use activation function to transform values from inputs to outputs but use linear transformation[1]. These layers have an equal number of nodes as a layer before [11].

Remark (Convolutional networks)**.** *Although many modern neural networks use convolutional networks, this paper does not deal with them. For technical reasons, our version of VeriNet does not support convolutional neural networks. However, it is necessary to mention that the verification of convolutional neural networks (CNN) is not significantly different from the verification of FFNN. The designed methods should be fully applied when added to any toolkit (for example, DEEPSPLIT [14] or Crown [24]) that can verify CNNs.*

2.2 Local robustness

In the context of neural networks, the term robustness comes from the biological specification of living species, where robustness means endurance against external or internal perturbations. Robustness in the context of the neural network is similar in the way that some bigger or smaller deviations from the normal situation cannot affect correct functioning [10]. For example, a robust human can work and survive at different temperatures or not die while eating a bit of toxic food. A robust network works correctly even with a dirty camera.

Local robustness formally extends this definition. For example, the Crown project describes local robustness by a neighborhood of input x_0 and ℓ_p ball around input where all values (in ℓ_p ball) need to have the same classification. If the tool falsifies the definition, the neural network for x_0 is not robust [28]. The VeriNet documentation has a more complex (but also better for formalization) definition – the definition we summarize below [11].

Definition 2.2.1 (Local robustness)**.** *Consider a tuple $\langle f, \psi_x, c \rangle$, where f is an FFNN*[2] *$f : \mathbb{R}^m \to \mathbb{R}^n$, ψ_x is set of constraints $\{l_i \leq x_i \leq u_i | l_i, u_i \in \mathbb{R}\}_{\forall i \in \{1,2,\ldots,n\}}$, where l_i and u_i are defined as $x_i \pm \varepsilon$ and are lower and upper bounds for input values, and classification $c \in \{1, 2, \ldots, m\}$. Let $f(x)_i$ be i^{th} output of FFNN. Neural network is locally robust when for each x' satisfying ψ_x and each $t \neq c$ applies $f(x')_c > f(x')_t$ [13].*

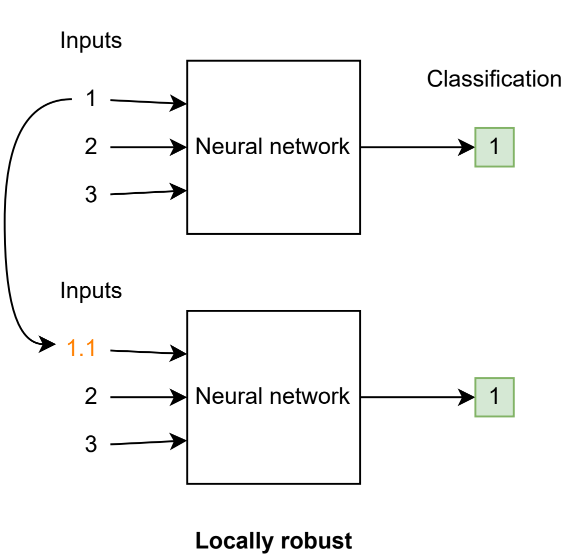
Formal verification of neural networks is based on breaking local robustness condition $f(x')_c > f(x')_t$. The verifier tries to find a counter-example $\overline{x}$ for which applies $f(\overline{x})_c \leq f(\overline{x})_t$. If the verifier does not find any counter-example $\overline{x}$, the classification of input values is called safe. Otherwise, when the verifier finds a counter-example, it calls unsafe because it found a counter-example. In the following chapters, we will find out that algorithms for finding these counter-examples are not necessarily complete these days.

An example of a counter-example situation is in Figure 2.1. We can see that the expected behavior of a robust neural network is that small perturbation can not change classification if the network is locally robust.

[1] $y^i = y^{i-1} + b$, where y^i is an output of layer, i is layer index, and b bias.

[2] Fast-forward neural network.

Remark (Adversarial result)**.** *When working with verification tools, we may also encounter the notion of an adversarial result. Where the counter-example (result) is anything that falsifies some constraints, the adversarial result is some maximum allowable disturbance of inputs to some classification. If the case input constraints exceed the limit given by the adversarial result, then the network changes output classification [18].*



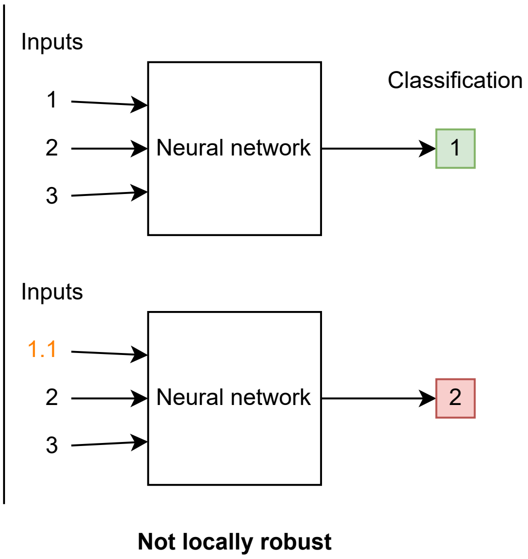


Figure 2.1: You can see a locally robust network on the left side and not locally robust network on the right.

2.3 Completeness

When working with different verification toolkits, we encounter different features. For example, which networks it supports (types of layers, activation functions) it supports, what methods it contains, what it verifies (local robustness), and also whether it is complete (or for which networks). To this is added the concept of soundness, which only says that if the representation cannot calculate a value, then the existing network is not able to calculate it too.

If we say that the tool is complete, then we can say that the tool covers the 1:1 behavior of the neural network. Thus, if the representation of a neural network within a complete tool says that the network can take on some value, then an existing network can indeed take on that value. If it cannot take on some value, then neither a real network can take on that value [18].

Some tools, such as ExactReach [26], create a complete representation on the first iteration. The problem is that it only works for the piecewise linear ReLU activation function and that this tool can practically not verify any more extensive network. Tools complete for networks with sigmoid and tanh activation functions do not appear much. Today's trend, which follows both VeriNet [11] and Crown [28, 27, 24], is an incomplete representation of behavior that they gradually refine to some complete representation (branch and bound). As a result, they do not use unnecessarily accurate representations for uncomplicated cases

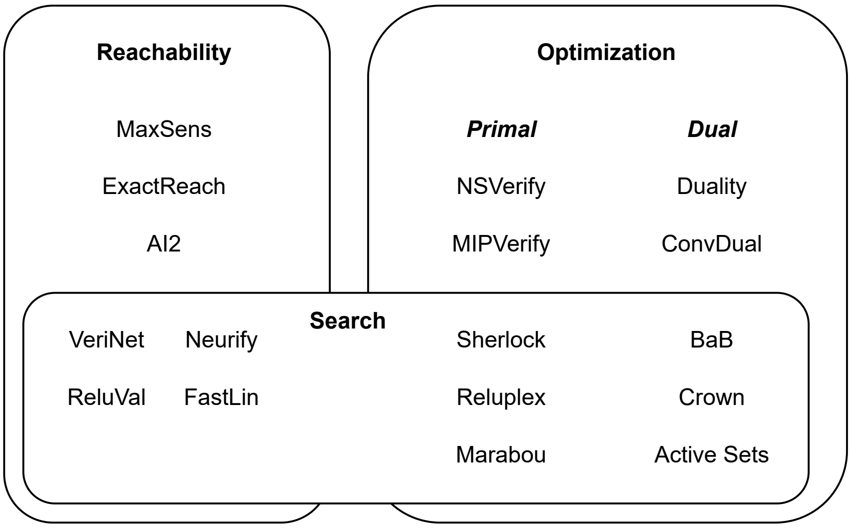


Figure 2.2: Scheme of recent methods. Taken, updated and modified from [18].

and insufficiently accurate ones for problematic cases. This incompleteness leads to much better scalability for more extensive networks.

2.4 Basic verification directions

We can read in the literature [18] that there are three basic directions. They are called verification by reachability, optimization, and by search. For the mentioned methods, we work with a set X of all possible inputs, set Y of allowed outputs, and the function $\mathbf{f}$ as a neural network. See Figure 2.2 for an overview of current solutions.

2.4.1 Reachability

Reachability methods constructs a set of reachable outputs R(X, $\mathbf{f}$). The construction is done by taking the sets of input values and, using various methods, creating representations of NNs, propagating the input values through the neural network up to the output layer. A solver (or another algorithm) decides which classifications are reachable with a given representation at the output layer. For a set of these obtained classifications, it must hold that it is a subset of the possible classifications, i.e.:

$$R(X, \mathbf{f}) \subseteq Y$$

These methods divide between those that do or do not perform over-approximations. ExactReach [26] does not perform over-approximations, so it is complete for its set of problems but can only be used for NNs with ReLU activation functions and cannot be used for medium or more extensive networks. Ai2 and MaxSens [25] are not complete, but the range of verifiable networks is larger [18].

2.4.2 Optimization

Primal optimization methods stand on trying to falsify assertion:

$$x \in X \implies y = f(x) \in Y$$

In this case, the neural network is a constraint structure that, by various algorithms, is optimized. For example, ILP [5] toolkit iteratively tries to estimate the maximum input set X_E of neural network concerning the correct outputs Y. Suppose $X \subseteq X_E$, the neural network is safe for that case. Current solutions are limited to using only ReLU activation functions [18].

The second approach to optimization methods is dual optimization methods. These methods work on the principle that they try to optimize output constraints that, if they violate each other, the network is not safe for a given case. Compared to primary optimization, these bounds are much easier, and thanks to, for example, Lagrangian relaxation (Duality [9]), it is possible to implement other activation functions than ReLU.

However, using relaxations is necessary to use specific propagation methods, leading to incompleteness. In addition, the constraints created in this way are different than in the case of primal optimization [18]. It is thus necessary to create new solvers that can solve dual problems. One of them is the Active Sets [19] solver, which works in combination with search methods.

2.4.3 Search

Where standard reachability and optimization methods try to create an area and prove what values a given network can acquire, search methods find concrete examples (or counter-examples) that would falsify some condition. So, where the reachability methods must prove the entire reachable output set, search methods only need to prove that there is no counter-example to claim otherwise. Alternatively, find a specific counter-example instead of proving an area that violates local robustness.

One of the implementations of these methods (combination of search and reachability) is the VeriNet [13, 14, 11] toolkit, which is the main object of this thesis. The following Chapter 3 primarily describes the elementary concepts that build these verification algorithms. It is also worth adding that current competition algorithms such as Crown [28, 27, 24] or Active Sets [19] consist of searching with dual optimization. So if we talk about some state-of-the-art algorithm, it is practically certain that it will be a tool based on some form of search.

2.5 Disciplines using neural networks

As mentioned in the Chapter 1, one of the essential milestones of neural networks was the use of the ACAS Xu [23] collision detection model. Nevertheless, that is not all. The use of neural networks has become a regular part of routine life essentially and their use today reaches areas that until recently could only be done by a human.

2.5.1 Autonomous driving

One of the current hot topics today is the transportation industry's future. In addition to the tendency to leave internal combustion engines, it is a matter of autonomous driving. Various algorithms for object detection, traffic sign identification, route control (GPS), or driving that autonomous car use stands on neural networks [4].

In addition, we encounter the need to correctly aggregate information from various sources and adapt the NNs to them. For example, to perceive the surroundings, such a vehicle draws data from three sources - the camera, LiDAR (laser beams for perceiving depth), and radar. Examples of known systems using neural networks that autonomous vehicles use include HydraNet, ChauffeurNet, or NVIDIA self-driving car [4].

On the one hand, there is a lower need for continuous learning and, therefore, less need for fast verification. On the other hand, the required size of the network increases with a generally more extensive number of inputs and the need to respond to virtually everything in a large area around the car. For example, one of the works [15] on neural networks for autonomous driving systems for embedded devices designed a network with just over 150,000 trainable parameters. We assume that these numbers would be much higher in reality because this system focuses only on using data from the camera.

The need for verification is evident in the case of autonomous management - traffic safety. Every traffic accident potentially brings economic losses in the case of car damage and potential transport of goods. And the loss of lives in worse cases.

A little ethical reflection

Ethical and moral responsibility is also problematic in this case. If an accident occurs, who is responsible? The author of the system, the one who sold the vehicle, the one who verified the system, the one who bought it? Therefore, in addition to the fact that this and other works try to deal with a formal technical approach to the given problem of neural networks, it is necessary to consider legal and ethical risks. Another problem may be that maintaining such a system requires obtaining a lot of data, which can mean a significant invasion of privacy.

2.5.2 Deep learning in healthcare

The previous example replaces the ordinary person, the driver, and the following replaces original security such as passwords or fingerprints. This example goes further. It replaces years of education and experience with the machine. It is a medical use that we classify into two main directions - diagnosis [2] and treatment [8].

In the basic sense of the word, diagnostics work on the principle of a database of patients with a given set of symptoms, test results, personal characteristics, and their final diagnosis. Then neural networks use such sets to train a network that serves to diagnose a specific patient with an unknown diagnosis. The resulting network can then detect, for example, cardiovascular disease, cancer, or diabetes [2].

Figure 2.3 shows a fundamental consideration of how such a model could work for some common diseases A and B. For example, today, this detection tool could detect and distinguish between Covid-19 and influenza, which have very similar symptoms. Symptom A could be cough, symptom B olfactory loss, and laboratory test A standard antigen test that is not very accurate and is not sufficient for diagnosis.

In the case of cancer, relatively harsh treatment is used, including radiation, chemotherapy, or surgical removal of the tumor. From this point of view, it is essential that the diagnostic tool used is well trained and verified, as a wrong diagnosis can lead to fatal consequences for the patient.

The problem with the neural network in this case, and even more so in the case of using the neural network for surgical interventions, is the already mentioned fact that neural networks function as a black box [8]. The black box means that the neural network takes

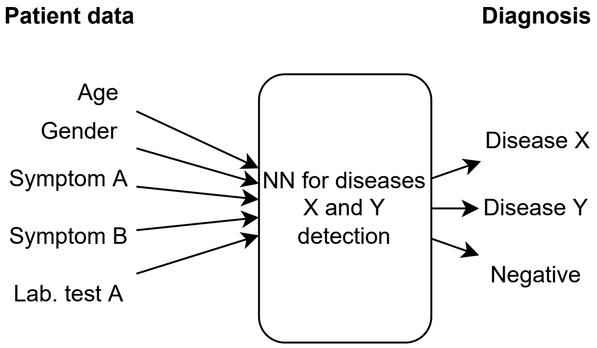


Figure 2.3: An example of using a classification network to detect X and Y diseases related on age, gender, symptoms A and B and with the possibility of laboratory test A. Inspired by [2].

the inputs and returns the output—nothing in between. Unlike a doctor who has years of experience and education and can say why and how he came to a given conclusion and wrote a medical report, we do not know anything about the calculation method in a neural network. Therefore, it is necessary to significantly improve the tools for the formal analysis of neural networks.

A little ethical reflection

Compared to the other two examples, the author of the verification system gets into much greater responsibility than in the previous two examples. In the case of autonomous driving, the user can intervene in the vehicle's driving in the event of a visible failure. While illness diagnosis or surgery, there may be only the machine trained and verified by their creators. Thus, the network creators and those who verify the network have the destinies of human lives in their hands.

On the other hand, the availability of qualified medical care in many less developed societies is poor. Such generically learned machines can provide at least the necessary care.

2.5.3 Face recognition

One of the most common areas where neural networks occur is image processing. After all, it is already quite common today for mobile phone publishers to write "boosted by AI" on their packaging. However, there is a slightly safety-critical discipline apart from recognizing food, text, nature, or other objects. This discipline is the face recognition with which we can unlock the various devices.

The main benefit of using neural networks is training the network directly to a specific person in real-time, moreover, with personal appearance changes [17]. For example, Apple boasts of using "Neural Engine" chips to accelerate the learning of NNs and thus offer comfort to their customers in real-time[3].

In this example, it can be relatively quickly emphasized that the excellent scalability of verification algorithms for neural networks could be pleasant. The user would not have to wait long, and he would know that nobody can exploit bugs in the face recognition system. Moreover, many people store essential data on their devices, such as bank logins, passwords,

[3] https://support.apple.com/en-us/HT208108

or internal job data. Thus, the security weakness of neural networks can lead to unpleasant consequences. An enormous problem can be when an unverified, poorly trained network is in a device owned by some high general or politician.

A little ethical reflection

The ethical question arises in the context of recognizing a face. It is necessary first to obtain the data, more accurately an image or, better, a 3D scan of the face. On the one hand, for example, phone manufacturers can be trusted that phone makers leave biometric data only on the device and do not send it anywhere or misuse them for unfair matters. If someone does not trust this system, the camera can be pasted or turned off within the SW. Alternatively, the person does not have to buy it.

In the context of crime, cameras often multiply on the streets, and it is not difficult to create a complete population map when using face recognition technology. That is the main area where facial recognition technology can be misused. This tool is even being abused today by the People's Republic of China in its Black Mirror-inspired social credit system, where people gain and lose credit not only for common offenses and crimes but also for disloyalty to the government. In this case, the author of the system that verifies this system becomes an accomplice of the regime.

In the context of crime, cameras often occur on the streets, and it is not difficult to create a complete population map when using face recognition technology. That is the main area where facial recognition technology can be misused. This tool is even being abused today by the People's Republic of China in its Black Mirror-inspired social credit system[4], where people gain and lose credit for common offenses and crimes and disloyalty to the government. In this case, the author of the system that verifies this system becomes an accomplice of the regime.

[4] https://en.wikipedia.org/wiki/Social_Credit_System

Chapter 3

Main concepts

This chapter deals with the basic building blocks of a large part of tools for neural network verification. We explain the primary verification cycle, which occurs in VeriNet, and its three basic blocks. We also briefly mention the competitive Crown tool and tools created by Bunel et al.

3.1 Verification cycle

The toolkits like VeriNet [11], Crown [28], BaB [7], and others consist of three main blocks (see Figure 3.1) – construction of a network representation (usually some bound propagation method), processing by a solver, and possible refining of the solved problem. The algorithm always starts by creating a representation and terminates with an obtained result. An exception is a timeout, which occurs primarily for more extensive networks and is a relatively common way to terminate an algorithm [11].

The third option of termination of the verification cycle, linked to the system architecture and can be handled in various ways, is termination due to lack of memory or excessive rounding error. Rounding errors can terminate the algorithm when it is impossible to create bounds within the representation or rework phase. It occurs when we break the elementary condition *lower_bound* < *upper_bound*.

3.2 Linear relaxation

Before mentioning bound propagation methods, it is essential to realize that the activation functions used in standard neural networks are non-linear. This non-linearity, as we explain in Section 3.3.2, is not suitable for some bound propagation methods, such as symbolic interval propagation, and therefore so-called linear relaxations are used.

Linear relaxation is a process where non-linear functions (as $ReLU$, *sigmoid* or *tanh*) are converted to linear over-approximations. The result consists of n-tuples of linear constraints for node input and output. Valid outputs that respect these constraints are part of the overestimation of the output space. For example, to define the linear relaxation of the most typical activation function of neural networks, $ReLU$, Henriksen et al. [11] refer to derived relaxation:

$$ReLU(x, y) = \{y \geq 0, y \geq x, y \leq \frac{x_u(x - x_l)}{x_u - x_l}\}$$

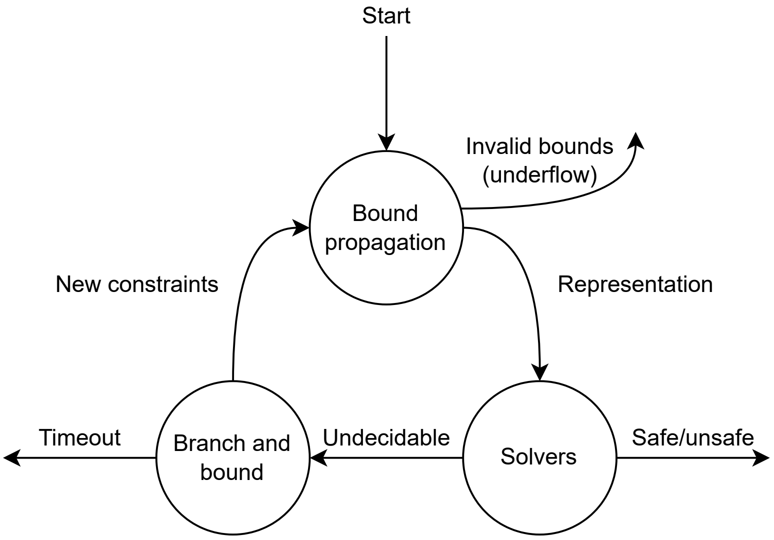


Figure 3.1: The main cycle of VeriNet and others.

where x_l and x_u are lower and upper bounds of the input. Basically, we intuitively create a triangle, where two sides are based on the *ReLU* function and the third side is the connection between the two farthest points.

However, this relaxation is not entirely suitable, as it contains three linear constraints, which leads to a more precise output space, but at the same time, leads to a more demanding calculation for further verifications. Therefore, we subsequently encounter two characteristic bounding lines during the verification: upper linear relaxation and lower linear relaxation. So if we use the same example of the *ReLU* activation function, we get a set of linear constraints:

$$ReLU(x, y) = \{y \geq \frac{x_u x}{x_u - x_l}, y \leq \frac{x_u(x - x_l)}{x_u - x_l}\}$$

where x_l and x_u are lower and upper bounds of the input. To illustrate, we can see in Figure 3.2 that such relaxations are actually two parallel lines glued to the *ReLU* activation function [11].

The elementary principle is intuitive, but there are many problems with it. The first one is the calculation of the already mentioned bounds x_l and x_u. Within VeriNet, their calculation solves propagation methods (SIP, ESIP), which propagate the input bounds through the network.

The second one is that propagation methods use these linear relaxations, but, as can be noticed, there is a significant overestimation. This overestimation means that the errors of the following bound increase with each new layer with the relaxed activation function. That, in the end, causes false counter-examples [11]. The accuracy of these estimates is one of the issues discussed, and different research groups take different approaches (for example, some other toolkits use Lagrangian decomposition for refinement [6]).

Due to overestimations, the toolkits with linear relaxations are no longer complete by default. This incompleteness we can solve while using the *ReLU* activation function, which is piecewise linear. However, for other nonlinear functions, such as the already mentioned tanh and sigmoid, this generally leads to a sound but incomplete solution [11, 18].

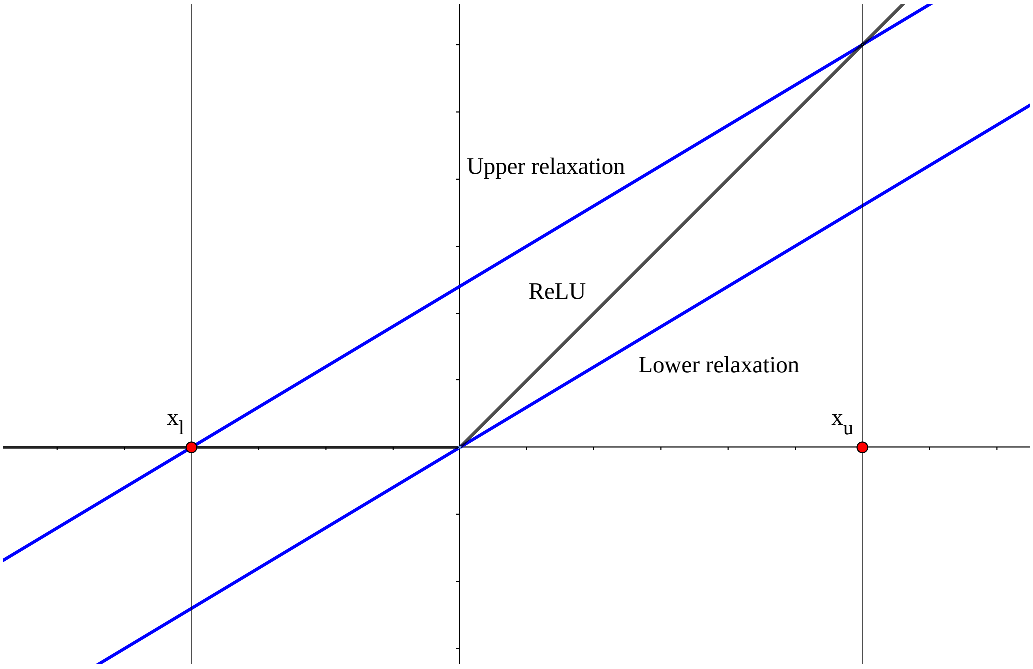


Figure 3.2: An example of relaxation of ReLU activation function.

In contrast to *ReLU*, there is also the problem that the creation of linear relaxation may not necessarily be feasible with the help of an analytical solution. Henriksen et al. [11] represents an iterative approach for *sigmoid* and *tanh* activation function, where it is possible to obtain relatively precise relaxations.

In general, linear relaxations are a partial departure from complete solutions, but thanks to the introduction of linearity into the verification problem, the scalability of the verification tool increases. Some other solutions, such as the Crown [24], then increase accuracy at the expense of speed by introducing quadratic relaxations. Regardless, we do not address them in this work.

3.3 Representation methods

An elementary problem of neural network verification is converting node notation with edges to something that a verification tool can solve. Approaches to this are different, but today we primarily talk about the so-called bound propagation methods. The bound propagation methods try to make some estimation of $f(\boldsymbol{x})$ from the model notation, where $f(\boldsymbol{x})$ represents the behavior of the neural network and $\boldsymbol{x}$ the vector of the inputs. Depending on the activation methods and layers used, these methods are then differently applicable. The result of bound propagation methods is some boundaries, constraints, equations (linear, mixed-integer, some intervals), and others in verifying usable data [18, 11].

Remark (Original approaches)**.** *The original methods for verification used MIP coding of ReLU nodes, which was complete but at the same time very computationally demanding. The next step was their relaxation of LP constraints, which led to incompleteness, but at the same time, the field of verifiable neural networks expanded [18].*

In this work, we focus primarily on symbolic representations because the toolkit (VeriNet) used in this work uses them. More details on the given methods are in the main sources [13, 11, 23].

3.3.1 Naive interval propagation

In order to find counter-examples, we need to identify possible outputs based on the intervals for each input. For a naive implementation, Henriksen et al. [13] gives the following formulas (for FFNN):

$$z_l^i = W^{i+}\boldsymbol{y}_l^{i-1} + W^{i-}\boldsymbol{y}_u^{i-1} + \boldsymbol{b}^i$$
$$z_u^i = W^{i+}\boldsymbol{y}_u^{i-1} + W^{i-}\boldsymbol{y}_l^{i-1} + \boldsymbol{b}^i$$

where z_l^i and z_u^i describe the lower and upper bounds of layer i, y_u^{i-1} is the output upper bound of the previous layer, y_l^{i-1} is the output lower bound of the previous layer, b^i is the bias of layer i. We generate W^{i+} from the weight matrix as:

$$W_{k,h}^{i+} = \begin{cases} W_{k,h}^i & W_{k,h}^i > 0 \\ 0 & \text{else} \end{cases}$$

The real output of each layer is obtained with activation function $\sigma(x)$ of layer i as:

$$y_l^i = \sigma^i(\boldsymbol{z}_l^i)$$

$$y_u^i = \sigma^i(\boldsymbol{z}_u^i)$$

These formulas say that each node in each layer calculates the minimum and maximum possible values z_l^i and z_u^i, which stand on a matrix of the minimum and maximum values y_l^i and y_u^i. These we obtain from the previous layer outputs multiplied by a matrix of weights of edges between the current and previous layer in FFNN.

The advantage of naive interval propagation is its simplicity. However, the disadvantage is that it does not include conditional dependencies of minimum and maximum values between nodes across the network and thus significantly overestimates the output intervals' ranges. Thus, this method causes many undecided case occurrences, making it necessary to split the solved domain into more subdomains [13].

3.3.2 Symbolic interval propagation SIP

A big step away from this naive implementation introduced the ReluVal [23] and Neurify [22] toolkits – a symbolic interval propagation (SIP) [18]. Instead of propagating specific values, SIP propagates linear equations. According to experimental results, this method leads to a reduction of overestimation compared to naive implementation. In contrast to naive implementation, the formulas for each layer change to:

$$eq_{low,in}^i(\boldsymbol{x}) = W^{i+}eq_{low,out}^{i-1}(\boldsymbol{x}) + W^{i-}eq_{up,out}^{i-1}(\boldsymbol{x}) + \boldsymbol{b}$$

$$eq_{up,in}^i(\boldsymbol{x}) = W^{i-}eq_{low,out}^{i-1}(\boldsymbol{x}) + W^{i+}eq_{up,out}^{i-1}(\boldsymbol{x}) + \boldsymbol{b}$$

where $eq_{up,in}^i$ and $eq_{low,in}^i$ are symbolic upper and lower bounds and $eq_{up,out}^{i-1}$ and $eq_{low,out}^{i-1}$ are symbolic output bounds of the previous layer. In contrast to the propagation formula of naive interval propagation, the lower and upper input and output values of y^{i-1} and z^i

are replaced by the equations eq^{i-1} and eq^{i}. From working with specific values, we get to working with symbolic notation

This method leads to strongly non-linear equations because we usually use non-linear activation functions in commonly used neural networks. The solution is the usage of linear relaxations, which cancel out non-linearities while creating linear overestimation [18, 11, 23]. Therefore, we take the calculated input upper and lower equations from the previous formula and substitute them with the calculated relaxation. Formally:

$$eq^{i}_{low,out}(\boldsymbol{x})_k = r^{i}_{l,k}(eq^{i}_{low,in}(\boldsymbol{x})_k)$$

$$eq^{i}_{up,out}(\boldsymbol{x})_k = r^{i}_{u,k}(eq^{i}_{up,in}(\boldsymbol{x})_k)$$

where $r^{i}_{l,k}$ and $r^{i}_{u,k}$ are lower and upper relaxations for node k in layer i.

To create specific relaxations, it is necessary to calculate both specific bounds for upper and lower relaxation. These we calculate as:

$$z_l = min(eq(\boldsymbol{x})) = \sum_{i|a_i>0} a_i x^l_i + \sum_{i|a_i<0} a_i x^u_i$$

$$z_u = max(eq(\boldsymbol{x})) = \sum_{i|a_i>0} a_i x^u_i + \sum_{i|a_i<0} a_i x^l_i$$

where $eq(\boldsymbol{x}) = \sum_i a_i x_i$ and where each x_i is bounded by $x^l_i \leq x_i \leq x^u_i$.

In most cases, SIP creates a better representation (tighter bounds) of larger neural networks than the naive method, but it still has its shortcomings. This method is still not able to trace all the inter-dependencies between the individual nodes and still creates a considerable number of false counter-examples. For this reason, symbolic propagation is followed by error-based symbolic propagation (ESIP) and also reverse symbolic propagation (RSIP) within the DEEPSPLIT extension [11, 14].

3.3.3 Error-based interval propagation ESIP

While standard symbolic interval propagation works with lower and higher bound relaxations, the error-based interval propagation (ESIP) method presented here works only with the lower one. It replaces the upper relaxation and the higher bound equation with a specific error value describing the upper relaxation distance from the lower one. Thus, the ESIP further propagates errors and bound equations only through the lower relaxations.

Principle

Each layer thus contains at its input its matrix $E^{i}_{in} \in \mathbb{R}^{m_i \times m'_i}$ representing errors from the previous layers, where m_i is the number of nodes in layer i and m'_i is a number of all nodes in previous layers. We use this matrix for verification via solvers on the output layer and for the splitting strategy heuristic.

Continuous errors and equations of behavior are propagated in an intuitive way using weights for individual edges W^i and for the propagation of equations also with bias $\boldsymbol{b}$:

$$E^{i}_{in} = W^{i} E^{i}_{out}$$

$$eq^{i}_{in}(x) = W^{i} eq^{i}_{out}(x) + \boldsymbol{b}$$

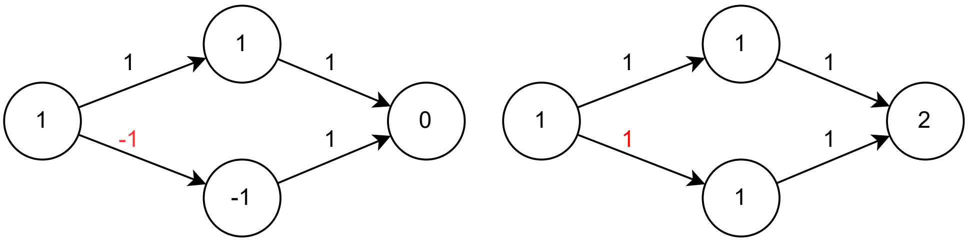


Figure 3.3: Two NNs with one different edge. The nodes contain the propagated error value from the left node. Activation function is $f(x) = x$.

Henriksen et al. [11] provides further necessary formulas for describing the ESIP method. First, how to calculate a new error on a node k in a layer i that describes the maximum possible distance between the value created by the upper relaxation and the lower relaxation:

$$\epsilon_k^i = max_{z^i \in [z_{l,k}^i, z_{u,k}^i]}(r_{u,k}^i(z) - r_{l,k}^i(z))$$

Second, the method of calculating new errors through the node with the usage of only lower relaxations $r_{l,k}^i$:

$$(\hat{E}_{out}^i)_{k,:} = r_{l,k}^i((E_{in}^{i+1})_{k,:})$$

where the resulting output error matrix is concatenation of the propagated errors and new errors:

$$E_{out}^i = [(\hat{E}_{out}^i), diag(\boldsymbol{\epsilon})^i]$$

And very similarly, propagated equations are obtained as:

$$eq_{out}^i(x)_k = r_{l,k}^i(eq_{in}(x)_k)$$

And third, we need to get new lower $z_{l,k}^i$ and upper $z_{u,k}^i$ bounds for each node. To do this, we use the input equations eq_{in}^i obtained from the previous layers and also the error matrix E_{in}^i obtained from the previous layers as:

$$z_{l,k}^i = min(eq_{in}^i(x)_k) + \sum_{h|(E_{in}^i)_{k,h}<0} (E_{in}^i)_{k,h}$$

$$z_{u,k}^i = max(eq_{in}^i(x)_k) + \sum_{h|(E_{in}^i)_{k,h}>0} (E_{in}^i)_{k,h}$$

Benefits

Thanks to all these errors, we can tell what maximum or minimum values are and obtain each node's output using only one relaxation. Another advantage is that thanks to these errors, which we propagate through the neural network, we can cancel each other out or multiply the dependencies between the given nodes. For example, in Figure 3.3, we can see that in the network on the left error, the error cancels out, while in the network on the right, it doubles.

Thanks to these sums and subtractions of errors from previous layers, ESIP manages to suppress the conditional dependency issue. In turn, the absence of the propagation of upper relaxation increases the overall overestimation of the network's behavior. However, experimental results show that ESIP generally leads to significantly better boundaries, and thus, the ESIP generates fewer counter-examples than SIP [13, 11].

Moreover, the concrete error values propagated to the output layer and summed together give a pretty accurate overview of what overestimation is produced by which node. These errors then tell which nodes are pulling overestimation of the correct class down. Moreover, which nodes are pulling up overestimation of potentially counter-example classes. Based on this knowledge, the VeriNet can create a very effective heuristic within the branch and bound phase.

3.4 Solvers

In the previous section, we created a representation and got some output behavior at the output layer. The representation at the output layer can tell us a specific output changes when a particular input changes or if it can reach a specific value. Nevertheless, we do this to check whether the network is locally robust or not, and that is why we use different solvers to find out.

The result of propagation methods is a set of constraints, linear equations, bounds, and other helpful verification data. With that, it is possible to search for counter-examples or other network behavior, and, for example, VeriNet uses the LP solver and local gradient search for its processing. The selection of these tools is also an essential topic within verification tools.. For more details, see [18].

The choice of solver depends on how we code the neural network verification problem. For example, in dual optimization, one of the works [19] designs and implements the so-called Active Set Solver. Some other methods encode the network for other solvers such as MILP, SMT, SAT, or other custom solvers. Different solvers then have different domains of what they can process and how fast they can process it [11]. In general, the less complex the problem to be solved and the less complex the task is, such as having only a linear behavior, the faster the solver usage is. For example, a system of linear inequalities is much easier to solve than a system of quadratic inequalities. However, a less complex solution leads to a less accurate solution, and therefore the solver needs to be called multiple times, so it is necessary to look for a balance. So, for example, using a simple LP solver can be very fast, but again it does not have to process as accurate representations as MILP. We can demonstrate this in the example of the previously mentioned representation methods.

Naive interval propagation would be simple for solvers. It consists only of specific boundary values. The solver used needs only to check the intersections of the intervals formed by bounds. On the other hand, in contrast to the more complex ESIP, this method completely lacks any interdependence of nodes within the network, so the result returned by the solver has a great chance of being a false positive.

3.5 Refinement

As mentioned in Section 3.3, today's state-of-the-art representation methods lead to the overestimation of output intervals. Thus, there are a lot of false counter-examples that do not exist in a given input domain. We can prove the existence of real counter-examples with the help of various solvers, as we mention in the following chapters. On the other hand, proving the non-existence of such cases is a lot more complicated, and most state-of-the-art algorithms go through some refinement phase. It is also important to mention that this phase often adds completeness to the basic incomplete algorithms (at least for ReLU sites) [18, 11].

3.5.1 Existing splitting strategies

These refinement methods usually work by dividing the main problem into smaller ones. Usually, we select a node and then divide its input in half (or in a different ratio). Then the algorithm solves sub-problems and adds a new split constraint to a solver. The selection of these nodes is an essential aspect of verification algorithms, and different approaches lead to differently efficient algorithms.

While the original tools did not rework and did not have to use any strategies, newer ones, such as Neurify [22], moved towards a hierarchical strategy. The strategy is to gradually take nodes from the input layer to the output layer while dividing the inputs of the selected node. This strategy leads to the expansion of solvable problems. However, we cannot split all the inputs of neural network nodes in a reasonable time [11].

In this regard, VeriNet has introduced an adaptive splitting strategy with a heuristic that uses the results from the ESIP phase and deduces the nodes that will be divided first and later according to the effect on the output layer. As a result, VeriNet can find and split necessary inputs sooner and reduce the overall number of required splits [13]. The problem with this adaptive method is that it does not look at the order of split nodes at all and is thus stateless. This stateless logic can lead to a “wrong” split node order and thus slower verification. Moreover, this strategy tends to prefer some layers. Thus it does not split fairly in the whole network.

We describe splitting strategies more in the following chapters, and the new we present and experimentally evaluate new ones in this thesis.

3.5.2 Heuristics

Modern verification tools use adaptive splitting strategies in different ways. Usually, they need some heuristics to determine which node is in the given representation the most significant. Sometimes these heuristics choose the node with the longest distance between bounds or the node with the highest gradients of representation. Alternatively, they can calculate heuristics on some representation characteristics. The third case is also the case of VeriNet.

The VeriNet uses mentioned errors on individual nodes to observe their two undesirable impacts - lowering the lower bound of the correct classification `c` or increasing the upper bound of the wrong classification `t`. The goal of the refinement is to reduce both of these influences (reduce overestimation), and the VeriNet heuristic tries to find such nodes by looking for the most significant errors. Thus, the VeriNet toolkit introduces the value of impact score, which it calculates as:

$$s(h) = \gamma_c max(E^m_{c,h}, 0) - \sum_{t \neq c} \gamma_t min(E^m_{t,h}, 0)$$

where γ is so-called weighting factor and E is the error matrix at output layer from ESIP phase[1] – E_c as the error matrix of correct output and E_t as the error matrix of other outputs.

[1] For details see subsection 3.3.3.

For safe classes, which the VeriNet proved in its previous iterations, we use $\gamma = 0$, for the correct class $\gamma = n$ where n is the number of potential counter-examples, $\gamma = 1$ for other unproven classes. This heuristic does not include any indirect impacts – the difficulty of splitting between the input layer or first layers is more challenging to compute than splitting within later layers. We need to recalculate all subsequent layers in the newly created branches (previous layers remain the same). On the other hand, splitting within the first layers has a more significant impact on decreasing over-estimations caused by linear relaxations of non-linear functions [13, 11]. We discuss these effects more in Chapter 5.

These heuristics are one of the goals for further improvements. For example, in the extension of VeriNet DEEPSPLIT, they create an impact score as a combination of direct, indirect, and propagation effects [11, 14]. There will be a discussion on the importance of the quality of heuristic functions in the following chapters.

3.5.3 Splitting

We discussed some strategies and heuristics to create new branches by dividing the current branch and thus making more accurate representations. We do this branching by adding split constraints to the current representation of the node inputs (and LP-solver). The split constraint then splits the node input in the case of the ESIP method (in VeriNet toolkit) into two "halves" as:

$$eq_i(\boldsymbol{x}) + \sum_{k|E_{i,k}>0} E_{i,k} \geq s$$

$$eq_i(\boldsymbol{x}) + \sum_{k|E_{i,k}<0} E_{i,k} \leq s$$

These split constraints s then differ depending on the activation function of the node. For *ReLU* nodes, split constraints are $s = 0$, because this constraint leads to the cancellation of the non-linearity of the *ReLU* activation function. This cancellation also means that the algorithm is complete for networks consisting only of *ReLU* layers. If we created 2^N branches where N is the number of nodes, all nodes would have exact linear behavior, and thus the representation would have complete behavior [11].

For other activation functions (s-shaped *tanh* and *sigmoid*) σ are split constraints s counted as midpoint:

$$\sigma(s) = \frac{\sigma(z_l) + \sigma(z_u)}{2}$$

where z_l and z_u are lower and upper symbolic bounds to input calculated by ESIP [11].

For an illustration of splitting, suppose we have an overestimated result if we assume that we have some computed input bounds over positive and negative values and a node applies some linear relaxation of the ReLU function on them. However, if we introduce a split constraint $s = 0$, we get two intervals, which are located purely on two linear sections of the ReLU function, so no overestimation occurs. We can see such a situation in Figure 3.4.

3.5.4 Branch and bound

Combining the splitting strategy, heuristics, the splitting method, and the solver's outputs, as described in the sections and subsections above, is created one of the basic building blocks of modern verification algorithms, the so-called branch and bound algorithm.

This algorithm, or rather the programming paradigm, was not initially intended to verify neural networks but, for example, for solving various optimization problems such as

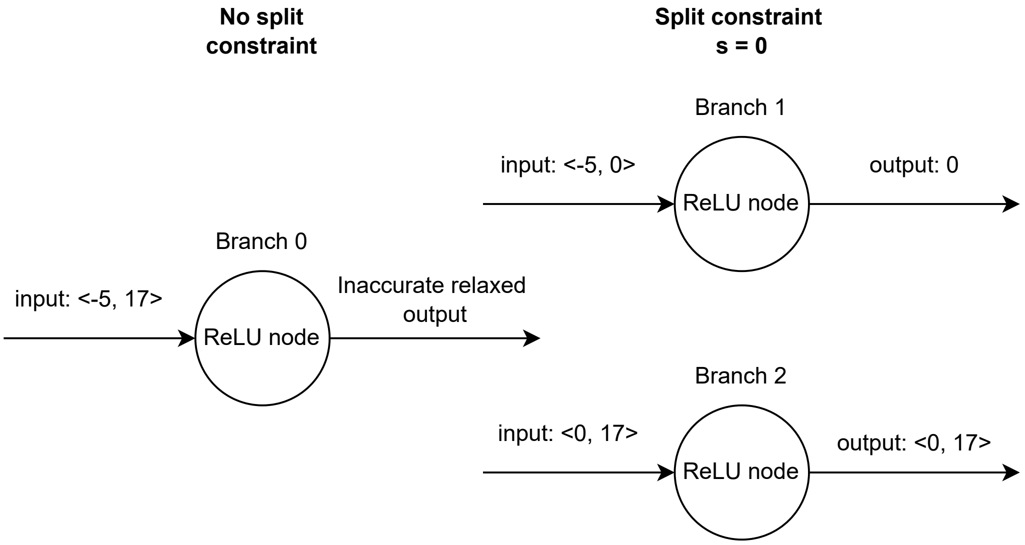


Figure 3.4: Simplified example of adding split constraint. Branch one restricts inputs only to less than or equal to the split constraint and the second one vice versa.

the traveling salesman problem or knapsack problem[2] [7]. One of the original tools that adopted this method for neural network verification is the BaB verification toolkit[3] [7] (named after Branch and Bound).

The branch and bound method constructs a tree divided into sub-domains based on some input domain. This tree aims to gradually reduce the size of the problem which we process and thus solve the smaller sub-problems. The given problem does not have an existing (safe, robust) solution, and we must prove it in all branches, or the problem has a solution (unsafe), and thus, it must find at least one branch which violates some condition. Safe branches we prune. The satisfiability of the branch and the goal of the search depend on the tool used [18, 11, 7].

In the case of VeriNet, we are talking about finding a real counter-example or eliminating all potential. Therefore, if the LP solver, in the case of VeriNet, states that there is no intersection on the output layer and therefore no potential counter-example, then the branch is pruned. If we prune all branches, the network is safe for the case (unsatisfiable). If we find a real counter-example in any branch, the algorithm stops in all remaining branches. The case then we report as unsafe (satisfiable) [13, 11].

Figure 3.5 shows how the branch and bound phase of the VeriNet verification algorithm works. The node symbolizes the problem branch and always shows the number of current potential counter-examples. The branching symbolizes the addition of split constraints on the input domains of the nodes.

In the left tree in Figure 3.5 we can see that we pruned all branches of all counter-examples. I.e., the processed domain and its overestimation do not generate any intersections which the LP-solver could find. In the right tree, we can see that one of the branches found a real counter-example during the branching and the whole algorithm then stopped.

[2] For more examples, see https://en.wikipedia.org/wiki/Branch_and_bound

[3] [18] describes that the Planet and Reluplex tools can also be seen as an implementation of branch and bound.

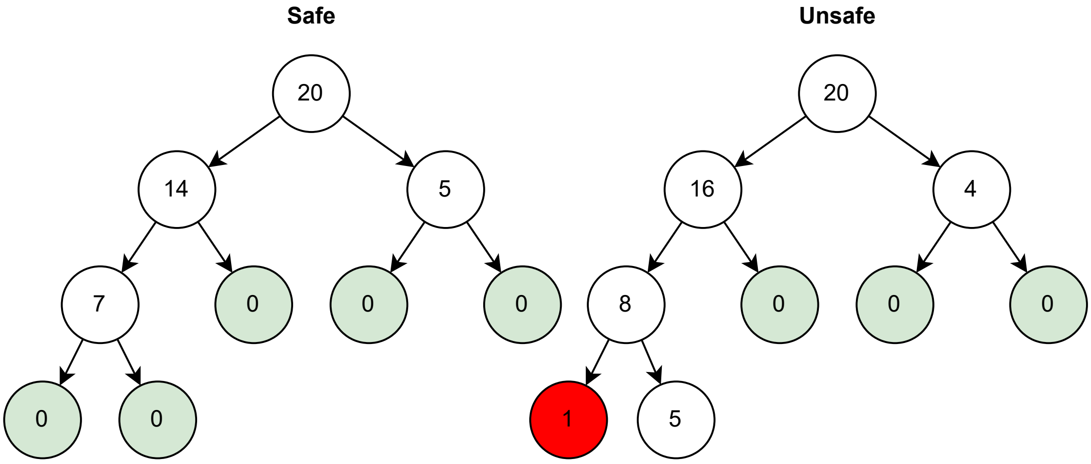


Figure 3.5: Example of a tree for the branch and bound phase of a VeriNet toolkit. The number in the nodes describes the number of potential counter-examples. Green nodes are proven to be safe. Red nodes are proven to be unsafe.

In the case of the BaB tool, we sought a global minimum. In this case, the tool has the condition that in the case of proving a negative value of local minima, the branch, tree, and we mark the whole case as unsafe. If it becomes positive, the case is safe, and we prune the branch. The goal is to prune all branches and find a global minimum higher than 0 or to prove that the global minimum is lower than 0 [7].

Different algorithms use different methods for working with the branch and bound algorithm. However, we can distinguish three elementary building blocks - selection strategy, domain splitting method, and some decision algorithm (solver), which can decide in a set $\{satisfied, unsatisfied, undecided\}$.

Many SoA algorithms use branch and bound, but the sub-blocks often differ significantly [24, 19, 14]. These differents are also because branch and bound often lead to extreme branching when it is necessary to create and verify tens of thousands of branches to solve a given case. After a specific timeout, this situation leads to the network being unverifiable in many cases. This exponential branching is one of the most significant current scalability problems for neural network verification.

It is also necessary to keep in mind that this method of exponential branching has considerable memory requirements. In the case of VeriNet, these requirements are at least partially reduced by the non-recursive implementation of queue-based branching [11, 18]. Another problem may be the so-called underflow result. It says that the division of branches failed to create valid bounds and that we have to terminate the algorithm.

Various efforts to improve this method at the level of splitting strategies we discuss in this work, but a key topic in working with this algorithm is its parallelization. For example, the Crown [28, 27] toolkit in its newer versions approaches massive GPU parallelization. However, the VeriNet toolkit uses only CPU parallelization, but VeriNet$_{BF}$ extension [12] uses GPU.

3.6 Other solutions

The techniques described above primarily focus on VeriNet and similar tools. The following subsections describe a few other exciting solutions.

3.6.1 Methods according to Bunel, De Palma, et al.

It is worth mentioning the latest projects around the Oxford effort of Rudy Bunel, De Palma et al. They bring some tools that, according to the Crown authors [29], are part of the latest generation of verification tools. An important fact is that Bunel et al. brought the branch and bound phase [7] to the field of verification. Thus, they significantly increased the scalability of verification tools and brought a new generation of algorithms to neural network verification.

Verification via Lagrangian Decomposition

This solution deals with too much looseness of calculated bounds, which leads to unnecessarily large inaccuracies and thus more difficult verification of neural networks. The work thus improves the properties of bounds using relaxations formed by Lagrangian decomposition [6], which overcomes the so-called Lagrangian relaxations [9]. Unlike VeriNet, this is an optimization search approach to neural network verification.

This work builds on dual algorithms, where the authors claim that their new method allows at least as accurate bounds as previous approaches, and this solution α, β-Crown follows. The authors boast in their work that they can stop their algorithm anytime. This stopping is advantageous in setting between the performance price and accuracy. That means that we can stop the algorithm and obtain a roughly current intermediate result from it, which may not be the tightest possible, but is still usable in the subsequent phases of the algorithm. Another benefit of their approach is easy GPU parallelization [6].

Compared to primal optimization, the difference in this approach is that it is necessary to use dual solvers to solve problems obtained by Lagrangian decomposition. They take them from previous approaches to dual optimization, but they also developed in the follow-up approaches. For example, they further design Active Sets Solver [6, 19].

Verification via Active Sets

The main topic of this solution is working with dual solvers, where they propose and implement the so-called Active Set Solver, which, unlike existing solvers (LP), can solve dual problems. Like the Crown solution, this method also represents alpha and beta optimization variables and is based on the above work based on Lagrangian decomposition. This verification tool allows massive GPU parallelization. The toolkit uses branch and bound phase [19].

The benefit of this work is that Active Set Solver better process tighter boundaries. This processing is advantageous because tighter boundaries are more accurate, leading to better verification results and better estimation of network behavior. However, at the same time, tighter boundaries consist of many more constraints that consume much more computing power to solve with a solver. They achieve this by storing an active set of the dual variable [19].

Their tool is again well parallelizable in terms of the branch and bound phases, the calculation of bounds on specific layers, and the presented solver itself [19].

3.6.2 Crown

The Crown tool views formal neural network verifications differently than VeriNet. This toolkit is primarily famous for winning the 2nd International Verification of Neural Networks Competition, and the Huan Zhang group is longly involved in neural network formal analysis. Unlike VeriNet, Crown is more of an ecosystem of not only verification tools, which they gradually develop and expand, among other things, for compatibility with other activation functions[4].

From their point of view, during the seminar on the verification of neural networks, four different generations of NNs verification tools have been created since 2017. The first generation consisted of existing solvers and the second generation of extending scalability with the presentation of incomplete tools. The third and fourth are interesting in that they both talk about the development of tools based on the branch and bound method. According to them, the fourth generation brings GPU acceleration to this method [29]. This finding is intriguing. Although the extension of VeriNet, DEEPSPLIT, was very close to their solution in the already mentioned competition [3], according to this definition, it is a tool one generation behind because DEEPSPLIT is not GPU accelerated. On the other hand, the creators of VeriNet are currently working on GPU acceleration, primarily in the context of the propagation phase.

The two main tools developed under the "Crown brand" are the α-Crown and β-Crown, where α and β indicates unique optimizable variables α and β. Both tools bring massive GPU parallelization and are currently probably (at least according to the last competition [3]) the fastest solutions available.

Unlike VeriNet, their principle is not symbolic interval propagation (SIP, ESIP, RSIP. . .) methods but bound propagation methods based on Lagrangian optimization using linear and quadratic bounds. As a result, the Crown can verify given inputs with relatively tight boundaries. Also, unlike VeriNet, the Crown's principle is not to search for reachable states but to optimize and search for global minima. We can include their approach among the search and optimization methods [24], but resources on this topic vary. For example, the book [18] includes them between search with reachability methods.

In the context of this work, a branching strategy can be interesting. Crown uses a well-established BaBSR calculation strategy that quickly estimates and searches for the nodes with the highest significance in a given network and selects the best one. In addition, they have tested Crown also with an FSB strategy similar to BaBSR and serves on the principle of imitating the bound propagation method for the few best selections by the BaBSR method and selecting the one with the most significant impact. The authors then claim that they also dealt with the Graph neural network (GNN) branching strategy and that their tool is easily extensible in this regard [24].

[4] For more details see: https://github.com/huanzhang12/alpha-beta-CROWN

Chapter 4

VeriNet toolkit

VeriNet is a library of modules used to verify neural networks based on local robustness. It is one of the so-called search algorithms with a focus on reachability. In this chapter, we focus on the main parts of the algorithm pipeline. We take the deep description from articles [13, 14] and the diploma thesis of Mr. Henriksen [11].

4.1 Algorithm overview

The main algorithm of the VeriNet toolkit consists of four blocks (phases; see Figure 4.1). The first phase starts after loading inputs to the verification toolkit. We need to know the structure of the neural network – the number of layers, size of layers, activation functions, weights, biases, and other parameters. We derive input constraints which from training input (primarily images) and some ϵ simply as:

$$input\_node\_constraint = (input\_value - \epsilon; input\_value + \epsilon)$$

where ϵ is the distance for which we require the algorithm to be locally robust.

First phase inputs also include correct classification. We can predict (user chooses correct classification) or calculate it from the input values result – classification is the node at the output layer with the highest value. This phase uses ESIP in the default VeriNet implementation, but we can replace it with other methods. For example, we can replace it with the combination of SIP and RSIP, as they do it in the DEEPSPLIT [14] extension (see 4.5). The result of this phase is the new representation of the input neural network. That includes a matrix of bounds for each node in each layer, a matrix of error values, and output equations[1].

In the second phase, the algorithm seeks to find potential counter-examples in a given representation. The algorithm uses Gurobi LP-Solver[2], which processes the combination of the output equations, the errors, and the split constraints. Thus, it tries to find potential counter-examples through the application of linear programming (optimization). This solver works on the premise that the equation on the correct output node must always return the highest values from all equations. If the LP-solver calls *unsatisfiable*, there are no output interval intersections, no possible counter-examples, and thus the network is safe for the case. If the LP-solver calls *satisfiable*, there may be (but not necessary)

[1] See subsection 3.3.3.

[2] https://www.gurobi.com

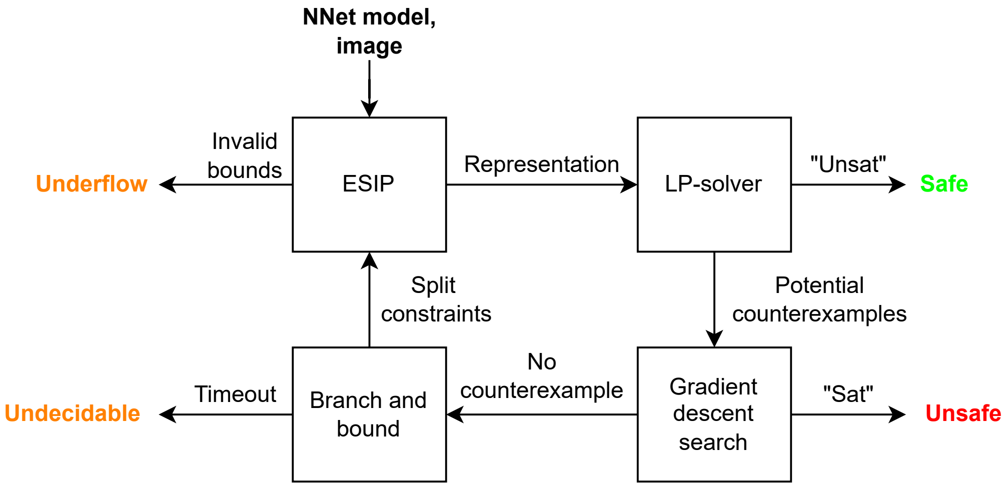


Figure 4.1: VeriNet pipeline scheme. Taken and polished from [11].

some counter-examples. It is just a potential counter-example as it works with a relatively significant overestimation of network behavior.

In contrast to the second phase, the third phase seeks to find real counter-examples. For searching real counter-examples, the VeriNet toolkit uses gradient descent search. If it finds some real counter-example and calls it $satisfiable$, the VeriNet toolkit calls the input case of neural network unsafe. However, standard representation methods, including the VeriNet ESIP, are relatively high overestimated. Thus, if this phase does not find a counter-example, it can either mean that the search area was too broad or there is no counter-example.

Therefore, if the solvers do not find the solution, the fourth phase of branch and bound occurs. This phase is a direct implementation of the branch and bound phase described in the subsection 3.5.4. This algorithm then selects the best appropriate node and adds split constraints to its input. This step creates two new branches, allowing the original solved domain to split into two smaller ones. Running processes then take these new branches, and each process re-performs the entire VeriNet pipeline with some optimizations.

If any of these branches is unsafe, the neural network is unsafe for the input case. The neural network is safe for the input case if all branches are safe. If new branches do not find a solution again, we create the two branches from the original branch again. The algorithm thus creates new branches until it finds the right solution or until it expires the time limit (timeout). An alternative way to terminate the algorithm is the so-called underflow. It says that it was impossible to create the correct new bounds because of the accuracy of floating-point arithmetic.

4.2 Propagation methods

The algorithm implements an error-based symbolic interval propagation (ESIP) method with few improvements leading to better optimization. For example, the ESIP method calculates the bounds for the whole network only for the first iteration. New bounds and equations are calculated only for the layers that follow the current split node. This optimization leads to higher numbers of branches calculated per time unit.

However, this method is still the bottleneck of the verification cycle. The main reason is that the current version is not parallelizable on the GPU or CPU and, therefore, challenging to accelerate. An exception in this may be matrix operations used in calculations by the standard Python library Numpy which is well optimized.

The input of this phase is, logically, a neural network model. The authors encode it with the help of a slightly modified NNet model, which initially comes from the neural networks used in the Acas Xu collision detector. Such a model contains both the amounts and widths of layers, the maximum and minimum possible values on the inputs, individual connections, and other helpful information. In addition, the improved version of VeriNet provides the ability to specify activation functions on individual layers and the types of connections between those layers, including the ability to specify convolution kernels. Another input is some set of inputs (image), and another is correct classification. It is the optional parameter because the toolkit can replace it with the statement that all inputs should have the same resulting classification.

The input of this phase is, logically, a neural network model. The authors code it with the help of a slightly modified NNet model[3], which initially comes from the neural networks used in the Acas Xu collision detector [23]. Such a model contains both the amounts and widths of layers, the maximum and minimum possible values on the inputs, individual connections, and other helpful information. In addition, the improved version of VeriNet provides the ability to specify activation functions on individual layers and the types of connections between those layers. Other inputs are some set of inputs (image) and optionally correct classification. Optional because the toolkit can replace it with the statement that all inputs should have the same resulting classification [11].

4.3 Solvers

VeriNet uses a Gurobi LP-solver. It gives the algorithm information about the existence of potential counter-examples. Also, it adds information about which outputs these counter-examples generate, which we can use in the subsequent phases of the algorithm. In general, we do not perceive this phase as significantly time-consuming.

VeriNet toolkit uses a PyTorch optimized ADAM tool to search real counter-examples default. It does a gradient descent search $L(\boldsymbol{x}) = y_c - y_t$, where $\boldsymbol{x}$ indicates input, the y_c indicates the output value of the correct classification and the y_t indicates the output value of the potentially incorrect classification. The authors set the default setting to do five iterations with a step size equal to 0.1. The optimizer finds real counter-examples relatively quickly with this setting, even with minimal branching. Thus, VeriNet, by default, calls a local search only every five iterations (every fifth branch) [11].

4.4 Branch and bound phase

The main goal of this phase is to find the problematic node, select some of its inputs, divide this input and create new branches according to this split. Unlike some other tools, such as CROWN [28], this phase is parallelized only through the CPU. The principle is that individual processes, so-called VeriNetWorkers, gradually take unprocessed branches from the queue. After completing them, it either creates two new branches (undecided)

[3] https://github.com/sisl/NNet

from them, or when successfully solving the verification task, it passes the result on in the manner mentioned in 3.5.4.

Although the general principle of the branch and bound algorithm might suggest, this branching phase is not directly recursive. It mimics recursion using a classic queue, where it places unprocessed branches. This queue use generally leads to an order of magnitude better memory utilization than when using classical recursion.

In the default version of the program, VeriNet uses adaptive refinement, which estimates the most influential node to split. This phase does not look at the previous split nodes or the subsequent split nodes, so the node selection depends only on the current representation of the neural network in the given branch. The following experiments will focus on reworking this phase.

4.5 Existing VeriNet extensions

Given that verification of neural networks is a very current topic and, at the same time, the scalability and thus the practical usability of verification algorithms is low, many new tools are emerging. This section will list the two most significant enhancements to VeriNet.

4.5.1 DEEPSPLIT

DEEPSPLIT is an extension of the VeriNet toolkit. In addition to improving the compatibility of Python libraries and minor development enhancements, this extension brings significant improvements in two main ways [14].

They made a significant change in the representation methods phase. Instead of directly calculating bounds and equations through the ESIP method, this tool combines two methods – RSIP and ESIP. The RSIP method is very similar in principle to the SIP and ESIP methods, but unlike them, it accesses the network from the other side – it propagates bounds from the last layer to the first layer.

The combination of these two different methods works by first calculating the representation using the RSIP method. ESIP then works with the values calculated by the RSIP method and for each layer counts its representation (bounds). Then it selects the bound that is tighter and therefore more accurate. As a result, it finds fewer false counter-examples, and at the same time, the areas in which we search for real counter-examples are smaller [14].

The second significant change is the extension of the heuristic function with indirect and propagation effects. Indirect effects describe the importance of linear relaxations on the accuracy of the boundaries estimations in the following layers - splitting a node in the previous layer reduces the effect of linear relaxation on the following layers. The propagation effects describe the difficulty of splitting within specific layers - the more profound the node's layer locates, the easier the splitting is. This improved heuristic leads to a better selection of split nodes and thus reduces the number of split nodes needed to resolve the case [14].

In addition to these improvements, DEEPSPLIT changes the original Gurobi LP-solver to a better one. It also improves input parameters of solvers – it adds the possibility of solving problems other than local robustness, improves the search for specific counter-examples, and much more. Together, these improvements led to DEEPSPLIT placing second in the 2nd International Verification of Neural Networks Competition (2021), close behind the α, β-Crown toolkit, making it one of the best state-of-the-art solutions [3, 14].

4.5.2 VeriNetBF

Another version of VeriNet is VeriNet$_{\mathrm{BF}}$, which brings significant improvements over the original VeriNet in many ways. In addition, this work focuses on bias field perturbations and also brings an extension of representation methods.

The extension presents a combination of undemanding SSIP (Standard-SIP) and more demanding and accurate RSIP (Reversed-SIP) for neural network representation. This combination works so that SSIP finds and creates a representation of more stable and less critical nodes, and then VeriNet$_{\mathrm{BF}}$ computes more important nodes using the RSIP method. This combination then advantageously combines the simplicity of SSIP and the accuracy of RSIP [12].

Another advantage over the original VeriNet implementations is GPU acceleration within representation methods. This acceleration leads to significantly better results but at the same time creates a problem with the need to estimate the memory requirements for calculating the given operations. RSIP has huge memory requirements on the GPU and can often run out of memory. In addition, it also provides 64-bit precision compared to the 32-bit precision in most verification algorithms, which is more time-consuming and accurate [12].

The algorithm boasts that it can better verify tools that analyze images from the medical environment thanks to the bias field analysis. For example, in Magnetic Resonance Imagining (MRI), where images may be damaged by intense artifacts. Bias field analysis can verify resistance to these perturbations. The algorithm also boasts that the most extensive network they verify reaches the size of 6.5M nodes, which is a significantly higher number than the NNs we verify in this work [12].

Chapter 5

Extension design

This chapter describes the extension design and implementation details for new branching strategies within the VeriNet toolkit. All proposed strategies are universally applicable and modular to any other toolkit with similar principles as VeriNet (branch and bound, ESIP). All parts of this chapter were designed and implemented during work on this bachelor thesis.

We propose three different strategies in the first three sections. After the main description, we briefly discuss implementation and the class design. In the next Chapter 6, we experimentally evaluated all ideas from this chapter.

5.1 Memory strategies

This section introduces methods that increase the speed of branching at the expense of reducing the quality of node selection. The expected result of these methods is an increase in the required number of branches to solve the given verification tasks but at the same time a reduction in the time required to solve them.

5.1.1 Simple memory strategy

During each iteration of the branch and bound phase, several steps need to be taken - select the node, split it, and create new branches. All of these are non-trivially demanding operations, and therefore this first strategy tries to save on the first-mentioned step.

The basic idea is that instead of counting a new heuristic, each time we split a new node, the heuristic is counted only once in a while. Specifically, we create a stack of n best nodes, and only when they run out is a new heuristic called. The heuristic then returns the n best results instead of one result. These we again store on the stack, from which further iterations of the branch and bound phase take new nodes.

The potential benefit of this method is to save on the operations required to perform heuristics, which include several matrix multiplications, matrix additions, and the best node selection (sorting algorithm). In addition, this method assumes that the estimated impact scores of individual nodes do not change as much over time and that the quality of the heuristics would not deteriorate as much.

The disadvantage may be the deteriorating accuracy of the heuristic for higher n. For sigmoid and tanh networks could be a disadvantage impossibility to choose the same significant node twice in one stack iteration. Another disadvantage is that this strategy does not include the order of split nodes. According to the stack memory, it only takes them

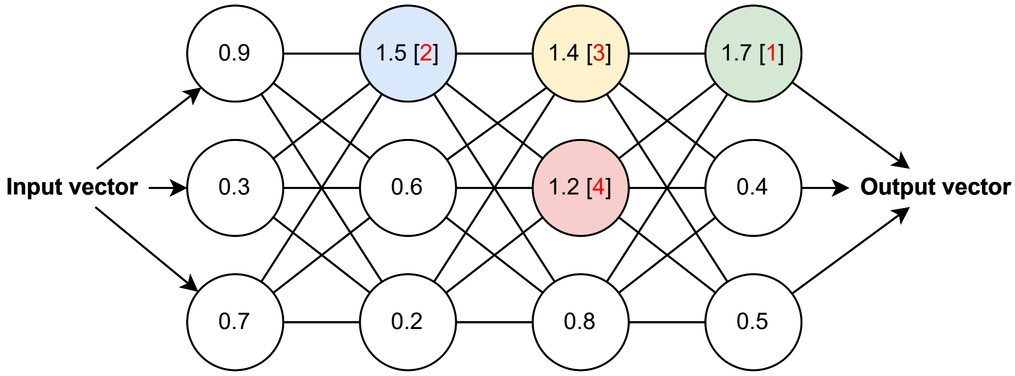


Figure 5.1: An example of a neural network where the first number is the impact score and the second number (optional) is the split node order for **simple** memory strategy. Memory size n is equal to 4.

from the best to the worst. This order can be disadvantageous, as it is more advantageous to divide the nodes at the beginning and then at the end of the network.

For example, as we can see in Figure 5.1 with some impact score values, the node in the last layer is divided first, then in the second, then in the third, and finally also in the third. Thus, given that the new branch is necessary to calculate the representations of the only following layers, it raises the number of needed operations. For example, if we swap the node splitting in the last layer and the second layer, we save six new node representation calculations. If we split in the last layer, we get two representations in which we next split the node in the second layer. Whereas if we split in the second layer, we recalculate bounds in the following six nodes, and after splitting in the last layer, we do not have to recalculate them again.

5.1.2 Sorted memory strategy

The second strategy we propose is the sorted memory strategy. This strategy extends the previous simple memory strategy, where we perform heuristics, n best nodes are selected, and these we save on stack. In the case of a sorted strategy, we sort the obtained results from the input layer to the output layer to reduce the unnecessary recalculation of nodes. The selection of nodes we illustrate in Figure 5.2 (same network as in the case of the simple memory strategy).

The main advantage of this strategy over the simple memory strategy and the default adaptive strategy[1] is that it reduces the number of node recalculations. It thus partially includes indirect effects that the VeriNet heuristic does not account for [11]. As in the case of a simple memory strategy with a larger n, the heuristic accuracy decreases.

5.1.3 Reverse sorted strategy

This strategy aims not to improve verification tools but to show the effect of the order of the split nodes on the verification itself. Its principle is the same as for the sorted memory strategy in Section 5.1.2, we only reverse the order of the divided nodes in such a way that the node from the last layer goes first.

[1] Default VeriNet [11] strategy, briefly mentioned in Section 3.5.1.

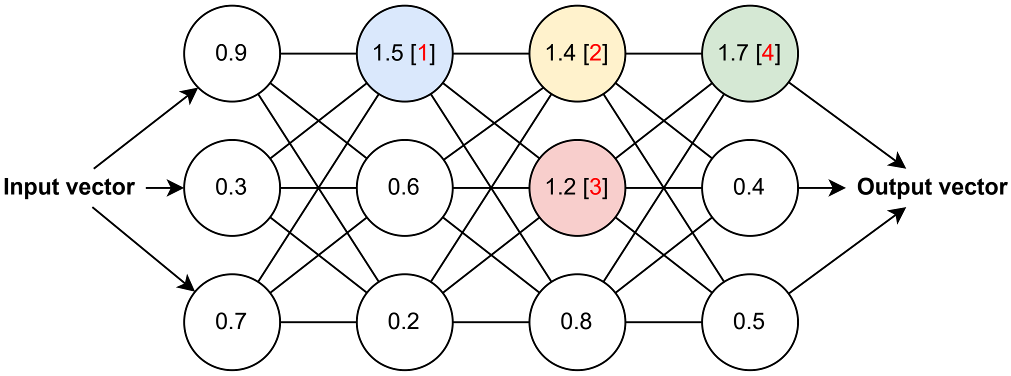


Figure 5.2: An example of a neural network where the first number is the impact score and the second number (optional) the split node order for **sorted** memory strategy. The memory size is equal to 4.

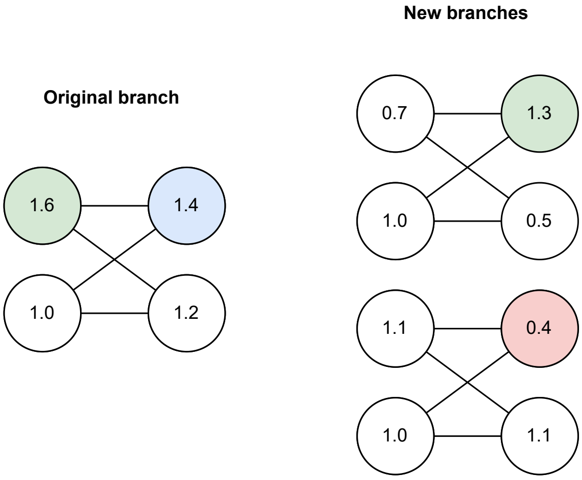


Figure 5.3: Example of branch mirroring effect. Green and red nodes are current split nodes and blue one is a planned split node.

It will be interesting to observe two behaviors in experiments with this solution. On the one hand, experiments will measure the reduction in the rate at which we split nodes per unit of time. Moreover, on the other hand, these experiments may show the effect of the order on the number of branches needed to solve the case.

5.1.4 Branch mirroring

Another significant disadvantage of both described methods may be the problem of branch mirroring. For example, in Figure 5.3 we can see that in the new branches, we split a significant node in one branch, while in the other branch, we split a node with a meager impact score. In both branches, we split the same nodes for some time depending on the size of memory n. This splitting means that in one series of branches, overestimation can be constantly limited in one direction (of correct or incorrect output), which does not approach the solution of the given task. The result can be one or more unnecessarily deep lines of branches that slow down the task.

5.2 Semi-hierarchical strategy

This section proposes a novel strategy for the branch and bound method that combines previous approaches.

5.2.1 Comparison of hierarchical and adaptive splitting

Previous approaches used two main ways how to choose split nodes. Neurify [22] proposed hierarchical splitting, one of the first approaches to the branch and bound algorithm. VeriNet [11], which follows Neurify, proposed a novel adaptive splitting strategy, significantly improving branch and bound mechanisms.

In the case of a hierarchical strategy, we follow an intuitive selection of split nodes. We start with the first node of the first layer and continue to the second and third until we get to the last node of the last layer. Formally, let us consider a list of indices of the neural network nodes `indices` as a one-dimensional array and variable `i` initialized as `i=0`. We get a new node as `node = indices[i++]`. The advantages of this approach are its simplicity and respect for the correct split node order we mentioned in Subsection 5.1.1. However, the disadvantage is that significant nodes we might split late.

The principle of the second approach, the adaptive splitting strategy, is choosing the most significant node in each branch and bound iteration of the representation of the neural network. For this purpose, we use some heuristic functions. If we again suppose array `indices`, we choose split node as `node = indices[ heuristic( representation )]`. This approach chooses the best nodes sooner, but it does not respect the correct split node order. However, the adaptive splitting strategy generally leads to having a better result on larger networks [11, 14].

For example, as shown in Figure 5.4, the second step of the hierarchical strategy plans to split a node with a minimal impact score of 0.3. In contrast, an adaptive split strategy splits nodes with significant influence on network behavior and faster reduces overestimation in the representation.

5.2.2 Best by layer strategy

This strategy represents a combination of an adaptive and a hierarchical splitting. The principle is to calculate the heuristics only for the current layer, from which we select the most critical node. After each iteration (creation of a new branch), we shift the index of the current layer.

We can see an example in Figure 5.5, where in the first iteration, the node with a value of 0.9 has the best score in the first layer. We split the node, and the algorithm continues in one of the new branches so that it currently has the highest node with a value of 1.2 in the second layer.

5.2.3 Potential advantages and disadvantages

This method should have several advantages over previous mentioned strategies. On the one hand, we limit the heuristic calculation for a specific layer, so if the neural network has n layers, then the number of necessary operations for branch calculation is reduced to $\frac{1}{n}$. On the other hand, it solves problems solved in DEEPSPLIT extension [14] more intuitively. It solves propagation impacts by introducing the order into divided branches, and indirect impacts it solves by giving adequate space to all layers. This fairness is related

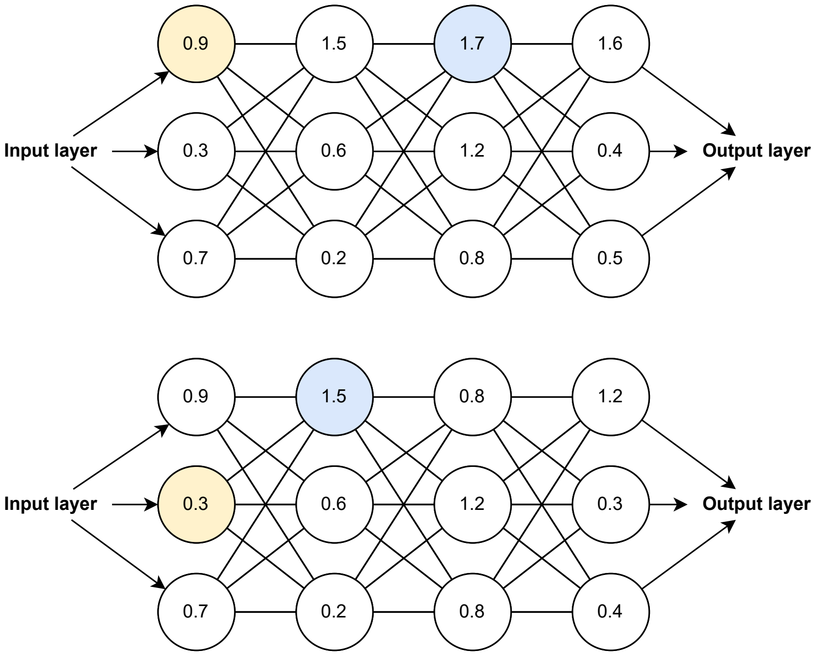


Figure 5.4: Yellow nodes were chosen by hierarchical and blue by adaptive strategy. The second sub-figure illustrates the NN after dividing by the blue node in the first sub-figure.

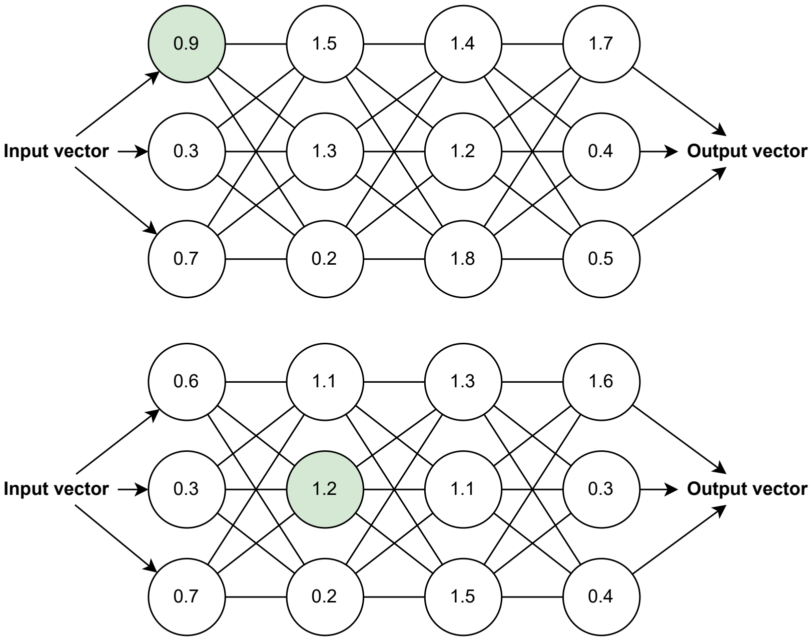


Figure 5.5: Green indicates the nodes that we split according to the best by layer strategy.

to the current problem of VeriNet [11] heuristics, which tends to prefer nodes in later layers of the network at the expense of earlier ones.

Remark (Prioritized layers)**.** *However, while experimenting with networks beyond the default VeriNet package, it has been shown that the VeriNet adaptive heuristic prefers different layers for different networks. More research would be beneficial.*

Compared to the memory-based methods, it has the advantage of less heuristic distortion and also cancels the branch mirroring effect. The advantage may also be the absence of the need to select the memory size, which dramatically affects the quality of memory methods.

The disadvantage of this method may be excessive fairness. We may split less often than necessary some layers with nodes that generate significant overestimation. In addition, this method does not include the possibility of different layer sizes, so if a layer is significantly smaller or larger, it will still go as often as the other layers.

5.3 Alternating impact strategy

As mentioned in section 3.5.2, VeriNet calculates heuristics as the sum of some positive and negative errors. Due to these errors, we create a kind of reasonable heuristic when the most significant node is always selected. In this section, we describe a method that, similarly to the reverse sorted strategy, is used primarily to test the behavior of the verification algorithm.

The goal of this strategy is not to select the best nodes, but the best nodes for positive or negative errors separately. As mentioned in section 3.5.2, the original heuristics have the following formula:

$$s(h) = \gamma_c max(E^m_{c,h}, 0) - \sum_{t \neq c} \gamma_t min(E^m_{t,h}, 0)$$

As a part of this new strategy, heuristics are modified to apply to odd/even branches:

$$s(h) = \gamma_c max(E^m_{c,h}, 0)$$

and to even/odd branches:

$$s(h) = - \sum_{t \neq c} \gamma_t min(E^m_{t,h}, 0)$$

The potential advantage of this strategy is that branching will not reduce the impact in only one direction. The second advantage may be reducing the number of operations required for heuristics.

On the contrary, the disadvantage may be many potential counter-examples. Thus alternating impact strategy will reduce their overestimation less often. Another disadvantage may be asymmetry in that there may be a high reduction of overestimation of correct classification. At the same time, there may be a significantly lower reduction in the overestimation of potentially incorrect classifications.

5.4 General implementation details

As part of the implementation, it was necessary to resolve a few details that do not directly result from the design of new strategies.

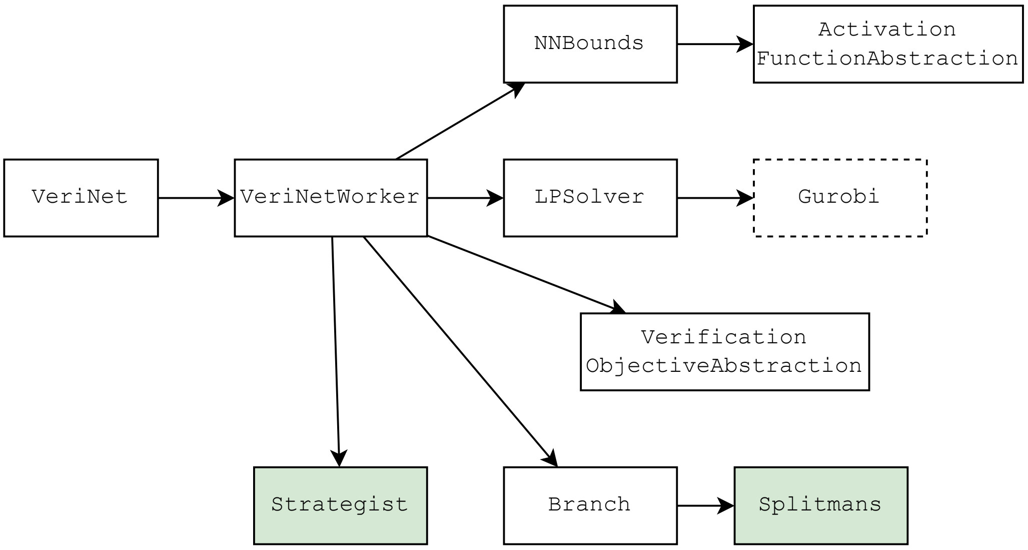


Figure 5.6: The class dependencies diagram of VeriNet extension. Diagram taken and extended from [11].

5.4.1 New classes

Our approach includes an extension by two classes - the Strategist and the Splitmans classes. We design the Strategy class to include static methods for selecting a node to divide, and the Splitmans class we design to add the necessary strategy information to the branches. We can see the proposed scheme in Figure 5.6, where the classes marked in green are the core of the extension.

The Splitmans class primarily includes some memory of the selected nodes, the maximum memory size, or the last split layer. To determine the last split node, it is also necessary to keep the reference to the last selected node.

The Strategist class contains static methods for specifying a new split node and updating data in branches, especially in the embedded Splitmans class. The main goal of this class is to replace the adaptive splitting strategy presented in [11] with more advanced ones. The following sections show the proposed strategies.

5.4.2 Changes in default classes

We implement several changes to create compatibility between extension modules and the default toolkit. The first one is the addition of the memory variable to the main `VeriNet` module. This variable we use in all memory strategies determines the memory size for any strategy. The default value of this variable is 1.

Each time we initialize the main `VeriNet` module, the primary Splitmans entity is initialized, including the value for memory size. When creating additional branches, we send a deep copy of this original entity to one branch, and the other takes over the reference to the original entity. References are possible since we no longer use a closed branch. As a result, we transfer the data needed to work with all strategies for each new branch, i.e., memory index, memory, and layer index. We call new strategies in `_branch()` method of the `VeriNetWorker` class.

Chapter 6

Experiments

This chapter discusses the performance of VeriNet while trying to investigate the behavior of the verification algorithm. It includes a section about our experimental setting, the experiments, and one section for zoomed results to branch implosions we discovered. We also carried out many experiments which did not fit here. We placed them on our GitHub and the submitted attachment.

6.1 Experimental setting

For all experiments, the environment we used was a computer with Ubuntu 20.04 LTS, 16 GB RAM, CPU AMD Ryzen 3 1300X processor, and GPU GTX 1660 with 6 GB memory. The installed libraries are the same as recommended by the authors [11]. The following subsections describe our goals, testing and training dataset, and networks we tested.

6.1.1 Objectives of experiments

Before we proceed to experiments, we must set our objectives. The main ones include:

- Solving more undecided cases. Default VeriNet and other tools have limited scalability of solvable cases, and we want to improve it.
- Preservation or acceleration of already solvable cases.

In addition to these primary objectives, we performed experiments to observe other behaviors of the verification algorithm. That includes branching speed for different strategies, numbers of needed branches, or general observation of the VeriNet behavior.

Remark (Results)**.** *As we mentioned in chapters before, the VeriNet returns for given cases three possible results. These are:*

- *Safe – The VeriNet says the network is locally robust for a given input.*
- *Unsafe – The VeriNet says the network is not locally robust for a given input.*
- *Undecided – The VeriNet was unable to decide in a given timeout.*

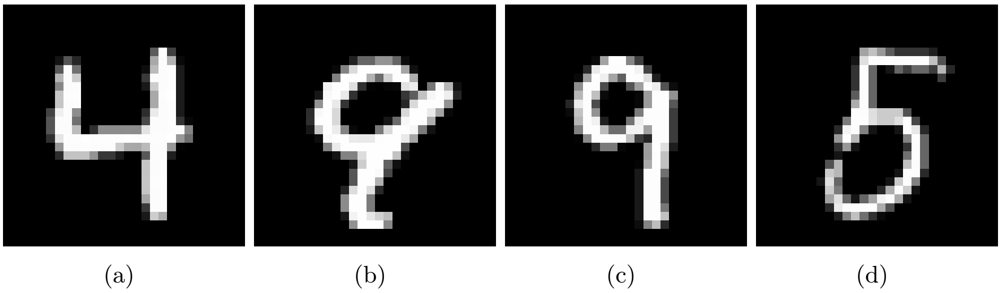


Figure 6.1: The four (a) on the left was relatively unproblematic and was among the less problematic cases. The nine (b) was mistaken for eight by medium-sized networks and either was unsafe very soon or was verified for a long time. The nine (c) then made significant problem with larger and better networks (sigmoid, tanh), when there occurs exponential branching. This case was misclassified with the one. The five (d) was also problematic and was usually confused with the six.

6.1.2 Dataset

For network creation and subsequent verification, it is necessary to use some data for network training and then part of this data for verification as tested inputs. For these purposes, institutions, companies, or other groups create datasets. These are a database of images or records with their correct classifications.

We worked with one of the most common datasets to train neural networks, MNIST. It contains relatively basic pictures of black and white digits with a low resolution of 28x28 pixels. In the context of experiments, we observed that some images were problematic for verification while others were usually undemanding. However, in general, the images were problematic differently for each network – the examples with short comments we have placed in Figure 6.1.

VeriNet uses a clear format without an extension to represent the images. It contains 784 consecutive numbers that describe the intensity of the input. These can take values from `0.0` to `255.0`. The numbers after the decimal point are usually 0. The advantage of this notation is that it does not contain any disturbing additional information. VeriNet reads a set of numbers assigned to the given neural network inputs in a given order.

VeriNet authors have also worked with a network trained on the dataset Cifar-10 while experimenting. Their knowledge refers to their trained network as the largest known verified network[11] (it has over 100 000 nodes). However, as we mentioned in Remark 2.1.1, we did not perform experiments with convolutional networks for technical reasons.

6.1.3 Used networks and strategies

We took trained networks and their models from the VeriNet[1] and Marabou[2] repositories. In Table 6.1, we show which networks we benchmarked.

For some networks, we mentioned in the comment that they were either too large or too small. For too large, we encountered both the problem of weaker setting and too high timeout for the ability to monitor any behavior. We have decided that it is unnecessary to carry out deeper benchmarks for too small networks.

[1] https://github.com/vas-group-imperial/VeriNet-OpenSource

[2] https://github.com/NeuralNetworkVerification/Marabou

Source	Function	Layers	Nodes	Comment
VeriNet	ReLU	2	100	
VeriNet	Sigmoid	6	3000	Best trained network
VeriNet	Tanh	6	3000	
Marabou	ReLU	10	100	
Marabou	ReLU	10	200	Slightly reduced the number of experiments
Marabou	ReLU	20	800	Only few experiments
Marabou	ReLU	6	1536	Too large
VeriNet	ReLU	2	48	Too small
VeriNet	ReLU	2	1000	Only few experiments

Table 6.1: Table of experimented networks. All networks were trained on the MNIST dataset. The nodes are evenly distributed in the layers. The number of nodes includes only nodes in hidden layers.

For ReLU networks, we can state that it had an upper limit for practical experiments at about 800 nodes in our setup. Surprisingly for sigmoid and tanh networks, there was no problem with 3000 nodes. However, we also observed that, despite good verifiability, we were unable to demonstrate any significant results on these networks. Most cases ended in one branch, and the rest we have not resolved by either our or the original version of VeriNet.

Remark (Scalability of verification)**.** *We noted that universally better-trained networks are easier to verify. For this reason, VeriNet authors were able to verify a convolutional network with more than a hundred thousand nodes. The size influences only the time needed to create a representation, while the need for refinement comes from the quality of networks. For example, if we have a poorly-trained or well-trained network, we can decide quickly, but we meet with many undecidable cases for a "half sufficient" network.*

We experimented with the first three networks from Table 6.1 for all strategies. Our semi-hierarchical strategy shows significant improvement over the original adaptive splitting strategy with these networks. Thus, for deeper comparison, we extended the set of benchmarked networks with the rest networks mentioned in Table 6.1.

While experimenting, we use various strategies. For clarity, we created short abbreviations for each one:

- Adaptive splitting strategy – **ADS**
- Semi-hierarchical strategy – **SHS**
- Memory-based strategies – **MS**
  - Simple memory strategy – **SMS**
  - Sorted memory strategy – **STS**
  - Reverse sorted memory strategy – **RSS**
- Alternating heuristic strategy – **AHS**

We are authors of all mentioned strategies but the first one. For all memory-based strategies, we use memory size 20.

6.2 Main experiments

In this section we describe our observations from benchmarked experiments. Each subsection include experiments with one network, primarily with semi-hierarchical strategy. We focus on the number of solved cases and time needed for solving decidable cases.

Remark (Epsilon)**.** *While working with experiments, we need to define the value of epsilon. As we mentioned in previous chapters, the epsilon value tells how wide are neighborhoods of inputs we verify. For example, if we have epsilon with value 1 and input with value 5, we verify robustness for inputs in the interval* $< 4, 6 >$*.*

6.2.1 VeriNet – 100 ReLU nodes in 2 layers MNIST network

This network contains two hidden layers of 50 nodes. Implementing a proposed semi-hierarchical strategy here led to the solution of two new cases in a given timeout (900 seconds), as shown in Table 6.2. Thus, there were no unverifiable cases for the epsilon 5. However, sorted and reverse-sorted memory strategies show worse results than default strategies.

Epsilon	1	2	5	10	15
Solved safe cases SHS	97	93	78	24	3
Solved safe cases ADS	97	93	77	23	3
Solved safe cases STS	97	93	76	23	3
Solved safe cases RSS	97	93	75	20	2
Solved unsafe cases (all)	3	7	22	71	89
Timeout cases SHS	0	0	0	5	8
Timeout cases ADS	0	0	1	6	8

Table 6.2: Comparison of the numbers of results of different splitting strategies. MNIST ReLU 2 × 50. Timeout 900 seconds.

In Table 6.3, we can see that a semi-hierarchical strategy significantly improved the time of safe cases over the default adaptive splitting strategy if we include only the same classifications.

Epsilon	1	2	5	10	15
Safe cases time SHS	2.26 s	2.46 s	748.04 s	913.50 s s	47.72 s
Safe cases time ADS	2.28 s	2.65 s	834.71 s	1313.14 s	86.99 s
Unsafe cases time SHS	0.06 s	0.14 s	0.49 s	1.62 s	2.05 s
Unsafe cases time ADS	0.06 s	0.16 s	0.50 s	1.67 s	1.89 s

Table 6.3: Comparison of sums of times necessary for obtaining the results for each epsilon (timeout 900 seconds). Removed newly resolved cases. MNIST ReLU 2 × 50, semi-hierarchical.

Table 6.4 shows particular cases that changed their classification from undecided to safe. We reduced time from the timeout of 900 seconds more than twice for the first case. Also, we can see that the semi-hierarchical strategy significantly reduced the number of needed branches.

The semi-hierarchical strategy presents a significant improvement in the verification of this network. In contrast, the sorted and reverse sorted memory strategies reduced the

Epsilon	Time SHS	Branches AHS	Branches SHS
5	431.30 s	61897	36599
10	663.67 s	74805	59277

Table 6.4: Comparison of newly resolved cases. Original time was 900 seconds (timeout) with adaptive splitting strategy (ADS) for both cases. MNIST ReLU 2 × 50.

number of solved cases. However, we proved that the split node order significantly impacts verification algorithms that use ESIP.

Within the following experiments, we work with similar tables and metrics.

6.2.2 Marabou – 100 ReLU nodes in 10 layers MNIST network

We can see a significant improvement in this network with a semi-hierarchical strategy. The improvement includes reducing the number of undecidable cases from 30 to 24, which was achieved primarily by significantly reducing the number of branches needed to solve a particular case. Specifically, in Table 6.5, the most significant absolute decrease occurred with the middle epsilon 5 and a small one with epsilon value 2. Conversely, for epsilon 10, we solved no additional case. These different decreases could indicate a trend for this strategy to work better at lower epsilon values. However, it is still only a tiny sample.

Epsilon	1	2	5	10	15
Solved safe cases SHS	83	72	12	0	0
Solved safe cases ADS	83	71	7	0	0
Solved unsafe SHS	17	25	70	97	100
Solved unsafe ADS	17	25	70	97	100
Timeout cases SHS	0	3	18	3	0
Timeout cases ADS	0	4	23	3	0

Table 6.5: Comparison of the numbers of results of the default (ADS) version and the semi-hierarchical strategy (SHS). MNIST ReLU 10 × 10. Timeout was 900 seconds.

In Table 6.6, we can see how much time the verification process took to determine the specific states for each epsilon. We can see that the semi-hierarchical strategy achieved significantly better results for all epsilons for safe cases.

Epsilon	1	2	5	10	15
Safe cases time SHS	32.24 s	759.06 s	9.84 s	0.00 s	0.00 s
Safe cases time ADS	112.73 s	996.38 s	15.81 s	0.00 s	0.00 s
Unsafe cases time SHS	0.38 s	0.69 s	4.07 s	5.28 s	2.19 s
Unsafe cases time ADS	0.38 s	0.69 s	3.98 s	5.15 s	2.16 s

Table 6.6: Comparison of sums of taken time for each epsilon and each result. Removed newly resolved cases. MNIST ReLU 10 × 10.

This strategy also has a relatively significant impact on the number of branches needed to solve specific examples. If we exclude cases where the result has not changed, we get Table 6.7. We can see that the implementation of the strategy significantly reduces the number of branches needed to solve safe cases, with the lowest epsilon even to less than one-third. Regarding the safe part of the table, we can assume that the semi-hierarchical strategy reduces the average number of branches needed to resolve the case and speeds up the branching process.

Epsilon	1	2	5	10	15
Branches for solved safe cases SHS	57.36	2249.45	126.43	nan	nan
Branches for solved safe cases ADS	184.61	3095.73	382.14	nan	nan
Branches for solved unsafe cases SHS	1.00	1.00	1.80	2.37	1.00
Branches for solved unsafe cases ADS	1.00	1.00	1.77	2.37	1.00
Branches for timeout cases SHS	nan	106548.67	78531.11	53653.33	nan
Branches for timeout cases ADS	nan	85457.00	70283.22	50611.00	nan

Table 6.7: Comparison of average numbers of solved branches for cases. Cases with different results were removed. Timeout 900. MNIST ReLU 10 × 10.

The last Table 6.8 demonstrates how the strategy overcomes previously undecided cases. We can see that the number of branches needed to solve has significantly decreased. A smaller decrease occurs only in the last case when there is significant acceleration in the speed of branches solution per time unit. For the original version of VeriNet, the results could be a few branches close, or the result might be hundreds of thousands of branches far. However, to obtain this information, we would have to run the algorithm with a possible order of magnitude higher timeout.

Epsilon	Time SHS	Branches AHS	Branches SHS
2	566.25 s	185345	120361
5	108.52 s	113892	24989
5	454.15 s	116803	75111
5	342.30 s	119094	67557
5	882.46 s	97540	79735
5	595.33 s	111825	110509

Table 6.8: Comparison of newly resolved cases. Original time was 900 seconds (timeout) with adaptive splitting strategy (ADS) for all cases. MNIST ReLU 10 × 10.

6.2.3 Marabou – 200 ReLU nodes in 10 layers MNIST network

We achieved the best improvement within this network with 10 layers of 20 nodes. As shown in Table 6.9, the number of undecided cases in the 1800 second long timeout dropped from 44 to 36. We can assume that the visible acceleration would be even higher with an increase in timeout.

We can see in Table 6.10 that, while excluding cases with different results, we get an almost 50-fold decrease for epsilon 5. It primarily causes branch implosions which we describe in the following Section 6.3.

If we compare the behavior of the verification tool in terms of the average number of solved branches for cases where the results were the same, we get the following Table 6.11. In this comparison, we can see an increase in the number of solved branches for undecided cases. We also see a significant reduction in the number of branches needed to solve

Epsilon	1	2*	5**	10	15
Solved safe cases SHS	92	38	6	0	0
Solved safe cases ADS	91	33	4	0	0
Solved unsafe cases	6	11	32	94	100
Timeout cases SHS	2	18	19	6	0
Timeout cases ADS	3	23	21	6	0

Table 6.9: Comparison of the numbers of results of the default (SHS) version and the extension (ADS). MNIST ReLU 10 × 10. * includes 57 cases and ** only 57 examples in total.

Epsilon	1	2	5	10	15
Safe cases time SHS	140.81 s	65.05 s	75.99 s	0.00 s	0.00 s
Safe cases time ADS	1213.89 s	1344.85 s	3696.19 s	0.00 s	0.00 s
Unsafe SHS	0.14 s	0.30 s	0.89 s	3.34 s	2.35 s
Unsafe ADS	0.16 s	0.30 s	1.08 s	3.29 s	2.34 s

Table 6.10: Comparison of sums of taken time for each epsilon (timeout 900 seconds) and each result. Removed newly resolved cases. MNIST ReLU 10 × 20, semi-hierarchical.

safe cases. The reduction is almost 40 times for epsilon 5, which causes frequent encounters of implosive cases. One of these cases we describe by the network below, and more details we tell about them in Section 6.3.

Epsilon	1	2	5	10	15
Branches for safe cases SHS	106.58	145.12	1213.50	nan	nan
Branches for safe cases ADS	786.12	2475.55	46137.50	nan	nan
Branches for undecided cases SHS	90215.50	61607.83	40872.58	34447.00	nan
Branches for undecided cases ADS	80922.50	57208.83	37311.52	30536.50	nan

Table 6.11: Comparison of average number of solved branches for cases. Cases with different results were removed. Timeout 900. MNIST ReLU 10 × 20.

In Table 6.12, we can see that the semi-hierarchical strategy solved new cases significantly. Over timeouts, it speeds up almost four times, and for epsilon 2 it speeds up over eight times. Moreover, we do not know the real time needed to solve these cases within the original strategy, so that the speed-up may be even higher.

Epsilon	1	2	5
Solved safe cases time SHS	1292.84	1124.96	1417.37
Unsolved cases time (timeouts) ADS	1800.09	9000.39	3600.11

Table 6.12: Comparison of sums of times of newly resolved cases over unsolved cases. Timeout 900. MNIST ReLU 10 × 20.

6.2.4 Marabou – 800 ReLU nodes in 20 layers MNIST network

The deepest network is a network from the Marabou package, containing 20 layers, each with 40 nodes. This network was complicated to verify, and the complete logs were created only for the lowest two epsilon values. In addition, it failed to gain some awareness of general improvement using a semi-hierarchical strategy because of this difficulty. It would be necessary significantly raise the timeout for experiments (more than 1800 seconds).

However, the extension managed to implode the number of necessary branches in one particular case to solve the example. In this case, for the clarity given in Table 6.13, the resulting time decreased 2045 times. The number of branches decreased 1737 times. This implosion occurs more times during experiments, and we describe these anomalies in the next Section 6.3.

Epsilon	ADS time	SHS time	ADS branches	SHS branches
1	1800.07 s (t)	0.88 s	64286	37

Table 6.13: Comparison of newly resolved case. (t) means timeout.

6.2.5 VeriNet – Sigmoid and tanh networks

For these two networks, it was not possible to demonstrate a significantly positive effect of our strategies on the number of resolvable cases. For tanh, it even decreased slightly. Moreover, it was impossible to find a significant trend in the number of resolving branches per unit time. However, it is worth adding that the sigmoid network performed well in verification and was the best of all tested networks. It had only a minimum of unsafe cases and undecided cases.

Epsilon	0.005	0.01	0.015	0.02	0.025	0.03
Safe cases	99	99	97	97	96	95
Undecided cases	0	0	2	1	1	2
Unsafe cases	1	1	1	2	3	3

Table 6.14: Numbers of results of cases according to the results for sigmoid MNIST network. Same for all strategies. Timeout 900 seconds.

Epsilon	0.005	0.01	0.015	0.02	0.025	0.03
Solved safe cases SHS	100	100	99	97	87	47
Solved safe cases ADS	100	100	99	98	87	49
Solved unsafe cases	0	0	0	0	0	1
Timeout cases SHS	0	0	1	3	13	52
Timeout cases ADS	0	0	1	2	13	50

Table 6.15: Numbers of results for tanh MNIST network. Timeout 900 seconds.

To show how specific strategies have performed in terms of branching speed, we show the results of specific strategies on the undecided cases of the tanh network in Table 6.16. We can see that a semi-hierarchical strategy generally does not work well with the tanh network. That is probably because the default strategy prioritizes nodes in later layers that can be split faster and usually do not reach the first layers. On the other hand, we can see quite convincingly the best performance of the sorted memory-based strategy. With the reverse sorted memory-based strategy, we see that it did not do poorly in branching speed, primarily due to the large memory. A problem that we can not see in this table is the significant deterioration in the number of branches needed for solving the case.

Epsilon	0.015	0.02	0.025	0.03
Branches ADS	6803.00	4876.00	10100.23	9919.96
Branches STS	**7149.00**	**5816.00**	**12648.77**	**11133.50**
Branches RSS	6722.00	5770.00	11434.31	nan*
Branches SHS	5544.00	4742.00	8746.077	8467.00
Branches AHS	5832.00	4745.00	10606.54	nan*

Table 6.16: Comparison of average number of solved branches for undecided cases by different strategies. Timeout 900. Tanh MNIST network. * means no experiments were performed. STS is sorted memory strategy.

The main problem with these networks is that the toolkit solves the case during the first branch or creates an exponential number of branches. It is thus possible that the extension would be an improvement if the timeout was set higher. However, the default VeriNet experiments within the original thesis [11] were done on a significantly better setting and with a timeout four times higher. Still, the number of solved cases with default VeriNet was similar to that obtained in this thesis. In addition, with a higher timeout, rounding errors increase, and VeriNet is not complete for tanh and sigmoid networks. So even a higher number of resolved branches would not result. The solution of these networks probably lies in better bounds created by network representation methods.

However, the acceleration of the branch counting speed did not prove to be sufficient to deal with more cases. In general, it has not been possible to prove the usefulness of any new strategy on both sigmoid networks and tanh networks.

6.3 Branch implosions

For all strategies, it was possible to watch two main metrics - the number of branches needed to resolve the case and the speed of resolving these branches per unit time. Generally, we can see some expected behavior. The number of needed branches for semi-hierarchical strategy was lower, for memory-based strategies higher. This tendency also applies to branching speed.

The exceptions that appeared when comparing different strategies we named branch implosions[3]. For example, as described in Subsection 6.2.4, the network did not give any significantly interesting results, except for one implosion. There the number of required branches has decreased at least 2045 times.

This case is not the only one. Moreover, it occurs independently of the strategy used, so sometimes the random case implodes for the adaptive splitting strategy and sometimes for the semi-hierarchical one – the situation of this implosion in favor of other methods we have not found in the experiments. Some of the implosions we found we placed in Table 6.17.

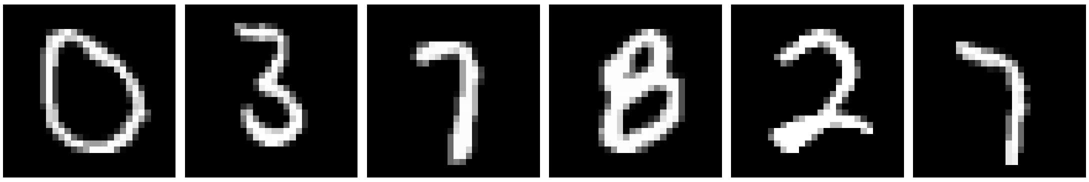
The exact reason why this happens remains unknown, but it is likely that while some nodes are generally suitable for splitting, some are excellent. Moreover, as we can see in Figure 6.2, the images do not look exceptional.

On the one hand, there is the idea that current heuristic methods are not sufficient, and it is necessary to create even better ones. On the other hand, we offer the idea of a

[3] From another perspective, we can call them branch explosions.

Network	Epsilon	ADS time	SHS time	ADS branches	SHS branches
MNIST 20 × 40	1	1800.07 s (t)	0.88 s	64286	37
MNIST 10 × 20	1	1038.33 s	26.27 s	59633	1611
MNIST 10 × 20	2	1800.07 s (t)	6.98 s	94946	573
MNIST 10 × 20	2	1800.05 s (t)	7.89 s	75062	725
MNIST 10 × 20	2	1800.10 s (t)	1.46 s	89384	125
MNIST tanh	0.02	127.09 s	900.09 s (t)	1835	11314
MNIST 6 × 256	2	46.85 s	1772.78 s	1423	63595

Table 6.17: Examples of found branch implosions. (t) means timeout.



(a) (b) (c) (d) (e) (f)

Figure 6.2: Images that caused branch implosions.

new strategy. We could create two different representations on an ongoing basis according to two different strategies, from which we choose the better one. In general, this strategy would roughly double the performance and memory requirements, but at the same time, there would constantly be branching implosions in favor of the strategy.

Another thing that remains unknown is whether these cases are more or less unique or whether they can be achieved artificially in some way. If so, it would be a revolutionary moment for the whole verification with algorithms with branch and bound phase. However, these cases are about over ten in the several thousand cases we examined.

6.4 Summary

We have experimented with different strategies on various benchmarks. Our observations led to three independent findings:

- We proved that the proposed semi-hierarchical strategy significantly improves verification over the original adaptive splitting strategy. We observe a significant speed-up of already solvable cases and a decrease of previously undecidable cases.
- We proved that the order of splitting nodes significantly impacts branching speed. However, the acceleration of our memory-based strategies is smaller than the deterioration of heuristic quality. Thus, it does not lead to verifying more cases.
- We have observed unexpected anomalies we named branch implosions, which led to significantly different processing times and the numbers of needed branches for solved cases.

Chapter 7

Conclusion

In this work, we designed a semi-hierarchical strategy based on selecting the best node from the layer. We proved that this strategy has a general effect on ReLU networks, resolving a more significant number of cases and speeding up already resolvable cases. For some epsilons, the number of calculated branches for solving the case is almost 40 times lower for safe cases. For some specific implosive cases, this requirement has dropped several 1000 times. Moreover, our strategy solves more branches per time unit. Thus we encountered a 50-fold decrease in time for some cases.

In addition, this strategy is very modular. It only needs a heuristic for evaluating nodes at least on a given layer and can thus be implemented in any other state-of-the-art solution. In particular, the branch and bound phase strengthened by this new strategy and GPU acceleration could bring even better results.

Thanks to various strategies, branch implosions have also been discovered. Their targeted search could significantly speed up verification algorithms based on the branch and bound phase.

We also proposed an alternating strategy and three memory-based strategies in this work. However, despite slight improvements in some metrics, their significant contribution has not been demonstrated. Nevertheless, we suppose they can find their application in algorithms that either has more demanding heuristics or have a more significant effect on the order of the split nodes on the speed of verification. Another possibility is that it will be possible to find applications within similar solutions with more powerful devices or another choice of parameters.

7.1 What to do next?

Verification of neural networks is one of the current hot topics for scientists worldwide. They publish new ideas, methods, and experiments. This work does not capture all the possibilities of formal verification of neural networks. This work responds to only one of the topics offered in the original thesis concerning VeriNet [11]. So, to build on the work of the original authors, we also present open possibilities for further research:

- One of the problem domains of verification is time. Therefore, it would be beneficial to perform more experiments in better settings with higher timeouts.

- Memory-based methods have two fundamental weaknesses - heuristic degradation and unlimited parameter selection. Beneficial could be a better heuristic function considering the changes that splitting makes.

- We would like to see an implementation of the semi-heuristic strategy in other verification toolkits. We would like to see how different toolkits with different heuristics and different network representations would behave.

- The current VeriNet cannot solve convolutional networks for technical reasons, so the experiments with them could be beneficial in this respect as well.

- In general, this work gives some basic methods of working with a state approach to the branch and bound phase, which is almost non-existent. However, we can create as many possible strategies for a single step as how many nodes are in the network.

- We prove that there may be significant deviations in the number of needed branches for solving particular cases. Research for analysis of the existence of these branch implosions and finding out why this happens would be beneficial. The design and implementation of its target achieving would lead to significantly better results of verification tools.

- One of the issues that we do not address much is validation itself. Newly emerging tools already have relatively usable scalability. It would be appropriate to find out whether the results of verifications correspond to the quality of the network in practice. In addition, when working with local robustness in general, we must use some epsilon size values. Finding out which one is correct, or if we need more epsilon values for a single case, would also help speed up verification.

Bibliography

[1] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z. et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. 2015. Software available from tensorflow.org. Available at: https://www.tensorflow.org/.

[2] Amato, F., Lopez, A., Pena Mendez, M. E., Vanhara, P., Hampl, A. et al. Artificial neural networks in medical diagnosis. *Journal of Applied Biomedicine*. 2013, vol. 11, no. 2, p. 47–58. DOI: 10.2478/v10136-012-0031-x. ISSN 1214021X. Available at: https://jab.zsf.jcu.cz/artkey/jab-201302-0001.php.

[3] Bak, S., Liu, C. and Johnson, T. The second international verification of neural networks competition (vnn-comp 2021): Summary and results. *ArXiv preprint arXiv:2109.00498*. 2021.

[4] Barla, N. *Self-Driving Cars With Convolutional Neural Networks (CNN)* [[online]]. 2021. Available at: https://neptune.ai/blog/self-driving-cars-with-convolutional-neural-networks-cnn.

[5] Bastani, O., Ioannou, Y., Lampropoulos, L., Vytiniotis, D., Nori, A. V. et al. Measuring Neural Net Robustness with Constraints. *CoRR*. 2016, abs/1605.07262. Available at: http://arxiv.org/abs/1605.07262.

[6] Bunel, R., Palma, A. D., Desmaison, A., Dvijotham, K., Kohli, P. et al. Lagrangian Decomposition for Neural Network Verification. *CoRR*. 2020, abs/2002.10410. Available at: https://arxiv.org/abs/2002.10410.

[7] Bunel, R. R., Turkaslan, I., Torr, P., Kohli, P. and Mudigonda, P. K. A unified view of piecewise linear neural network verification. *Advances in Neural Information Processing Systems*. 2018, vol. 31.

[8] Chang, M., Canseco, J. A., Nicholson, K. J., Patel, N. and Vaccaro, A. R. The Role of Machine Learning in Spine Surgery: The Future Is Now. *Frontiers in Surgery*. 2020, vol. 7. DOI: 10.3389/fsurg.2020.00054. ISSN 2296-875X. Available at: https://www.frontiersin.org/article/10.3389/fsurg.2020.00054.

[9] Dvijotham, K., Stanforth, R., Gowal, S., Mann, T. A. and Kohli, P. A Dual Approach to Scalable Verification of Deep Networks. *CoRR*. 2018, abs/1803.06567. Available at: http://arxiv.org/abs/1803.06567.

[10] Gaňo, M. *Improving Robustness of Neural Networks against Adversarial Examples*. Brno, CZ, 2020. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Available at: https://www.fit.vut.cz/study/thesis/22999/.

[11] Henriksen, P. *Efficient Neural Network Verification via Adaptive Refinement and Adversarial Search.* 2020. Master's thesis. Imperial College London. Supervisor Lomuscio, A. Available at: https://www.imperial.ac.uk/media/imperial-college/faculty-of-engineering/computing/public/1819-pg-projects/Ef%EF%AC%81cient-Neural-Network-Veri%EF%AC%81cation-via-Adaptive-Re%EF%AC%81nement-and-Adversarial-Search.pdf.

[12] Henriksen, P., Hammernik, K., Rueckert, D. and Lomuscio, A. Bias Field Robustness Verification of Large Neural Image Classifiers. In:. The 32$^{\text{nd}}$ British Machine Vision Conference, 2021. Available at: https://www.bmvc2021-virtualconference.com/assets/papers/1291.pdf.

[13] Henriksen, P. and Lomuscio, A. Efficient Neural Network Verification via Adaptive Refinement and Adversarial Search. In: De Giacomo, G., Catala, A., Dilkina, B., Milano, M., Barro, S. et al., ed. *ECAI.* 2020. ISBN 978-1-64368-101-6. Available at: https://ecai2020.eu/papers/384_paper.pdf.

[14] Henriksen, P. and Lomuscio, A. DEEPSPLIT: An Efficient Splitting Method for Neural Network Verification via Indirect Effect Analysis. In: *IJCAI-21.* International Joint Conferences on Artificial Intelligence Organization, 2021, p. 2549–2555. ISBN 978-0-9992411-9-6. Available at: https://www.ijcai.org/proceedings/2021/0351.pdf.

[15] Kocić, J., Jovičić, N. and Drndarević, V. An End-to-End Deep Neural Network for Autonomous Driving Designed for Embedded Automotive Platforms. *Sensors.* 2019, vol. 19, no. 9. DOI: 10.3390/s19092064. ISSN 1424-8220. Available at: https://www.mdpi.com/1424-8220/19/9/2064.

[16] Leucker, M. Formal Verification of Neural Networks? In:. November 2020, p. 3–7. ISBN 978-3-030-63881-8.

[17] Li, X. AirFace: Lightweight and Efficient Model for Face Recognition. *CoRR.* 2019, abs/1907.12256. Available at: http://arxiv.org/abs/1907.12256.

[18] Liu, C., Arnon, T., Lazarus, C., Strong, C., Barrett, C. et al. *Algorithms for Verifying Deep Neural Networks.* 2020.

[19] Palma, A. D., Behl, H., Bunel, R. R., Torr, P. and Kumar, M. P. Scaling the Convex Barrier with Active Sets. In: *International Conference on Learning Representations.* 2021. Available at: https://openreview.net/forum?id=uQfOy7LrlTR.

[20] Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D. et al. Intriguing properties of neural networks. *ArXiv preprint arXiv:1312.6199.* 2013. Available at: https://arxiv.org/abs/1312.6199#.

[21] Tjeng, V. and Tedrake, R. Verifying Neural Networks with Mixed Integer Programming. *CoRR.* 2017, abs/1711.07356. Available at: http://arxiv.org/abs/1711.07356.

[22] Wang, S., Pei, K., Whitehouse, J., Yang, J. and Jana, S. Efficient Formal Safety Analysis of Neural Networks. *CoRR.* 2018, abs/1809.08098. Available at: http://arxiv.org/abs/1809.08098.

[23] Wang, S., Pei, K., Whitehouse, J., Yang, J. and Jana, S. Formal Security Analysis of Neural Networks using Symbolic Intervals. *CoRR*. 2018, abs/1804.10829. Available at: http://arxiv.org/abs/1804.10829.

[24] Wang, S., Zhang, H., Xu, K., Lin, X., Jana, S. et al. Beta-CROWN: Efficient Bound Propagation with Per-neuron Split Constraints for Complete and Incomplete Neural Network Verification. *CoRR*. 2021, abs/2103.06624. Available at: https://arxiv.org/abs/2103.06624.

[25] Xiang, W., Tran, H. and Johnson, T. T. Output Reachable Set Estimation and Verification for Multi-Layer Neural Networks. *CoRR*. 2017, abs/1708.03322. Available at: http://arxiv.org/abs/1708.03322.

[26] Xiang, W., Tran, H. and Johnson, T. T. Reachable Set Computation and Safety Verification for Neural Networks with ReLU Activations. *CoRR*. 2017, abs/1712.08163. Available at: http://arxiv.org/abs/1712.08163.

[27] Xu, K., Zhang, H., Wang, S., Wang, Y., Jana, S. et al. Fast and Complete: Enabling Complete Neural Network Verification with Rapid and Massively Parallel Incomplete Verifiers. In: International Conference on Learning Representations. *International Conference on Learning Representations*. 2021. Available at: https://openreview.net/forum?id=nVZtXBI6LNn.

[28] Zhang, H., Weng, T., Chen, P., Hsieh, C. and Daniel, L. Efficient Neural Network Robustness Certification with General Activation Functions. *CoRR*. 2018, abs/1811.00866. Available at: http://arxiv.org/abs/1811.00866.

[29] Zhang, H., Xu, K., Wang, S. and Hsieh, C.-J. *Neural Network Verification Tutorial* [[online]]. Last seen in 2022. Available at: https://neural-network-verification.com/.

Appendix A

Project usage

Project parts description

The storage medium (SD card) and the data submitted to NextCloud contain:

xhudak03_verification_NN.pdf Complete text of bachelor thesis.

README.md Short description of our project.

results Includes our experimental results. Includes scripts for printing tables and converting MNIST raw images to real images. It contains README.md with a short description.

VeriNet Includes the whole VeriNet project, including our extension. It also includes a set of benchmarks and script `script.sh` for running benchmarks. The folder contains README.md with a short description.

doc Includes all source codes from Overleaf LaTeX documentation. The structure is the same as the template.

Installation and usage

Since our implementation is only an extension of the existing VeriNet toolkit, we strongly recommend following the installation instructions of the original authors from:

- https://github.com/vas-group-imperial/VeriNet-OpenSource
- Appendices of the VeriNet original thesis [11]

In the following subsection, we describe the main requirements.

Mandatory libraries and software

For proper working of the VeriNet toolkit, the user device must have installed Python version at least 3.6. For all functionalities, it is necessary to have the following libraries installed with the appropriate versions:

We strongly recommend installing identical versions of the libraries due to possible incompatibilities between the new versions. During the installation, we encountered a problem with the compatibility of the NumPy library with other libraries - in this case, choose any that is compatible with others.

Name	Version	Name	Version
llvmlite	0.32.1	torchvision	0.5.0
torch	1.4.0	numba	0.47.0
matplotlib	*	scipy	*
ipykernel	*	tqdm	*
cython	*	numpy	*
onnx	*	Gurobi	*

Table A.1: Table of libraries and their versions.

Gurobi

For installation of Gurobi we recommend following the instructions at:

- https://abelsiqueira.github.io/blog/

Alternatively, our experience shows that we can easily install Gurobi by pip package-management system or Conda. However, we still have to download grbgetkey and install a license on our device. License is free for academic institutions.

Usage

Due to the need to prohibit CUDA parallelism, we created a short script `script.sh`. With this script, we can run a specific benchmark with the command:

- `./script.sh benchmark_name.py`

Strategies

Our primary strategy is a semi-hierarchical strategy, which we set by default. If we want to change the current strategy, we need to open `./src/algorithm/verinet_worker.py` and find function `_branch()` (approximately at line 340), add `#` before the current strategy, and remove the `#` from the new strategy. If we want to create a new strategy, we need to create a function that returns to variable `split` two integer numbers – the layer position and the node position.