Verification and validation of Machine Learning techniques

Julien Girard-Satabin

université
PARIS-SACLAY

Verification and validation of machine learning techniques

Vérification et validation de techniques d'apprentissage machine

Thèse de doctorat de l'université Paris-Saclay

École doctorale n° 580, Sciences et Technologies de l'Information et de la Communication (STIC)
Spécialité de doctorat: Informatique
Unité de recherche: université Paris-Saclay, CEA, Institut LIST, 91191, Gif-sur-Yvette, France.
Référent: : Faculté des sciences d'Orsay

Thèse présentée et soutenue à Paris-Saclay, le 9 novembre 2021, par

Julien GIRARD-SATABIN

Composition du jury:

Gilles DOWEK Directeur de recherche, INRIA (Université Paris-Saclay)	Président
Pawan KUMAR Professeur, Université d'Oxford	Rapporteur & Examinateur
Antoine MINÉ Professeur, Sorbonne Université	Rapporteur & Examinateur
Sylvie PUTOT Professeur, École Polytechnique	Examinatrice
Caterina URBAN Chargée de recherche, INRIA (École Normale Supérieure rue d'Ulm)	Examinatrice

Direction de la thèse:

Marc SCHOENAUER Directeur de recherche, INRIA (Université Paris-Saclay)	Directeur de thèse
Guillaume CHARPIAT Chargé de recherche, INRIA (Université Paris-Saclay)	Coencadrant
Zakaria CHIHANI Ingénieur-chercheur, CEA LIST	Coencadrant

Thèse de doctorat
NNT: 2021UPASG080

Remerciements institutionnels

Cette thèse n'aurait pu se dérouler dans des conditions décentes sans le soutien matériel de différents acteurs. Je tiens donc à remercier le Commissariat aux Énergies Atomique et Renouvelables (CEA) et l'Agence Innovation Défense de la Direction Générale de l'Armement (AID - DGA) pour avoir contribué au financement de ma thèse.

Je remercie également l'Institut National de Recherche en Informatique et Automatique (INRIA), le Laboratoire Interdisciplinaire des Sciences du Numérique (LISN, ex LRI) et l'université Paris-Saclay d'avoir fourni les infrastructures, ainsi que des opportunités d'enseignement.

Remerciements

Une thèse, c'est court et long à la fois. Court, parce que trois ans pour prendre ses marques dans un domaine scientifique que nos très maigres connaissances préliminaires nous permettent tout juste d'appréhender, comprendre ses problématiques et pouvoir y apporter un petit quelque chose, le tout en prenant en compte les aléa de la vie, c'est court. Long, parce que trois ans, à l'échelle d'une vie, ce n'est pas négligeable. Quand presque la moitié de la thèse se déroule en confinement strict avec un contact très limité avec les amis, la famille et les collègues, le temps parait s'étirer déraisonnablement.

Ces petits paradoxes temporels, mélange de temps court et long, je n'aurais sans doute pas pu les supporter seuls. J'ai été diagnostiqué d'une dépression durant ma thèse; dépression dont, à l'écriture de ce chapitre, je suis pratiquement guéri. Ce n'est pas quelque chose de facile à vivre du tout; j'ai eu l'immense chance de bénéficier de soutien et d'aide de toute provenance. La prévalence des troubles mentaux chez la population de doctorants tend toutefois à montrer que d'autres ont plus souffert que moi, et qu'il serait bon de s'interroger sur les mécanismes politiques et organisationnels qui fragilisent de brillantes personnes qui s'engagent dans le plus haut niveau d'études existant.

Voici donc le traditionnel mais ô combien attendu chapitre de remerciements. Les gens qui me subissent depuis un certain temps savent que je ne suis pas avare de compliments au point de parfois en abuser. Qu'ils soient assurés toutefois: chaque mot dans ce chapitre est soigneusement pesé, réfléchi et néanmoins tout à fait sincère.

Tout d'abord, je remercie en premier lieu mon encadrement de thèse, incarné dans les personnes de Marc, Zakaria et Guillaume. Être encadré par ces personnes à la rigueur scientifique exemplaire et à l'intuition impressionnante m'a aidé à comprendre et intégrer ce que devait être un scientifique. Les nombreuses incitations qu'ils m'ont donné à diffuser mon travail et prendre part à la communauté scientifique ont été appréciées à leur juste valeur. Je tiens enfin à profiter de l'occasion pour présenter mes excuses pour mon caractère parfois bougon, qui a pu se manifester lors de revues d'articles en dernière minute, ou de correction mineures mais importantes dans des présentations. Leur humanité chaleureuse m'a aidé à traverser des périodes sombres de ma thèse, et leur soutien sans faille quand des idées s'avéraient infructueuses (et dieu sait qu'il y en a eu) ou dans les dédales byzantins de l'administration de l'école

doctorale sont des souvenirs que je chérirai longtemps.

Je remercie les membres de mon jury d'avoir bien voulu relire mes travaux. Quand on est tout jeune chercheur, se voir juger par des experts qui peuplent une partie non négligeable de sa bibliographie peut être très intimidant. Je suis donc très reconnaissant de leur bienveillance et la pertinence de leurs questions durant la soutenance. Je remercie en particulier Gilles Dowek, que j'avais déjà rencontré lors de ma soutenance de mi-parcours et m'avait déjà donné à l'époque de judicieux conseils.

Un doctorant sans son laboratoire ne fait pas grand chose, et, bien loti que je suis, j'ai eu la chance de travailler au sein de deux équipes de grande qualité. Je remercie le Laboratoire de Sûreté et de Sécurité des Logiciels et l'équipe projet TAU de m'avoir accueilli et hébergé pendant ces trois ans. Ce remerciement s'adresse non seulement aux permanents de l'équipe, mais également aux doctorants, contrats courts et stagiaires. Vous avez tous contribué à votre manière à ma thèse, et vous êtes malheureusement trop nombreux pour que je vous cite tous ici. À défaut de personnes, je citerai des événements pour lesquels je suis heureux de vous avoir connu: les sorties escalades, les soirées Ghibli, les pique-nique à la Troche alors qu'on devait encore porter les masques, les randonnées, les soirées jeu de société, une dégustation fort arrosée de Beaujolais nouveau, des pauses café dont on aimerait qu'elles ne finissent pas, des après-midi aux Premier Samedi du Libre, de longues séances sur le Canapé de la Déprime, un séminaire de rédaction de thèse en montagne, nourrir un bébé au biberon en s'y mettant à plusieurs, de longues séances de discussion fructueuses en pause thé au deuxième étage d'un certain laboratoire de recherche en informatique. Je souhaite néanmoins remercier nominalement Maëva (de TAU) Frédérique et Ewa (du LSL). Leur véritable travail n'est que trop partiellement capturé par leur intitulé de poste, "assistantes administratives": elles ont constituté l'huile nécessaire à faire tourner les rouages des unités de recherche respective. Leur disponibilité sans faille et les innombrables petits coups de mains qu'elles ont donné méritent beaucoup plus que les remerciements que je formule ici. De la même manière, je remercie Tristana et Diane, qui ont été d'une réactivité rare pour organiser une soutenance en hybride, avec des modifications de planning la veille pour le lendemain.

Assez tautologiquement, je ne serai pas là sans mes parents ni ma famille. Derrière cette formule un peu bateau se cache des trésors insoupçonnés d'affection dont, il est vrai, un enfant a parfois du mal à apprécier la juste valeur. Il en faut du courage pour élever un enfant, il faut beaucoup d'amour aussi. Ma famille n'a manqué d'aucune de ces qualités, et m'en a donné à profusion. Je remercie donc Christian, Marie-Carmen, Mathilde, Consuelo, Jean-Raymond, Patricia, Marie-Lise, Paul, Jacqueline et Sarah d'être là, simplement. Merci à mamie Marthe, dont je suis certain qu'elle aurait été très heureuse de me voir soutenir.

Je remercie très fort et en continu les amis de Salle T Alumni et son adhérence. Merci Rémi pour tes blagues sur les anciens pays de l'Axe. Merci Léopold pour ta

barbe et ton humanité soyeuse. Merci Eugénie pour ta droiture d'âme qui m'a beaucoup apporté. Merci Pauline parce que t(chou) t(chou). Merci Florian pour être un chef crêpu dont le tout jeune fils m'a offert une motivation insoupçonnée. Merci Benoît pour être un chef barbu de fort bon conseil pour les choses de la vie, de la recherche et du hasard. Merci Servane pour être une fren d'excellente compagnie dont l'énergie et l'ingéniosité rare doivent être mentionnées. Merci Basile pour être un fabuleux bonhomme. Merci Baroux pour être agréable de discussion fertile, malgré nos désaccords (qu'ils puissent demeurer féconds!). Merci Aliénor pour être géniale, continue de devenir top. Merci Elliot pour avoir constitué une ancre et un modèle à de nombreux égards. Merci Quentin pour être un *conoisseur* de liqueurs qui, il faut l'avouer, ont été fort bienvenues à certains moments de cette thèse. Merci Lucille-Marie pour ta vivacité cachée. Merci Élise pour être giga mimi. Merci Zoé pour être zozo. Merci Manu pour avoir quelqu'un avec qui perdre mes moyens devant un fichier son saturé. Merci Aurore pour LE MANGER (et bien plus). Merci BPJ pour être l'excès de sucre et de gras incarné, Merci Guillaume pour être le meyeur grapin. Merci Étienne pour être une barbe fleurie capable d'expliquer n'importe quel jeu de plateau complexe à une assemblée inconnue. Merci Lukô pour être l'aventurier de mes amis. Merci Youssef pour rester un roc solide malgré la tempête. Merci Hugo pour être un daron. Merci Adrien pour être ce qui se rapproche le plus de l'incarnation de classe. Merci Florian pour être de bonne humeur constamment, et merci Aurélie pour ton énergie elle aussi constante, mais aussi haute. Je remercie enfin Benoît, Dara et Juliette pour avoir été mes colocataires successifs durant ma thèse: ils ont été au plus près de l'action, si on peut dire, et leur compagnie, discrète ou ample quand nécessaire (ainsi que leurs compétences culinaires qu'il faut féliciter) constitue une pièce non-négligeable de cet ouvrage. Les oublis éventuels de cette énumération ne sont pas intentionnels; toute offense contractée mérite compensation en boisson alcoolisée (ou pas), en restaurant ou en toute autre monnaie jugée pertinente. Sauf Emilien, lui c'est volontaire qu'il ne soit pas mentionné. À vous tous dans votre ensemble, merci pour les moments que nous avons partagé et les moments que nous partagerons encore.

Un grand merci aux tomos de NDM, qui sont dans ma vie depuis très, très, très longtemps. Xetausse, Bidoman, Troll, Nor, mon bon Mad, Rafou et Nens mes gars sûrs et tous les autres: vous m'avez aidé à tenir de toutes manières différentes (c'est d'ailleurs un peu la faute à Xetausse si j'ai commencé une thèse). Qu'il s'agisse d'avoir des gens avec qui discuter d'à quel point la seule série Nintendo qui vaille le coup, c'est Metroid (ou pas), de trop rares IRL ou parcourir certains canaux du Discord à la recherche d'une excuse pour ne pas boucler un article, j'ai passé avec vous beaucoup plus de temps qu'avec une part importante de mon entourage. Si ces liens sont en grande partie virtuels, ils n'en sont pas moins forts.

Commencer une activité sportive en troisième année de thèse, on me l'a déconseillé plusieurs fois. Si il est vrai que la troisième année est généralement plus intense en

charge de travail, il n'en est que d'autant plus important d'avoir des activités pour évacuer le stress et établir un sain équilibre entre vie personnelle et vie professionnelle. Je remercie donc chaleureusement Masato-sensei et l'ensemble des élèves de l'école Sayu - les Deux Spirales pour l'enseignement qu'ils me prodiguent. Le kenjutsu et le iaïdo, que j'apprends à leur côté, m'ont aidé à reprendre conscience que je n'étais pas qu'un cerveau qui habitait un corps malhabile.

Je remercie celles et ceux qui m'aident à cultiver mon humanité, depuis le début de ma thèse et avant.

Merci à Lou de m'apporter tant et de susciter du beau chez moi.

Merci à Ludovic d'être toujours là et de persévérer. Ta régénération ne fait que commencer.

Merci à Bill Wurtz, FromSoftware, Coffee Stain Publishing, Autechre, Clark, John Coltrane, FlyByNo pour avoir peuplé mes loisirs, mes esgourdes et mon mental durant ces trois ans.

Contents

Part I

Introduction

Chapter 1

Synthèse en français

1.1 Synthèse en français

L'apprentissage machine, en particulier au moyen des réseaux de neurones artificiels, connaît depuis une dizaine d'année une expansion impressionnante. détecteurs de collision d'aéronefs, aide au diagnostic pour différents cancers, aides aux décisions de justice, véhicules autonomes et capteurs d'anomalies d'ancrage sur des plateformes offshores sont autant d'applications faisant intervenir les technologies d'apprentissage profond au sein de systèmes critique; ouvrant des perspectives inexplorées pour les sociétés humaines. Bien que bénéfique en apparence, cette révolution a de quoi inquiéter à mesure qu'elle se concrétise: la fragilité de ces techniques d'apprentissage est désormais un fait scientifique établi. La taxonomie des vulnérabilités, qu'elles soient accidentelles ou malicieuses, ainsi que leur caractère imprévisible remet en question la possibilité d'intégrer des réseaux de neurones dans des domaines critiques qui pourraient pourtant en bénéficier.

A l'heure actuelle, peu de méthodes permettent de démontrer formellement la fiabilité d'un réseau de neurones. Par contraste, le domaine du logiciel critique, quant à lui, jouit d'une multitude de méthodes et techniques: *model checking*, simulation, interprétation abstraite, tests dirigés, etc.

L'objectif de cette thèse est de réconcilier l'abondance des techniques de vérification de programmes classiques et l'absence de garanties sur les réseaux de neurones, ce pour permettre aux logiciels critiques de conserver le haut niveau de confiance qu'ils ont atteint quand ils seront inévitablement modifiés avec des mécanismes d'apprentissage machine.

Nous étudions d'abord en détail les raisons qui empêchent l'application directe des approches de vérification formelle classique. Nous mettons ainsi en évidence que l'absence de spécification formelle inhérente aux systèmes dont les entrées sont à hautes dimensions, l'explosion combinatoire entraînée par la structure linéaire par

morceaux des réseaux de neurones, et l'absence de représentation adaptée à la vérification empêchent l'emploi de la plupart des approches.

De plus en plus d'acteurs industriels utilisent des simulateurs pour entraîner et tester leurs programmes. Cette pratique largement répandue et documentée est également employée pour les réseaux de neurones. Pour pallier le problème de la spécification, nous présentons CAMUS, une approche théorique permettant de formuler un problème de vérification formelle avec l'aide d'un simulateur. CAMUS offre une formalisation de la prise en compte d'un simulateur comme fournisseur d'entrées à un réseau de neurones. Le problème de vérification formelle est reformulé de sorte à prendre en compte les différents scénarios que le simulateur est capable de générer, permettant ainsi une définition formelle des entités composant la preuve.

La grande polyvalence et les résultats impressionnants des réseaux de neurones modernes viennent en partie de leur non-linéarité. Cette propriété fondamentale rend malheureusement très difficile leur vérification formelle, et ce, même si on se restreint à une structure linéaire par morceaux. Cependant, chacune de ces régions linéaires prise indépendamment est simple à analyser. Nous exploitons cette structure linéaire par morceaux pour proposer DISCO, un algorithme de vérification parallèle pour alléger l'explosion combinatoire, en opérant une séparation du problème original en sous-problèmes linéaires. Nous présentons également des résultats concernant la structure de ces régions linéaires ainsi que leur similarité.

Nous implémentons nos contributions au sein de l'Inter Artificial Intelligence Standard Encoding Hub (ISAIEH), un prototype de plate-forme d'encodage pour les réseaux de neurones à vérifier. ISAIEH est un logiciel écrit en langage OCaml qui implémente une représentation intermédiaire universelle pour tous les réseaux de neurones gérant le standard Open Neural Network eXchange (ONNX). Cette représentation intermédiaire offre un support pour l'expression de propriétés formelles, notamment la conversion du réseau de neurones au format standard de vérification via calcul de Satisfaction Modulo Théorie (SMT) SMTLIB2.

1.2 English summary

Machine Learning techniques, Neural Networks in particular, are going through an impressive expansion, permeating various domains, becoming the next frontier for human societies. Aircraft collision avoidance, cancer detection, justice advisors, autonomous vehicles, or mooring line failure detection are but a few examples of Neural Networks applications. This effervescence, however, may hold more than benefits, as it slowly but surely reaches critical systems. Indeed, the remarkable efficiency of neural nets comes at a price, more and more underlined by the scientific consensus: weakness to environmental or adversarial perturbations, unpredictability... which prevents their full-scale integration into critical systems.

While the domain of critical software enjoys a plethora of methods that help verify and validate software (abstract interpretation, model checking, simulation, bounded tests...), these methods are generally useless when it comes to Neural Nets.

This thesis aims at bridging formal software verification and machine learning, in order to bring trust in critical systems incorporating Neural Networks elements.

We first study the exact causes that prevent a straightforward application of existing verification techniques on Neural Nets. We state that those issues are threefold: the lack of formal specification on the inputs, the combinatorial explosion caused by the piecewise linear structure of Neural Nets and the lack of a representation common to Neural Nets and Formal Verification. To tackle those issues, we present CAMUS, a theoretical framework allowing the specification of verification problems on perceptual inputs using simulators. We exploit the piecewise linear structure of neural networks on DISCO, an algorithm of parallel verification, to mitigate the combinatorial explosion. We implement these contributions into ISAIEH, a prototypal platform for neural network encoding and verification.

Chapter 2

Introduction

At the time of the redaction of this manuscript, the ubiquity of software in our lives is not a question. This chapter is written with a text editor, on a laptop running an operating system. This same laptop was used to develop experiments, using various programming languages; to plan meetings (and because of the COVID-19 pandemic, attend to them) and discuss with collaborators. Part of this manuscript was written in a train, whose ticket was booked on an online platform. Said train was guided on assigned railways using semi-automated decision procedures. Music tracks were played to help during writing, streamed from a remote server; some of this music was composed using synthesizers and sequencer, both pieces of software. Those small moments of life, centered around this manuscript, were enabled and influenced by software.

But software does not only shape individual lives; it also governs collectives: communities, societies, nation states, corporations and non-profit organizations now rely on software to function to varying degrees. Establishing milestones and tracking their progression in software development companies is a critical process, powered by software. It is also software that controls critical parts of a nuclear power plant. Finally, modern social networks have a strong influence on the public debate by selecting and presenting opinions and facts. In his book, Code 2.0, Lawrence Lessig [Les10] defined the cyberspace as the global environment defined by machine code. Bulletin boards, personal blogs, commercial websites, bank and financial exchanges, every packet transferred on the internet is a part of cyberspace. This space, however vast, was more or less separated from the physical world. This separation still remains in places untouched by human presence, or in certain areas of human lives of which we choose to exclude software. But since the release of Lessig's book, software has become even more pervasive. Nowadays, a lot of human activities include software at some point – and is reciprocally shaped by it, blurring the border between cyberspace and physical space.

Remark 1. We think the word "physical space" is not the proper wording to designate

everything that happen outside cyberspace, since the consequences of activities taking place in the cyberspace have quite direct material consequences; internet works by connecting kilometer long cables, and datacenters have a growing ecological impact. We will keep the word “physical space” for the sake of common understanding, but other wordings such as “corporeal space” or “material space” may better suit what we mean.

As such, one may consider that software is a member of our society, as it is an artifact that is deeply interlinked within human lives and communities. It is not to say here that software produced by human is in any ways “aware”, “alive” or should be considered as anything else than a tool we use for our own designs. Rather, it is only to acknowledge the prevalence of this tool in modern lives, and recognize that this prevalence is shaping those lives on a wide scale, sometimes much larger than initially intended. Software is now an intermediary agent for all activities related to commerce, with digital payment. Software enabled communication and new ways to socialize - ways that allowed crucially needed socialization during the COVID-19 pandemic lockdowns.

Societies require a certain amount of trust between its members to properly function. People living in cities trust each other to not murder each other when they sleep. Parents trust the teachers of their children to educate them reasonably well while they are working. Citizens trust the people organizing elections to not tamper with the democratic process, so that their will is correctly translated in their vote. We expect physicians to properly do their job of healing us when we are hurt. When buying something from someone, we trust the seller to respect its part of the deal and give us the property of the commodity. We expect our supervisors and colleagues to advise us, give constructive feedback and help us and science in general to further enlight the future of humankind. All of those examples rely on informal rules (tradition, courtesy, willingness to do a good job) and formal ones (laws, oaths); and violations of these sets of rules tend to result in exclusion, or other kind of blame. Any member of society aiming to integrate within it must be trustworthy, in a way. Trustworthiness is enforced with education and behaviours that (partially) ensures that unwanted behaviours will not happen. As an artifact member of society, software should (ideally) be subject to the same expectations. Thus arise this question: what are the reasons we trust software enough to integrate it deeply in our societies?

Trust is a complex notion that may not seem in the scope of computer scientists. It intertwines emotions, unspoken rules and is subject to change depending on the context: it seems difficult to characterize and manipulate defined this way. A sub notion of trust is “reliability”. Reliability can be defined as the knowledge that someone or something will behave according as we expect it to behave. Reliability is a key component of trust in multiple interactions between humans and humans, or humans and artifacts. Someone that stays right to their word will be given more attention when they make

a promise. As long as we are employed by an institution, we expect to receive a salary each month. A train that is able to reliably transport us will build trust in train as a transportation system. A software that provably behaves according to a specification will lay ground for trusting it on standard operational conditions. In computer science and for this thesis, reliability can be rephrased as safety: that is, the quality a software has when it operates under normal operational circumstances according to what it is supposed to do.

With the rapid increase in processing power witnessed since the 60s, developping reliable software became a necessity for the field [21c]. Thus the science of software safety emerged. This science led to the birth of different techniques to spot unwanted behaviours during programming: among them are program debugging and best practices, type theory and its application on programming, programming and coding style. Test procedures to assert the correctness of the program on given inputs were developed, where different inputs are presented to a software to ensure that the actual answer matches the expected one. Bad memory management is a common source of errors, which led to the development of programming languages with embedded memory safety, such as Java and Rust. The life cycle of software was formalized to help its industrialization.

Among numerous existing techniques, we will focus on a specific family: *formal methods*. Formal methods are a scientific and technical field aiming to design techniques bringing strong mathematical guarantees on the safety of programs. The reliability element here comes from the sound mathematical reasoning those tools implement. Each step of formal verification is backed up by mathematical logic that derives from mathematical principles, with explicit definitions and reasoning rules. This soundness brings an additional layer of reliability by a program that is verified with formal methods. Applying formal methods to critical industrial software was met with successes. For instance the Paris subway lines 1 and 14 are fully automated; the correct behaviour of their software was proven using the Method B and Atelier B [Cle], a kind of formal methods; other components of subway lines can be proven correct as well [LSP07]. We call the process of verifying software using formal methods the process of *formal verification*.

Recently, a new kind of software was discovered: deep learning programs. Deep learning is a software development technique that propose another way to design programs. The core idea is that a deep learning program is able to perceive data in the human perceptual space (images, sounds, videos) and update its behaviour according to the result of said perception. One of the earliest goals of computer science is to mimic the human ability to process inputs from its physical environment, elaborate goals and formulating strategies to meet those goals regarding the environment state. Crucially, one strong objective is to develop the ability to adapt to changes in the environment, for instance being able to react to feedback provided by the environment

and consequently adapt the output, through some sort of learning process. Perceving, planning, adapting and learning are considered key elements to the development of intelligent processes, hence the wording "artificial intelligence" to call the field of computer science aiming at those goals. Deep learning helped to make huge progresses toward those overarching research objectives.

The perception and adaptation abilities of deep learning programs led us to build applications that were (theoretically) able to process the same "things" we process, and "adapt" to the feedback the humans gave to software. Here are some examples of softwares enabled with deep learning:

1. recommendation systems for entertainment, used for instance by Spotify (music streaming platform), YouTube (video sharing platform) and Netflix (movies and shows streaming platform)

2. predictive maintenance for asserting the optimal moment to perform maintenance on industrial settings [Tab+20]

3. medical assistance for screening radios, for instance against breast cancer [She+19]

4. automatic translations (this manuscript's readability and correctness really benefited from those) such as DeepL `https://www.deepl.com/translator`

5. art (see for instance the work of Alexey Popov and Tatiana Zobnina)

6. social surveillance software [Vin21]

As deep learning programs are software, we would require the same degree of trust to integrate them within human society. But the main quality of deep learning programs also constitutes its main flaw regarding reliability: since it processes inputs in the perceptual space, formulating specifications and applying existing tools is quite difficult. Another issue is that existing formal verification techniques are ill-adapted to perform on deep learning programs. Specifically, scalability is a crucial issue that still prevents from using existing techniques. We see some pervasive effects of those programs failures:

1. a semi-autonomous car accident occured because, according to the manufacturer's words, "[the car's] camera failed to recognize the white truck against a bright sky"[Haw19]

2. unexpected biases in recommendation systems (black people are more convicted than white people)

3. hacking of personal assistants based on voice [CW18]

Integrating further those programs within our societies while trusting them using formal methods is thus a difficult problem.

The aim of this thesis is to provide insights on how to tackle those issues in order to formally verify deep neural networks, in theory and in practice. Formal verification can be broken down into different pillars, or research interests:

1. specification: what do we formalize and how do we do it, what are the expected behaviours

2. techniques: what kind of mathematical reasoning will we use, and which guarantees it gives to us

3. heuristics: how do we take into account the specifics of the problem at hand to design and implement efficient verification algorithms

The work presented in this thesis studied those three pillars: we aimed to find a way to adapt them to deep learning software.

Outline

The first part of the thesis, part I, is made of two introductory chapters that are necessary to read in order to define our object of study. In chapter 3, we will define what software is, and how its behaviour can be checked using formal verification techniques. We will detail some of the formal methods we will use during this thesis. Chapter 4 will focus on deep learning software, and describe its specifics. It will also describe the current barriers that prevent us directly apply existing formal verification techniques.

The second part, part II, will focus on the difficulties to specify deep learning methods and our contributions regarding those. Chapter 5 will present our first contribution: CAMUS, a framework to formulate properties on deep neural networks on perceptive spaces. Chapter 6 will present the artifacts we developed to allow an easier formulation of said properties for neural networks.

The third part,part III, will focus on the combinatorial problem of deep learning verification. Chapter 7 will present an algorithm we proposed to improve the problem formulation of software verification. Chapter 8 will present an evaluation of this algorithm with an implementation, as well as further analysis on the problem formulation.

Finally, the last part, part IV) will present the future works and perspectives this thesis shed into light. A figure illustrating the outline of the thesis is available figure 2.1.

How to read this thesis

This thesis can be read linearly. Some chapters are tightly linked to each other while others are more independent. Chapters 3 and 4 are the introduction chapters and

should be read to understand the aim and scope of this thesis. Chapter 6 introduces a tool that is motivated by chapter 5. Finally, chapters 7 and 8 should be read together.

Each chapter has some reading helpers. We will provide summaries at the beginning of each chapter. They are written as compact descriptions of the chapter's aims, and show core results. A time-constrained reader can read each summary and hopefully grab a first understanding of this thesis.

Summary of the chapter

Summaries are enclosed within an orange box.

Following the different steps of a scientific development can sometimes be difficult: keeping up with notations and new concepts can quickly become overwhelming, and hazardous writing may fail to channel the attention on the key points of the reasoning. To try to address this issue, we will emphasize the key points of a chapter. Reading those alone will not be enough to fully understand the thesis, as they are context dependent. They are more to be seen as "anchors of focus", that are fully necessary to understand the rest of the chapter.

Key point

Keypoints are enclosed within a blue box.

Research questions are the topics we aim to research and (hopefully) provide an answer for.

Research questions

Research questions are enclosed within a bright green box.

Although chapters are focusing on a specific topic, some adjacent notions may interest the curious reader. At the end of some chapters, we will provide complementary references that may broaden the reader's views, but are not necessary to directly understand the chapter's core message.

Due to the high number of references in this thesis, we recommend reading it digitally using a pdf reader capable of link previewing, such as *evince* (Version 3.37.3 and above) on most of Linux distributions, *Adobe Acrobat Reader* (Version 2017 and above) on Windows.

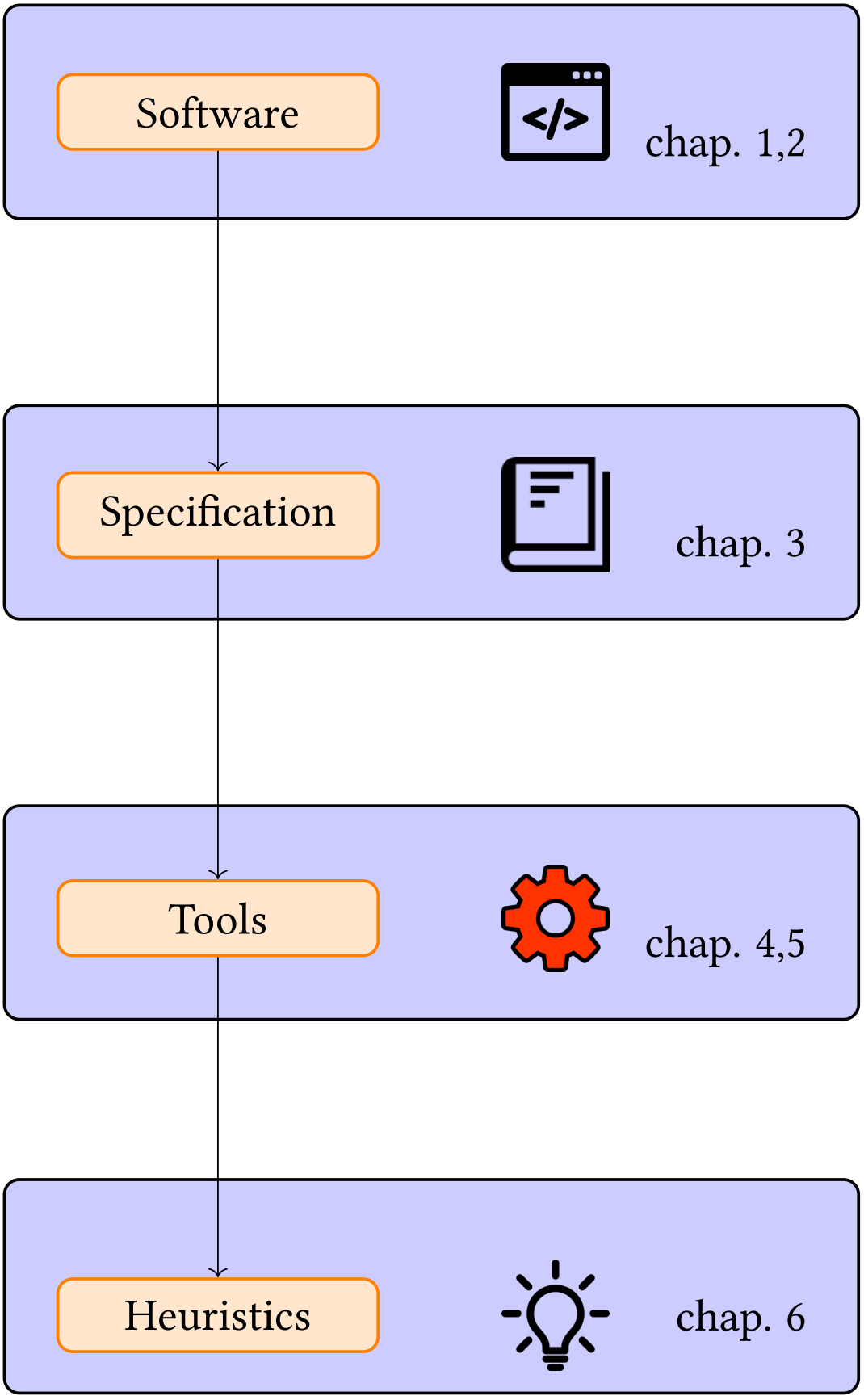


Figure 2.1: Graph representing the progression of this thesis with regard to the pillars of formal verification. Icons by Freepix on Flaticon.com

Chapter 3

Formal software verification

Summary of the chapter

We define software as a series of instructions written by humans or others to perform actions on the real world. Possible failure of software requires methods to ensure that its behaviour will not result in human harm.
We present formal verification, a field of research aiming to develop techniques that answer this goal.
We present some of those techniques:

1. Boolean Satisfaction (SAT) and Satisfaction Modulo Theory (SMT) calculi, that translate programs and properties as logical formulae to prove
2. Abstract Interpretation, that transforms program into another, abstract program, easier to verify
3. Symbolic Execution, that propagates symbolic variables within a program control flow, allowing to examine the possible paths and outputs taken by certain sets of inputs
4. Linear Programming (LP), that solves linear optimization problems

Providing methods bringing trust to software is an important goal: trust is an important component of social acceptance, and social acceptance is desirable if we want to further integrate software in the functioning of our societies. It is all the more true for software based systems whose failure may cause physical harm, ecological or economic loss. Such systems are called *critical systems*. Example of critical systems are power plants controllers or autonomous transportation software.

To enable trust on the technical side, there exist different techniques. Testing checks the behaviour of the program on a pre-determined single inputs, which helps

to detect the presence of bugs. Debuggers help the programmer to precisely monitor what happens during the program execution, leading to a better understanding of its inner working. Writting programs with certain languages can avoid entire classes of malfunctions.

The technique we will focus on during this thesis are those we call *formal methods*, a set of techniques and tools whose operation is grounded on different types of mathematical theories. Those techniques are used to analyse certain behaviours of programs. The practical results of those analyses are guaranteed with sound theoretical fundations, thus leading to an increased level of trust.

Those methods usually do not come for free, however. First, it is necessary to formalize the goal of the analysis (*what is the behaviour we want to verify*) in a clear, non-ambiguous way. Second, it is sometimes necessary to modify the program to make the verification process feasible, and to decide which parts of the program should be verified (*what is the subject of verification*). Third, a careful choice and parametrization of the verification tools is often necessary to obtain sound results on a set time budget (*how is the verification going to take place*). This chapter aims to give a sufficient understanding of formal methods through those points.

3.1 Representing software

But what is a software anyway? In the remainder of this thesis, we will use the following broad definition as a basis for our reasoning:

Definition 1. A *software* is an ensemble of instructions designed and written by a certain agent to accomplish a pre-defined objective. The instructions are written in a standardized way, so that a computer can understand those instructions. Said standardized way is called a *programming language*. The act for a program to be run to accomplish its purpose is called an *execution*.

Remark 2. The words *software* and *program* can be used more or less interchangeably; the latter has a slightly more technical meaning but for the scope of this thesis, the two words can be understood as meaning the same thing.

In definition 1, software designates both the file containing the sequence of instruction and the actual artifact ready to be used. To go from the former to the latter, the program is transformed through multiple steps. Each of those steps provides ways to represent the program differently. The representation we will use for most of this thesis is called the *control-flow graph*.

Definition 2. A control-flow graph (CFG) is a graph G with vertices V and edges E (denoted $G(V, E)$ in the following) describing all the states a program can take during

its execution. Vertices V represent memory states of the program, edges E represent instructions (jumps, conditional, loops) that transform the memory.

Let us illustrate this definition with an example: A CFG has an entry node (the

```
1 let v = read_int ()
2 let x = v
3 if x%2==0{
4       y=1
5 }
6 else{
7       y=0
8 }
```

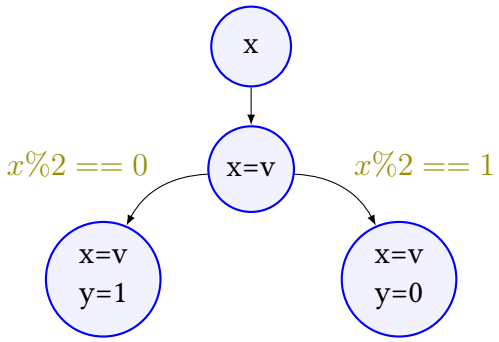


Figure 3.1: On the left: instructions for a dummy program. On the right, a simplified representation of this program using a CFG.

starting point of the program) and exit nodes (possible ways for the program to exit). This representation is helpful because it leverages graph theory tools to deduce properties on programs. For instance, a sub-graph not connected to the entry point denotes behaviour that will never happen under execution, "dead code" that can safely be removed. If all exit points are unreachable, this indicates an infinite loop. CFG will thus be our main way to represent programs (apart from source code snippets) for the rest of this thesis. Note that while CFGs are used to analyse and transform programs, we will mostly use them for their representation purpose.

Software is a human creation, and is involved in multiple human activities. Commerce, industry and science thus looked for ways to rationalize program design and implementation, and articulate human workforce to produce software of better quality. This search is obviously still ongoing, and one may see this thesis as a contribution to this effort. A lot of work exist on the organization of software production, which is beyond the scope of this thesis. A relevant aspect, however, is that software is a complex engineering artifact, that requires communication between multiple actors with different goals. A simplified vision on how software is made can be seen in figure 3.2.

Armed with a basic understanding of what is a program, we may now ask ourselves the following questions. Can a program fail? If so, what are the consequences? How can we spot this failure, and how can we prevent it?

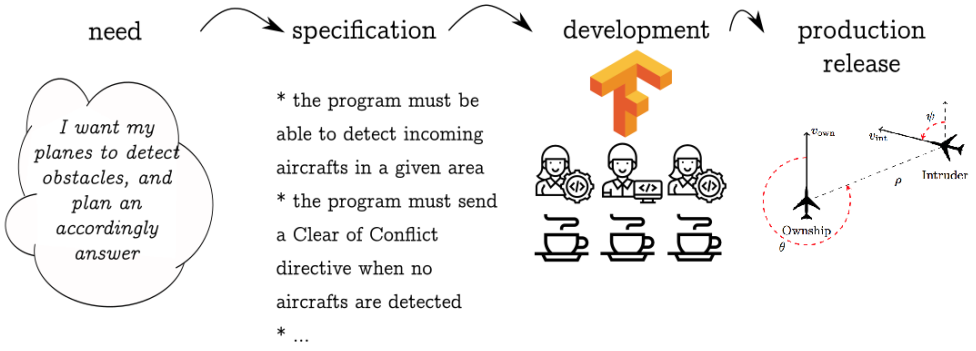


Figure 3.2: A possible way to design software

3.2 Why would we want to verify software?

As said in the introduction, software is taking part in our daily lives, and society deployed programs and adapted around them, to the point that failure in their operations can have dire consequences. The Colonial Oil pipeline [21a] and the Rouen hospital [Tri19] were both hit by a ransomware, a type of cyberattack. The pipeline was unable to operate during the attack, and the hospital had to rely on triaging patients without the help of computers, complexifying drastically the work of healthcare workers. A city water sanitization stations [Sta21] was left open to tamper with: a malicious user could have induced a mass scale water poisoning. Hospital, pipelines, water sanitization systems: all are examples of complex systems whose functions are partly exercised by software. It is then a legitimate concern for members of society to ask for guarantees to trust software to function according to its purpose. To plan a trip, a reliable scheduling system is expected. Running a hospital smoothly requires a power grid that functions all the time. And passengers of an aircraft expect its guiding and sensing systems to work sufficiently well so that they don't die during the trip because of a malfunction.

Software is a human creation, and thus is subject to failure. There is a high probability that the reader may already have experienced a program malfunctioning, with more or less frustrating results: unexpected failures while processing an administrative chore online, or the crash of a text formatter or spreadsheet editor with no backups…Failure of a train interlocking system [SCA15] would certainly have direr consequences. Because of errors made by the programmers and/or unexpected behaviours of the program, malfunctions are bound to happen if nothing is done to prevent them.

Of course, techniques helping to assess the quality of software predate the 21st century. We can broadly divide them in the following categories:

1. organizational methods: code reviews, separation between developing and testing, coding conventions

2. safety-by-design: using tools that increase the confidence we put into software, such as memory-safe programming languages, proof of underlying algorithms
3. runtime monitoring
4. quality assessment: after the software is made, technical certification can be achieved using tests or formal verification

This thesis mostly focuses on the last item: once the program is written and ready to run, how can one verify its quality? More specifically, we are interested in a certain property, called *safety*. There exist multiple formulations of software safety in the literature, see for instance ISO 26262 and ISO 21448 SOTIF for safety in autonomous vehicles. We propose the following definition that we think is enough for the scope of our work:

Definition 3. The *safety* of a software represents its ability to perform its purpose under normal operating conditions and avoiding malfunctions.

A key point here is that *normal operating conditions* and *malfunctions* need both to be characterized with a certain degree of precision, in order for technical tools to operate. For the dummy program presented on figure 3.1, one could want to verify that the program always returns an integer value. On more complex programs, one could formulate expectations on couple of inputs and outputs.

Remark 3. In definition 3, normal operating conditions are defined during the specification phase of the program conception. However, the malfunctions we illustrated at the beginning of this section are the result of an intentional action. Protecting software against intentional degradation falls into the field of *security*. In classical software, there exist a wide range of vulnerabilities, requiring to model a *threat model* describing what an attacker is able to do, leading to a wide range of different techniques. Fault injection and side-channel attacks are both attacks studied in security, and protecting against those require different approaches. Furthermore, protecting against those attacks is not the same thing as enforcing the absence of crash under normal operating condition. However, the programs we study in the rest of our thesis have special features (that we will define later on). Those features, that can make program unsafe, can be leveraged to perform intentional attacks. Symmetrically, protecting against an attacker and ensuring safety calls for the same approaches. Thus, this class of program blurs the line between “pure” safety and “pure” security. When the potential malfunction of the program is human-induced, we will classify it under “security” threat. When the potential malfunction can occur within normal operating range, we will classify it under “safety”.

Formal methods are a possible way to assess the safety of software, and the main subject of interest of this thesis. Under this wording lives a menagerie of research

fields and tools deployed on industrial settings; norms like DO-178C for avionics or ISO/IEC 15408 require their use in some critical systems. Drawing a history of such a wide variety is a complex work, and will necessarily be different between storytellers because of their teachers, their work and their affinities. The following is adapted from Julien Signoles's habilitation thesis [Sig18]. A possible starting point is to first ask ourselves why verifying if a program behaves accordingly to a specification is even a research problem. In his paper, Rice [Ric53] proves that there exist no exact, automatic static analysis procedure that can answer if a program has a non-trivial property. Any method wanting to verify non-trivial properties would need to go through a backdoor. Any existing family of formal methods can be understood as using a different backdoor to bypass Rice's theorem. The main component of all formal methods is a grounding in mathematical reasoning, which allows them to provide strong guarantees. We shall define in more details the specific methods that are the object of this thesis.

For this thesis, we will be dealing with analysis done after the program is completed. We will thus not work on certified code generation, nor typing systems (although we will be working with strongly typed languages).

But before diving further into the details of formal verification techniques, we will first define formally what it means to formally verify a program.

3.3 Formulating a verification problem

Let f be a program. Let $\mathcal{X}$ the space of all possible inputs, let $\mathcal{Y}$ be the space of all possible outputs. A verification problem postulates a *pre-condition* on the input space $\mathcal{P}(\mathcal{X})$ and a *post-condition* on the output space $\mathcal{Q}(\mathcal{Y})$.

Definition 4. A formal verification problem consists on verifying that $\forall x \in \mathcal{X}, \mathcal{P}(\mathcal{X}) \implies \mathcal{Q}(\mathcal{Y})$

The definition 4 does not make any hypothesis on the nature of the pre- and post-conditions. To take an example, let us imagine an array of numbers of size 16 bits, storing speed coordinates. A desired behaviour for this array is that, across execution of the program, it never overflows. Here, $\mathcal{P}(\mathcal{X})$ can engulf all $\mathcal{X}$. $\mathcal{Q}(\mathcal{Y})$ could be all possible outputs, excluding overflows errors (this example is inspired from the real-world Ariane 5 crash, the full post-mortem report is available here [Lio96]).

3.4 Formal verification techniques

It is now time to dive further into the specifics of the formal methods we will be studying and using during this thesis. We first define two important notions, coming from logic:

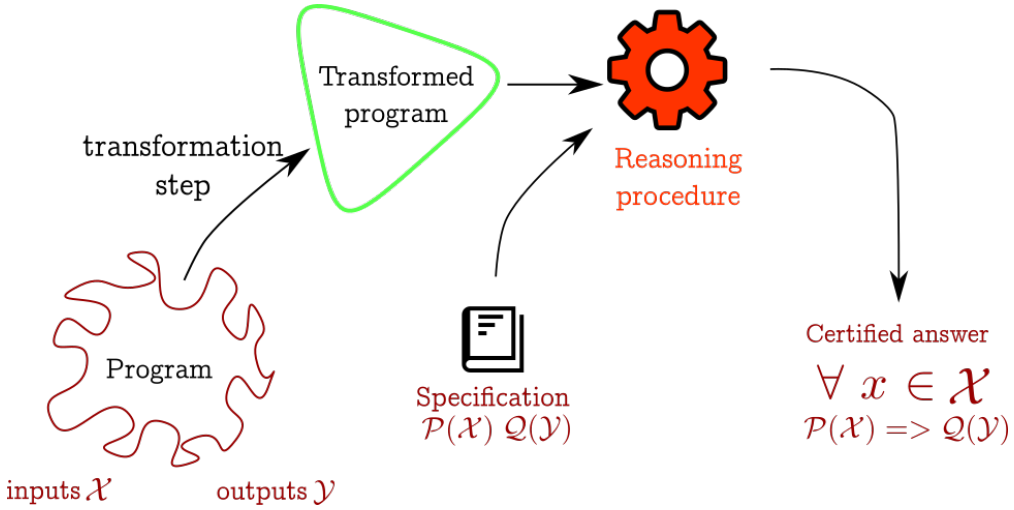


Figure 3.3: A schematic workflow of formal verification

Definition 5. A method is *sound* if it yields no untrue result. A method is *complete* if it processes all possible executions.

For instance, an algorithm that works only on a sub-part of the program will not be complete. An algorithm that can produce wrong answers will not be sound (hence, particular care must be taken during the implementation of verification algorithms).

We can broadly separate our techniques of study into two families:

1. exhaustive methods, sound and complete but more costly
2. overapproximative methods, sound but not complete, more scalable

Exhaustive methods: SAT and SMT calculus

Those methods are sound and complete, and usually depends on an exhaustive exploration of all possible situations, using search and simplification heuristics. The following explanation owes much to Gérard Berry's course at Collège de France [Ber15].

Problems to verify can be encoded and solved using Boolean calculus. Boolean calculus defines variables (also called atoms) with only two possible values: false or true. Available operators are the logical AND, or conjunction ($\wedge$), and the logical OR, or disjunction ($\vee$). The following rules are given:

- associativity: $A \wedge (B \wedge C) = (A \wedge B) \wedge C$
- commutativity: $A \wedge B = B \wedge A$
- idempotency: $A \wedge A = A$

- neutral elements: $A \wedge true = A$, $A \vee false = A$
- absorbing elements: $A \wedge false = false$, $A \vee true = true$
- distributivity
- negation $\neg$: $\neg true = false$, $\neg false = true$
- Morgan's laws:
  1. $\neg(A \wedge B) = (\neg A) \vee (\neg B)$
  2. $\neg(A \vee B) = (\neg A) \wedge (\neg B)$

Those sets of rules can be used to encode logical formulae, which are a way to express pre and postconditions.

Definition 6. A *clause* is a disjunction of literals. The *Cunjonctive Normal Form (CNF)* is a formula expressed as a conjunction of clauses.

The SAT problem is thus defined as follows:

Definition 7. A formula $F(x_1, x_2, \ldots, x_n)$ is satisfiable (SAT) if there are boolean values x_i making F true. A formula $F(x_1, x_2, \ldots, x_n)$ is valid if it is true for all x_i.

The main goal of SAT as a research domain is to find efficient ways to solve a SAT problem, that is to say, answer if a formula is SAT or not. Solving a SAT problem is a NP-hard problem according to the Cook-Levin theorem [Coo71]. In practice, since the first SAT solvers, multiple improvements were developed that allowed to solve most of real-world problems. The Davis-Putman-Logemann-Loveland algorithm [DLL62] is the backbone of most solving procedure, guaranteeing a sound and complete exploration of all formula instantiation by fixing the value of one variable at a time. Conflict-Driven Clause Learning [Mar+96] consists on learning from the failure. Two-watched literals [Mos+01] greatly reduced the algorithmic overhead of keeping track of the backtracking points and formula states. There are still open research questions, among which is parallelization; the community is dynamic, with an annual competition to push up new ideas: SAT-COMP `http://satcompetition.org/`.

Remark 4. Note that verifying if a formula is valid is equivalent to verifying that its negation is UNSAT: $F(x) = \text{VALID} \equiv \neg F(x) = \text{UNSAT}$

One of the limitations of SAT solvers is that they only compute booleans formulae. This limits the expressiveness of the encoded formulae and requiring greater expertise to encode a real world problem into a SAT formula. Let us take an example. Suppose one would like to solve the following problem:

$$((a = 1) \vee (a = 2)) \wedge (a \geq 3 \wedge ((b \leq 2) \vee (b \geq 3)))$$

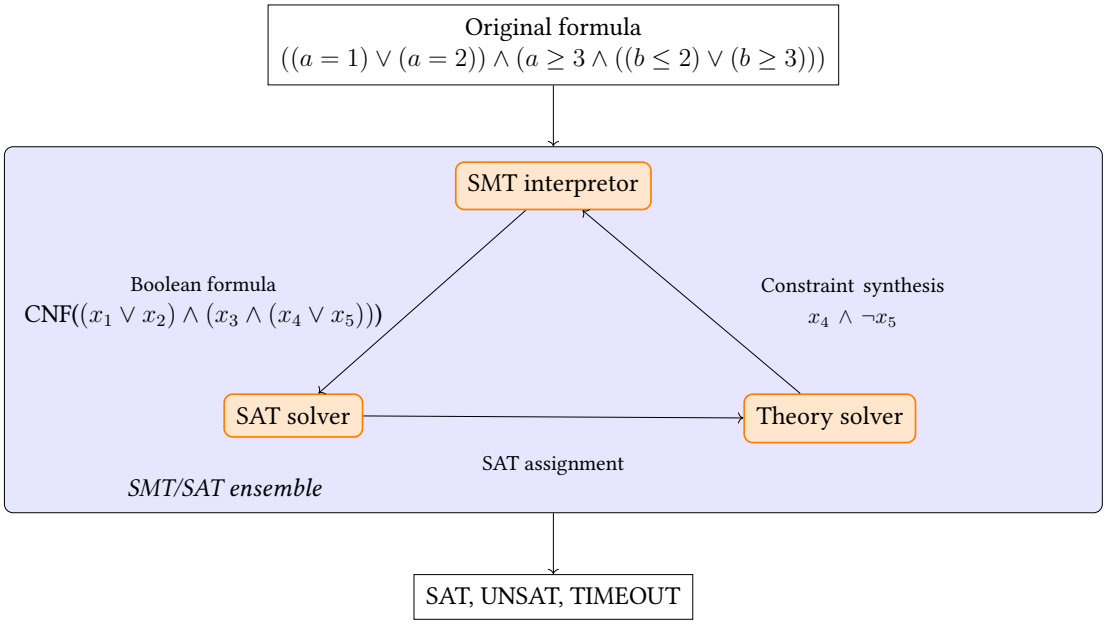


Figure 3.4: simplified workflow of SMT calculus

To be able to do this, one would need to identify a, b, 2 and 3 as belonging to the same kind of things (real-valued numbers), specifying the meaning of the arithmetic sum and of the inequality, and finally a way to solve the equation.

The Satisfaction Modulo Theory (SMT) calculus aims to answer this issue by providing a new conceptual tool: *theories*.

Definition 8. A *theory* is a set of symbols and rules specifying a semantic and a grammar for given symbols

Theories aim to translate a formula written in a certain formalism to SAT problems. For our example, it is possible to define SAT variables, boolean valued: $x_1 : a = 1, x_2 : a = 2, x_3 : a \geq 3, x_4 : b \leq 2, x_5 : b \geq 3$. The initial formula thus become: $(x_1 \vee x_2) \wedge (x_3 \wedge (x_4 \vee x_5))$, which then can be transformed into CNF and send to a SAT solver. The SAT solver generates a SAT assignment for the variables. This assignment is sent back to a Theory solver, that checks if the SAT assignment is valid in the target theory. Here, a possible assignment is $true$ for all variables. But according to the integer arithmetic, $b \leq 2$ and $b \geq 3$ is not possible simultaneously. The SAT assignment is then not valid in the target theory, so new constraints are generated: $x_4 \wedge \neg x_5$. This new constraint is added to the original problem, and the cycle continues until a definitive answer is given (or the solver times out). This overall process is presented figure 3.4.

Modern tools implementing SMT calculus notably include Z3 [dMB08] and CVC4 [Bar+11]. Like SAT, there exist an annual competition that provides common benchmarks, SMT-COMP `https://smt-comp.github.io/2021/`.

Overapproximation: abstract interpretation

Contrary to exhaustive methods, overapproximation methods are not complete, because they usually compute a version of the program that has been transformed to behave differently. However, they still have a soundness guarantee; they also tend to be faster than exhaustive methods.

Another way to circumvent Rice's theorem is to relax the exact nature of the verification procedure. This is the core of abstract interpretation, introduced by Cousot and Cousot in [CC77]. The seminal idea is to compute an overapproximation (or abstraction) of the program behaviour. Verification is then made on the result of the abstract program computation. Usually, this is made by modifying the program to compute sets of inputs (or *domains*), that describe certain behaviours of the input. An example with intervals, a simple domain, is available figure 3.5. Note the important limitation of intervals: given a value $x := [-1, 1]$, we have $x - x = [-2, 2]$ according to interval arithmetic, where it should be equal to 0. Intervals propagate imprecision each time an operation is applied to them. More accurate numerical domains exist, such as zonotopes [GGP09], but using them is more costly. The key of abstract interpretation is to find the good trade-off between precise analysis and reasonable execution times.

```
1 // x ∈ [−1, 1]
2 // y ∈ [2, 4]
3 let z = x + y
4 if z < 0 then
5   costly_function
6
7 else
8   simple_function
```

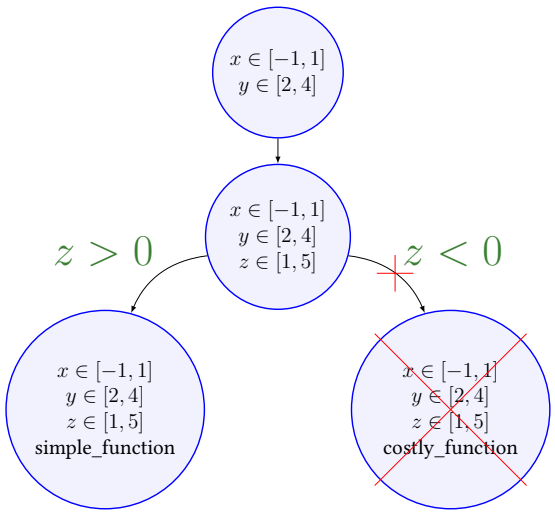


Figure 3.5: Using interval arithmetic, we are able to rule out the costly computation within the conditional.

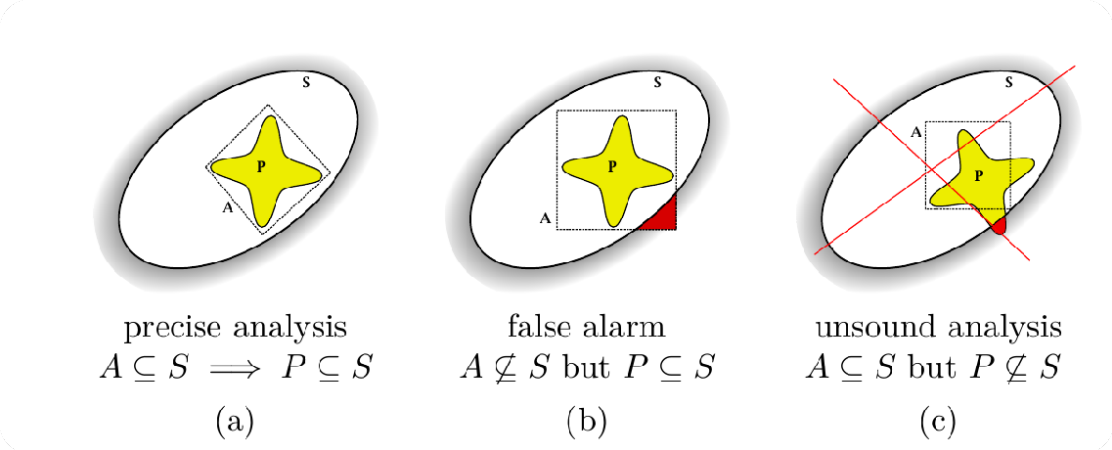


Figure 3.6: Stakes of abstract interpretation. On fig. (a), the abstraction A correctly encapsulates the behaviour of the program. On fig (b), A is too wide, and the verification procedure raises a false alarm. On fig. (c), A does not encompass all behaviours, resulting on an unsound analysis. Domains used in abstract interpretation must be sound, so case (c) is not supposed to happen when dealing with real tools. Credit: Antoine Miné

Symbolic execution

The key to symbolic execution is to make use of *symbolic* values instead of *concrete* values. The program control flow is altered to propagate symbolic variables, and the whole execution is understood as a function of symbolic inputs. Symbolic execution creates a mapping m between a variable v and a symbolic value e: this mapping can be seen as equality constraint. At each execution step of the program, m is refreshed to add new additional symbolic variables v or change the symbolic values mapped to v, according to the semantic of the program. When a branching point - often, a conditional c that involve multiple v - is met, c is morphed into a constraint translating the satisfaction of the conditional. It is then added to m: $m = m \wedge c$. In parallel, a new mapping is created: $m^{'} = m \wedge \neg c$, corresponding to the other side of the conditional. The two symbolic execution paths continue their execution independantly.

When an end point is reached, a constraint solver (often an SMT solver) solves the set of constraints formed by m and instantiate the symbolic values with concrete input values. When inputted in the original program, those concrete input values will result in the exact same execution path being taken. For instance, in the program described in figure 3.7, at line 3, we have variables x and y mapped to symbolic values without any constraints x_0 and y_0. Then, when we enter the procedure, at line 7, symbolic variable z is mapped to symbolic value $2 * x_0$. A branching point occurs at line 8. A first constraint, $y! = 2 * x$, is added to a copy of m (corresponding to the first left node on the figure). Since the procedure ends if this conditional is not verified, then the symbolic execution ends and the solver returns concrete values for x_0 and y_0

that satisfy the constraints added to m. A similar procedure occurs for the rest of the execution.

Symbolic execution is susceptible to limitations such as path explosions, when the number of branching is too high, or potential infinite execution. For programs with no loops however, this technique, mostly used for testing, can provide crucial information to other formal verification techniques, or be used in a stand-alone fashion.

```
1  let main =
2      x = symbolic_val
3      y = symbolic_val
4      procedure x y
5
6  let procedure x y =
7      let z = 2 * x
8      if z == y{
9          if y > x+10{
10             fail()
11         }
12     }
```

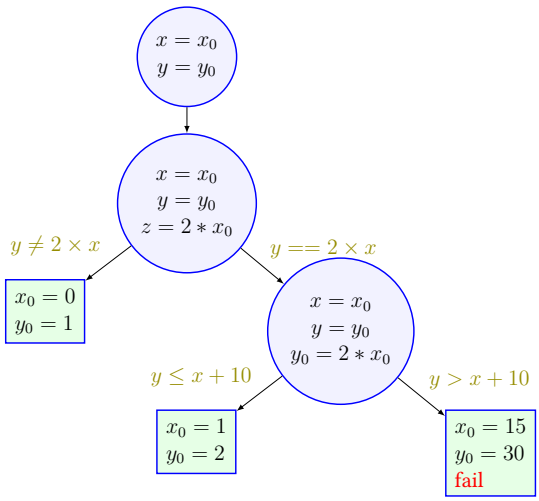


Figure 3.7: Example adapted from [CS13]. Round nodes describe the memory state of the program, rectangle nodes describe a possible concrete input leading to this execution path.

Linear optimization

Finally, let us introduce another technique that we will be using during this thesis: *linear optimization* (also called Linear Programming, or LP). The overall goal of linear optimization is to find the optimal solution for a linear function over a certain set of linear constraints. This technique, mostly used in applied mathematics, can also be used for software verification, providing the program can be encoded under a set of linear functions. Geometrically, this can be seen as finding the minimum value inside a convex polyhedron.

More formally:

Definition 9. For $x \in \mathbb{R}^n$ the real valued variables, $c \in \mathbb{R}^n$ the *objective function*, $A \in \mathbb{R}^{m,n}$ and $b \in \mathbb{R}^m$ the *linear constraints*, a *linear optimization problem* is the minimization problem

$$\min_{x, Ax \leq b} c^T x$$

Such problem can be solved using various algorithms. One of the most popular is the simplex algorithm, described for instance in [21b]. The basic idea is to navigate alongside the vertices of the constraints' polyhedron in order to look for conflicts between constraints. An optimal solution is found when no constraints are violated. Common solvers used to solve LP problems are Gurobi[Gur21] and GLPK[Mak].

To go further The interested reader can look at[CS13] for a pedagogic explanation of symbolic execution and its main challenges. Regarding abstract interpretation, a good starting point is the Antoine Miné tutorial [Min17]. The author of this thesis would encourage any interested reader on the topic of type systems to take the course of Xavier Leroy's [Ler19] on the developments of this idea on the field of mechanized verification.

Chapter 4

Programs that learn

Summary of the chapter

We present artificial intelligence, a field of computer science aiming to replicate the human abilities of perception, reasoning and decision-making in the material world. We focus on neural networks, a specific kind of program. Neural networks are able to process inputs of high dimension and detect patterns in them, thanks to the gradient descent algorithm, research in parallel computing and availability of massive labeled datasets. Neural networks are programs and as such, can fail. Small variations on their inputs can result on vastly different results, their control flow is not explicitly programmed and the data they manipulate is difficult to specify. Data as specification prevent the formulation of properties. Their non-linearity makes it difficult to apply classical formal verification, leading to combinatorial explosion.

In the previous chapter, we defined the overall process of formally verifying software. A program – represented by a control flow graph – is transformed and analyzed using several techniques for a specification to be checked against. So far, no specifics were given on what kind of software we will study. The aim of this thesis is to study the possibility to formally verify a certain class of software: *machine learning programs*.

4.1 A short history of data-oriented programs

If the words *machine learning* and *deep learning* are now commonplace in the everyday discourses, they refer to programming paradigms that are ancient – relative to computer science. The research field under which those techniques were developed is called *Artificial Intelligence* (AI). Since its birth however, AI has become much more than a research field. Humans imagined thinking and acting artifacts long before the

dawn of the digital computer. Greek inventor Ktesibios compiled and developed the principles of tamed mechanical force in numerous of his inventions, such as the clepsydra. The Jewish mythos of the Golem describe a humanoid figure made of clay and answering the wills of its creator. In Fritz Lang's *Metropolis*, an automaton is created to stem an unsuccessful revolt on a poor working class. Our imagination is thus populated with projection of artificial artifacts. The increasing integration of software in human life and society led new issues to be tackled by politics. In this context, we think it is important to define what is artificial intelligence (and its offspring, deep learning research) as a research field; and to produce a short history of this field to give some perspective to the reader.

A tentative definition of artificial intelligence

While going through the literature, we found that there was no unique definition for artificial intelligence. Rather, there are partial definitions: some focus on describing the tasks performed by a hypothetical artifact, others focus more on the processes used to build and run those artifacts. We propose here three of those partial definitions:

Definition 10. Artificial intelligence is:

1. "the study of the computations that make it possible to perceive, reason and act." [Win92]
2. "an agent that acts as to achieve the best outcome or, when there is uncertainty, the best expected outcome." [RND10]
3. "the ability of a digital computer [...] to perform tasks commonly associated with intelligent beings. The term is frequently applied to the project of developing systems endowed with the intellectual processes characteristic of humans, such as the ability to reason, discover meaning, generalize, or learn from experience." [Cop]

Those definitions present the three main objectives of artificial intelligence, namely, the characteristics an artificial intelligence program should aim to:

1. *perceive* information from the material space;
2. *create knowledge and meaning* from the perceived inputs;
3. *act rationaly* according to its knowledge, sometimes on the physical world.

As an example, on a rainy day, such programs could identify that the sky's colour is changing from blue to grey, conclude that some rain is about to pour and that a coating

of some sort would be useful to not be drenched by the rain, and proceed to look for the nearest coat.

Those three attributes led to different research fields: allowing a machine to perceive led to computer vision and audio processing; building knowledge and reasoning procedures led to knowledge representation and computational logics such as SAT calculus (presented in previous chapter) and expert systems; acting on the physical world is the topic of cybernetics, while designing and efficiently implementing an action policy borrowed from mathematical optimization. The idea of studying reasoning (or at least self-regulating) and acting artifacts can be tracked back to the beginning of the 18th century, with Watt's steam engine, or the formalization of the study of feedback systems as a discipline (automation) in the 19th. But those attributes were first put together and formalized by Alan Turing in his seminal paper [Tur50], where he also present his "imitation game" (later known as the Turing test): a set of tests that a computer would need to succeed in order to *seem* intelligent for a human observer. AI as a discipline began to flourish, spawning multiple approaches that led multiple encouraging early results. One such approach focused on developing a certain kind of programs: neural networks.

Formal neuron and perceptron (1943–1969)

The first description of a formal neuron is found in [MP43]. This model originally stems from the biological neuron, even if computer science neurons do not have much in common with biological ones. In this paper, the authors describe a computational unit (the neuron) outputting a value once a certain threshold is met, using an activation function. The *perceptron* algorithm[Ros58] describes how formal neurons could

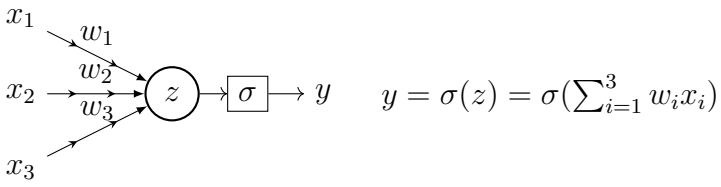


Figure 4.1: A formal neuron. σ is an activation function that introduce non-linearity in the neuron's behaviour (in the perceptron, σ was the Heaviside function)

reorganize their initial connections to achieve a given objective. See figure 4.1 for a description of a formal neuron. Theoretical limitations were exhibited: in [MP72], it was shown that a non-linear separable problem (such as the learning of the logical XOR) could not be represented with a linear perceptron. This was due to the single-layered nature of the perceptron, as well as the activation function used.

Another issue arose quickly. If programs of that time were able to provide impressive results on toy problems with small dimensionality (3 inputs and 3 outputs for

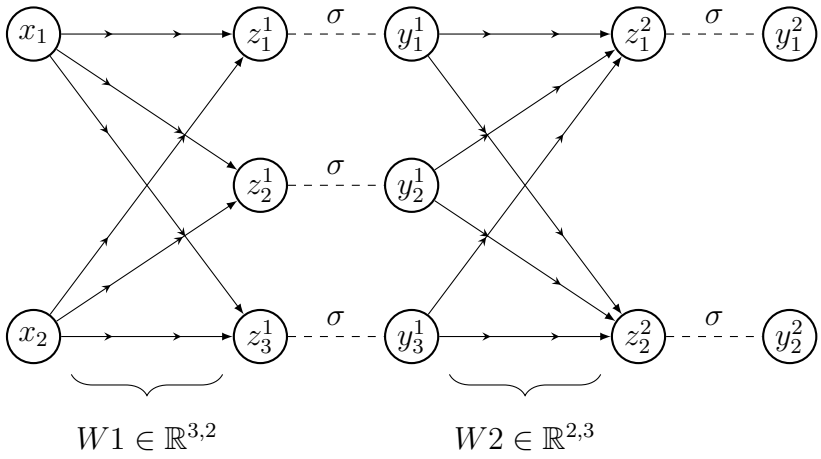


Figure 4.2: A neural network with two inputs, one hidden layer of three neurons and two outputs, no biases. Each neuron process the weighted sum of its inputs, followed by an activation function

an ADALINE[WH60]), results in other discipline showed that those programs did not scale up to larger problems. In [Mit21], the authors advance the possible cause that the combinatorial explosion (sometimes called "curse of the dimensionality"), was not foreseen by researchers and industries at the time, leading to overoptimism. We can note that difficulties to assess the hardness of an AI problem can be seen nowadays with autonomous driving, with industrials that fail to deliver (see the multiple declaration of Tesla's CEO Elon Musk [Owe21], or this video of a Waymo car being stuck by signaletic cones [JJR21]). The lack of expressiveness of perceptrons and the inability to tackle high-dimensional problems led the field of neural networks to go dormant for some time.

Backpropagation (1975–1986)

Presenting one single event as leading the rebirth of interest in neural networks would not be wise. In twenty years, the landscape of computer science changed a lot. The specifically-designed device implementing the ADALINE could be superseded by general-purpose programming languages like C. Computers were multiple order of magnitude faster than they were during the 60s, and they were now much more affordable. Computer science as a field grew more mature, and the design of software shifted from an engineering problem to a scientific process. Each field has its tradition however, and the tradition of deep learning considered the rediscovery of the backpropagation algorithm by several teams in a short period of time as the "rebirth" of neural networks. The history of this (re)discovery is somewhat a topic of discord in the field: let us cite [RHW86] in 1986 and [Le 86] as the papers that sparked this new trend,

and an earlier work [Wer90] from 1975. The interested reader can consult [Sch] for more context on that matter. Nevertheless, this method of propagating error inside deep neural networks is still the backbone of modern tools, and will be described with more details later. Some early practical results are described in [Le +89], were a neural network was able to detect handwritten digits with 1% error rate. Distributed programming and the continuous increase of processing powers of computers allowed tackling high-dimensional problems that were left undone at the end of the 60s, as well as new ones.

Massive availability of data and the Boom of machine learning (1989–present)

A final ingredient was required before the broth was ready. Availability of massive, annotated datasets allowed to leverage the theoretical computational power into practical uses. The Mixed National Institute of Standards and Technology (MNIST) dataset[Li 12], first introduced in 1998, comprises 60000 28×28 grayscale images of handwritten digits. For teaching self-driving cars to drive, the nuScenes [Cae+20] datasets is a collection of annotated videos containing locations of various entities at each timestep. CommonVoice `https://commonvoice.mozilla.org/fr` is a project to create a dataset of voice samples representative of the whole human population. ImageNet[Den+09] is a collection of more than 14 million images and 21000 classes used for image classification in general, subject of a competition: ImageNet Large Scale Visual Recognition Challenge (ILSVRC).

All was set for the new neural network boom. In 2012, the authors of [KSH17] presented an implementation of a neural network trained to detect classes on the ImageNet dataset. They scored first on the ILSVRC challenge, pushing the state of the art of computer vision. This sparked a vast reinvestment of research and industry in this field, of which this thesis is a descendant. The field is now swarming, to the point that historic machine learning conferences are having trouble to cope with the increasing number of publications. An analysis of the submissions at one of the top machine learning conference, NeurIPS 2020, is available at [Iva20]. Some key points include:

1. almost triple the number of submissions from 2017 than to 2020;
2. a constant acceptance rate during those three years;
3. a high number of United States organizations in the top publishers (in particular, Alphabet's subsidiary, Google, is the top publisher);

Trained sociologists and data experts [Syn20] show that the gender imbalance in AI is increasing, on par with the similar phenomenon prevalent in computer science in general.

State-of-the-art is moving fast in neural network research. We can nevertheless point the reader to some discoveries that we consider important for different fields. Skip-connections [He+15] introduced an architectural block that eased the learning of very deep architectures (the original paper presented state-of-the-art accuracy on ImageNet with 50 layers). Generative adversarial networks [Cre+18] introduced a learning scheme allowing a network to generate data from a learned distribution, with applications from music composition to fake picture generation. The Transformer architecture [Vas+17] dramatically reduced the required resources needed to train programs working on natural language, leading to vast improvements in text applications. Object detection for autonomous or semi-autonomous vehicles [Boj+16] make use of neural network techniques. Less consensual applications include facial recognition and reidentification of people for public surveillance (see [Ye+21] for a survey on this technique). Clearview AI[1] is one of the numerous companies that use machine learning for person reidentification. Tech company Canon used facial recognition to detect smiles, only allowing smiling workers to enter offices [Vin21].

4.2 Background and notations

Now that our readers have some element of context, we shall dive into our subject and introduce the basic concepts necessary to understand how a neural network works.

Mathematical background

Let $\mathcal{X}$ be an input space. For RGB images, this typically is a subset of $\mathbb{R}^{3*d_{in}}$ where d_{in} is the number of pixels of the image. Let $\mathcal{Y}$ be an output space. For our problems, this is typically a vector of $\mathbb{R}^p$ where p is the number of different entities to classify.

Definition 11. Let W be a linear operation $\mathbb{R}^{d_{in}} \to \mathbb{R}^{d_{out}}$, let σ be an activation function $\mathbb{R} \to \mathbb{R}$ applied pointwise to each coefficient of a vector; meaning for $a, b, c \in \mathbb{R} \times \mathbb{R} \times \mathbb{R}$:

$$\sigma \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} \sigma(a) \\ \sigma(b) \\ \sigma(c) \end{bmatrix}$$

We name *layer* the function composition $\sigma \circ W$. A *neural network* is a function $f : \mathcal{X} \to \mathcal{Y}$ that is composed by a succession of layers. We note X_i the input vector of layer i, W_i the linear operation occuring at layer i, parametrized by the parameters θ_i (σ has no parameter). We note $Z_i = W_i(X_i)$, $Y_i = \sigma(Z_i)$. Note that $X_{i+1} = Y_i$.

[1] `https://clearview.ai/`

The gradient of a (possibly multidimensional) quantity x with respect to the (possibly multidimensional) variable y is denoted by $\nabla_y x$. The partial derivative of x with respect to y is denoted by $\frac{\partial x}{\partial y}$.

We will study networks after their training, meaning the weights will be fixed. In our thesis, the architecture will be kept relatively simple and similar: we will only use matrix multiplication and element-wise addition for the linear operations.

Definition 12. Let $A \in \mathbb{R}^{n,p}$, $B \in \mathbb{R}^{p,m}$ and $d \in \mathbb{R}^{n,m}$ be three real-valued matrices. The *matrix multiplication between A and B* is the operation producing the resulting matrix $C \in \mathbb{R}^{n,m}$ where coefficients $c_{i,j}$ at coordinate i, j are $c_{i,j} = \sum_{k=1}^{p} a_{i,k} b_{k,j}$.

The *element-wise addition between C and d* is the operation producing the resulting matrix $C^{'} \in \mathbb{R}^{n,m}$ where $c^{'}_{i,j} = c_{i,j} + d_{i,j}$. Similarly, element-wise multiplication can be defined.

A popular activation function is the REctified Linear Unit (ReLU for short):

$$\text{ReLU} : x \in \mathbb{R} \mapsto \max(x, 0) \tag{4.1}$$

Neural Nets with ReLU is a common design choice for most use cases. Furthermore, ReLU is a *piecewise-linear* function: it is linear on $]-\infty, 0]$ and on $[0, \infty[$, which makes formal analysis easier. Thus in this thesis we will limit our study to Neural Nets with ReLU as activation functions.

Task definition: the example of classification

Among the multiple tasks one can ask a neural network to do, we will study *classification*. Let us consider a sample x from a dataset $\mathcal{D}$. Each x is assigned a label $y_i, i \in (1..p)$. For instance, in a dataset of animals, images of dogs will be labelled as "dogs", while images of cats will be labelled as "cats". The aim of the neural network will be to output the probability for an unknown (*i.e.,*previously unseen) sample to belong to one of the p classes. To this end, the last layer will have a log-softmax activation function:

$$\text{LogSoftmax} : x_i \mapsto \log\left(\frac{\exp(x_i)}{\sum_j \exp(x_j)}\right) \tag{4.2}$$

This choice models the fact that neural networks output log-probabilities for their outputs to belong to a certain class.

Preparing a dataset

To perform sample classification, an annotated dataset is required. The annotation process is costly, and is nowadays mainly made manually, using microworking platforms such as Amazon Mechanical Turk. See the work of Paola Tubaro [Tub21] for the

social impact of that kind of activity. The prohibitive cost of collecting, normalizing and annotating a sufficient amount of data can be mitigated by using public datasets. Usually, a dataset $\mathcal{D}$ is divided between disjoints splits, usually a training split $\mathcal{D}_{train}$ and a testing $\mathcal{D}_{test}$, to ensure that the evaluation of the neural network is not on already seen samples. When no dataset of sufficient size is available, it is possible to use a technique called transfer learning [Tan+18] to fine-tune a pretrained program to the target task.

Ethics behind dataset

For neural networks in particular, the choice of a dataset is not without consequences. Datasets are constituted by humans, who are not devoid of biases. Racism and sexism biases do appear in the collection of data. Neural networks are not trained to correct thoses biases if not specifically taught to do so. Since such technologies are already used in law-enforcement or in security sectors, a special care should be taken into ensuring that those algorithms treat fairly all the population, instead of enforcing stereotypes (people may be eager to trust the result of an algorithm because it is somewhat "more neutral" than a human). Some example of potential unwanted biases can be found in the Correctional Offender Management Profiling for Alternative Sanctions (COMPAS) algorithm [Mat+16]. The debate is still strong to assess the exact nature of those biases, and the research field of algorithm fairness is focusing on identifying and mitigating those biases.

Architecture and engineering

To train a neural network, one must assemble basic building blocks in a certain order. The dimensionality and chaining of those building can vary, yielding different results. Finding the correct learning algorithm and loss function is also necessary. All those elements are usually called *hyperparameters*. Hyperparameter tuning is a craft of careful engineering, that usually require to train the same program several times with a variation of the hyperparameters. Automatic machine learning (AutoML) aims to automate the hyperparameter tuning by using deep learning: we witness networks that are "learning to learn".

The learning algorithm: gradient backpropagation

Let $\mathcal{L}$ be a loss function, designed to calculate the error between the output of the neural network and a given ground-truth. Common loss functions for classification tasks include binary cross entropy, while regression tasks consider most often mean square error. For one given sample, binary cross-entropy is a kind of distance between two probabilistic distributions:

1. the perfect target distribution p^* followed by the desired outputs, of the form $(0, 0, 1, 0, 0, ...)$;

2. the approximation $\hat{p}$ of this distribution, which is the output of our classifier:

$$-\frac{1}{N}\sum_{i=1}^{N} p_i^* * \log(\hat{p}_i) + (1 - p_i^*) * \log(1 - \hat{p}_i) \tag{4.3}$$

Let $\delta_i = \frac{\partial \mathcal{L}}{\partial Z_i}$ be the variation of the loss according to the pre-activation of layer i, and σ be the application of the activation function.

For the final layer L, an explicit expression exists for δ_L:

$$\begin{aligned}
\delta_L &= \frac{\partial \mathcal{L}}{\partial Z_L} && (4.4)\\
&= \frac{\partial \mathcal{L}}{\partial Y_L} * \frac{\partial Y_L}{\partial Z_L} && (4.5)\\
&= \frac{\partial \mathcal{L}}{\partial Y_L} * \sigma^{'}(Z_L) && (4.6)
\end{aligned}$$

Going from equation (4.4) to equation (4.5) is done by applying the gradient chaining rule. For previous layers:

$$\begin{aligned}
\delta_i &= \frac{\partial \mathcal{L}}{\partial Z_i} && (4.7)\\
&= \frac{\partial Z_{i+1}}{\partial Z_i} * \frac{\partial \mathcal{L}}{\partial Z_{i+1}} && (4.8)\\
&= \frac{\partial Z_{i+1}}{\partial Z_i} * \delta_{i+1} && (4.9)
\end{aligned}$$

We have

$$\begin{aligned}
Z_{i+1} &= W_{i+1}(X_{i+1}) && (4.10)\\
&= W_{i+1}(Y_i) && (4.11)\\
&= W_{i+1}(\sigma(Z_i)) && (4.12)\\
& && (4.13)
\end{aligned}$$

Thus, injecting equation (4.12) in equation (4.9), we have

$$\delta_i = W_{i+1}(\sigma^{'}(Z_i))\delta_{i+1} \tag{4.14}$$

We are thus able to compute the gradient of the error regarding the outputs of each layer. Finally, to update the parameters at each layer, we need the following gradient:

$$\frac{\partial \mathcal{L}}{\partial Wi} = \frac{\partial \mathcal{L}}{\partial Z_i} * \frac{\partial Z_i}{\partial W_i} \tag{4.15}$$

$$= \delta_i * \frac{\partial Z_i}{\partial W_i} \tag{4.16}$$

$$= \delta_i * \sigma(Z_{i-1}) \tag{4.17}$$

$$= \delta_i * Y_{i-1} \tag{4.18}$$

The backpropagation algorithm is then the following:

1. *Forward pass*: For each layer i, compute linear operations Z_i and activations Y_i.
2. *Output error*: compute $L(Z_L)$
3. *Backpropagation*: compute δ_l for each layer
4. *Weight update*: $W_i \leftarrow W_i - \delta_i * Y_{i-1}$

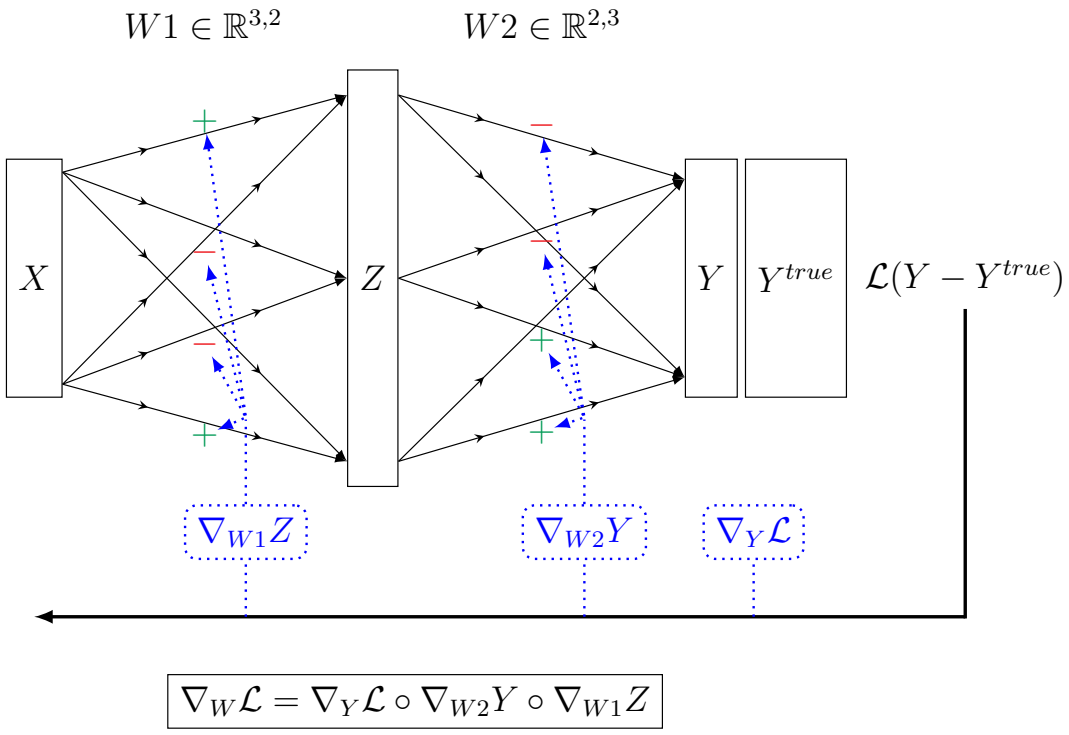


Figure 4.3: Gradient backpropagation algorithm

4.3 Vulnerabilities in machine learning

Algorithms described previously are implemented in practice within libraries that provide a variety of tools for the machine learning practitioner: most of the networks studied during this thesis were parametrized and trained using the PyTorch[Pas+19] library. TensorFlow[Aba+16a] is another popular choice. Those libraries being relatively complex, they are not exempt of bugs. However, the scope of this thesis is not to study library-dependent bugs. Across implementations, machine learning programs exhibit "odd" behaviours. The consistent reproduceability of those behaviours between implementation led the scientific community to consider their definition and mitigation a research problem. We now present some of those vulnerabilities.

Remark 5. *Bug* or *fault* are well-defined words in traditional software safety, but the literature on machine learning safety is not fixed yet on a common term. Another possible word is *vulnerabilities*, since this behaviour can be exploited by an external agent. We think the latter better convey the idea that the phenomena we describe are intrisinc weaknesses of neural networks in the way we currently train and use them, rather than something that can be pinpointed and fixed with an explicit process. *Frailty* will occasionally also be used.

Adversarial examples

Adversarial perturbations are small variations of an input that have been crafted so that the network misclassifies the noisy input, called an *adversarial example*. More formally, given an input $x_0 \in \mathcal{X}$, a network $f : \mathcal{X} \to \mathbb{R}^d$, a distortion amplitude $\varepsilon \geqslant 0$ and a distance metric $\|.\|_p$, a neural network is *locally ε-robust* if for all perturbations δ s.t. $\|\delta\|_p \leqslant \varepsilon$, $f(x_0) = f(x_0 + \delta)$. Figure 4.4 explains schematically the crafting of adversarial examples, and figure 4.5 an iconic adversarial example.

Adversarial examples were initially discovered in [Sze+13], where the authors maximized the error function of a neural network. Since then, the literature on adversarial examples bloomed. Multiple methods were proposed to generate adversarial samples, such as the Fast Gradient Sign Method (FGSM) in [GSS14], that relies on creating a perturbation in the direction of higher error gradient, and the Carlini method in [CW16], which aims to find the optimal perturbation. Their imperceptibility for humans and their transferability between networks and datasets [PMG16] make them a potentially dangerous phenomenon regarding safety and security; the authors of [Eyk+18] synthetize physical adversarial "patches" used in the material world, resulting on a misclassification. A popular mitigation is *Adversarial training*. The basic idea is to generate adversarial examples, assign them to the same label of the original sample and present them to the network among regular samples. Some examples of adversarial training can be seen in [Ara+19] and [Mad+17].

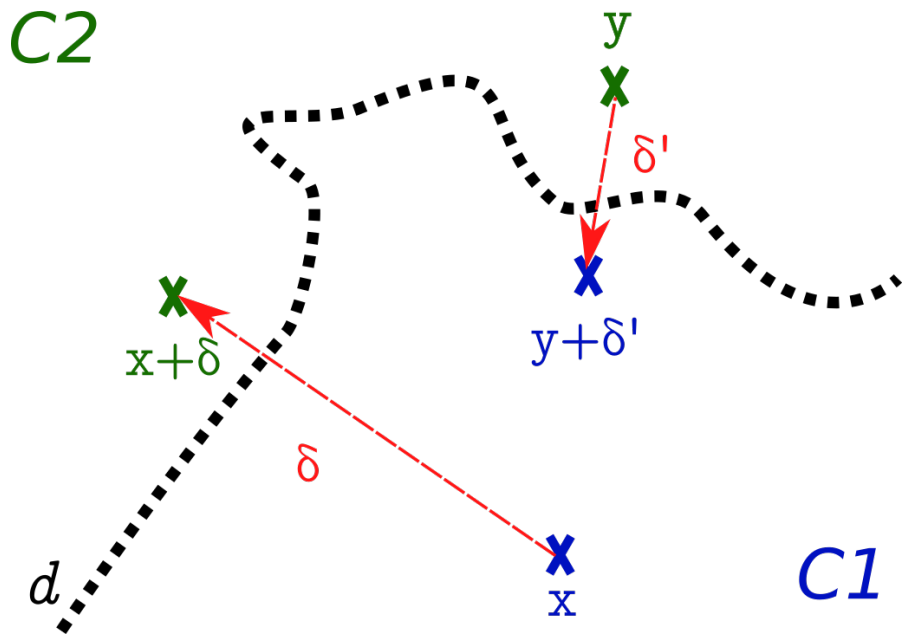


Figure 4.4: Simplified procedure to build adversarial examples. x and y are samples, δ and $\delta^{'}$ are perturbations, *d* is the decision boundary between classes *C1* and *C2*

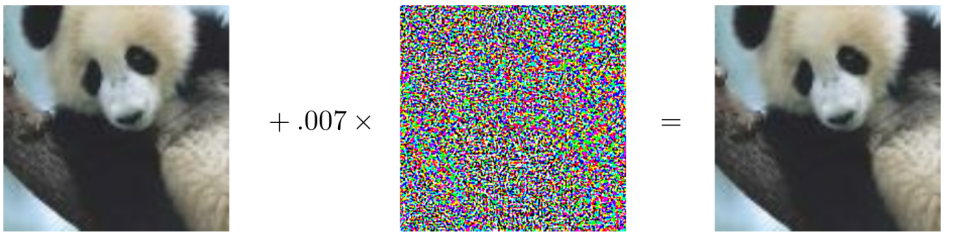


Figure 4.5: An iconic adversarial example illustration, from [GSS14]

To go further Other works focus on studying the theory behind adversarial examples. While the initial work[GSS14] suggests that adversarial examples are a result of a default in the training procedure, "bugs", investigations[Ily+19] suggest that (at least partly) adversarial examples may be inherently linked to the design principles of deep learning and to their resulting effects on programs: using any input features available to decrease the loss function, including "non-robust" features that are exploited by adversarial examples generation algorithms.

Privacy leaks

Another issue with machine learning programs is that the privacy of the data they learn is not guaranteed. It has been shown in [Tra+16] that it was possible to retrieve the parameters of a machine learning system only through its outputs, with a limited budget. In [Sho+17], the authors display a method allowing an attacker to check whether a given input has been used to train the neural network using probability outputs. This could lead to severe data leaks and directly threaten the confidentiality of the data used during training. The consensus is not clear on the practical application of this method (doubts are expressed for instance in [Tru+19]). Work is still ongoing [Cho+21] to assess the exact scope of inference attacks. To mitigate those issues, a combination of adversarial training and differential privacy is proposed in [Aba+16b] and [Ara+19].

Safety concerns, example of ACAS-Xu

Even without any intent to steal private data or to make the program malfunction, it is important to assess the ability of a neural network to output the expected outputs. Since the control flow is generated through an indirect method (minimization of an error function), even when trained on perfectly sanitized examples, it is still possible that the network learns "shortcuts" that allow it to perform very well in general, but fail on specific cases. Even when neural networks are trained on low dimensional, understandable inputs, it is thus necessary to verify that they respect their given specification, just like any other program.

A popular benchmark used in the literature is the Aircraft Collision Avoidance System for unmanned crafts (ACAS-Xu). This standard presented in [MJ16] describes a program that aims to analyze the surroundings of an aircraft (*ownship*), and output direction changes directives if another aircraft (*intruder*) is detected. The program models the situation as a 2D projection (altitude is ignored). This program has 7 input variables:

1. v_{own}: speed of the ownship;
2. v_{int}: speed of the intruder;

3. ρ: distance from the ownship to the intruder;
4. ψ: relative angle between the ownship and the intruder;
5. θ: relative angle between the speed vectors of the ownship and the intruder;
6. τ: time until loss of vertical separation;
7. a_{prev}: previous output;

Possible outputs are:

1. COC: clear of conflict, do not change direction;
2. WL: change direction to the left, lightly;
3. SL: change direction to the left, strongly;
4. WR: change direction to the right, lightly;
5. SR: change direction to the right, strongly;

The authors of [Kat+17] define some safety properties for a program to respect the ACAS-Xu specification, as well as an implementation as a neural network. Those properties are reproduced below:

1. ϕ_1: if the intruder is distant and is significantly slower than the ownship, the score of a COC advisory will always be below a certain fixed threshold;
2. ϕ_2: if the intruder is distant and is significantly slower than the ownship, the score of a COC advisory will never be maximal;
3. ϕ_3: if the intruder is directly ahead and is moving towards the ownship, the score for COC will not be minimal;
4. ϕ_4: if the intruder is directly ahead and is moving away from the ownship but at a lower speed than that of the ownship, the score for COC will not be minimal;
5. ϕ_5: if the intruder is near and approaching from the left, the network advises "strong right";
6. ϕ_6: if the intruder is sufficiently far away, the network advises COC;
7. ϕ_7: if vertical separation is large, the network will never advise a strong turn;
8. ϕ_8: for a large vertical separation and a previous "weak left" advisory, the network will either output COC or continue advising "weak left";

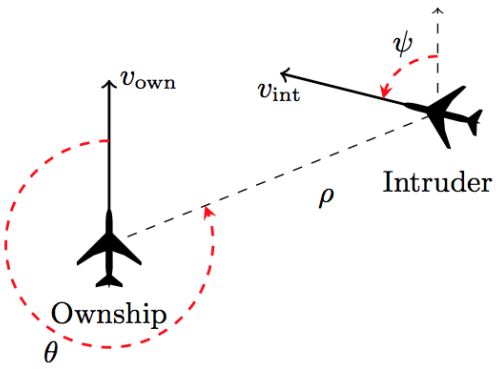


Figure 4.6: The ACAS-Xu model, with all input variables

9. ϕ_9: even if the previous advisory was "weak right", the presence of a nearby intruder will cause the network to output a "strong left" advisory instead;

10. ϕ_{10}: for a far away intruder, the network advises COC;

Verifying those properties on ACAS networks is challenging enough for Z3 and Gurobi, so it constitutes a good benchmark to test methods and tools developed for neural network verification.

4.4 Frailties of machine learning: a two-problem game

Machine learning programs differ in their design of "classical" programs in numerous ways, that prevent us to perform formal analysis. We will now expose those differences; trying to tackle those will constitute the backbone of our thesis.

What to specify

The very nature of the inputs of neural networks pose several issues:

1. Machine learning programs are working on high-dimensional inputs that are assigned a very high-level semantic (compared to classical programs): "image of a dog" encapsulates the definition of an image, a background picture, an animal, a dog; "hate speech" contains the definition of a text, a language, emotions, hate… Those "types" are highly context-dependent, yet they are the only source of specification. *Data as specification* leads to an *ambiguity in the data specification*

2. Contrary to classical programs, machine learning programs are not written through a deterministic process (be it a programmer, a compiler or a code generator), but through a stochastic algorithm. Even though the resulting program is functionally simple (ResNets [He+15] have jumps and some architecture can encode loops, but most of classical architecture are only using a succession of linear operations and activation functions), the control flow is meaningless by itself. There is thus an *ambiguity in the control flow*

3. Vulnerabilities defined in previous sections are not easy to spot (there is no "Adversarial Example" exception lifting) and even more complicated to correct, and this comes with a cost on the accuracy of the network. There is thus an *ambiguity in the specification to respect*

Key point

It is difficult to formulate properties on machine learning programs because of the following ambiguities:

1. ambiguity in inputs specification
2. lack of direct agency in the control flow
3. lack of knowledge of failure modes

Those points can be summed up into one problem, which we will tackle during our thesis: the *specification problem of machine learning verification.*

How to verify

Another issue is more directly linked with how we represent neural network for SMT or model-checking. If we consider a neural network under its CFG form, there is a possible branching for each activation function. The ReLU can be naively implemented as an if then else construct: $y = ReLU(x)$: if $x > 0$ then x else 0. This produces two mutually exclusive clauses for the SMT solver:

$$x > 0 \wedge y = x \tag{4.19}$$
$$x \leq 0 \wedge y = 0 \tag{4.20}$$

The solver must then explore those two independent paths. Since this case split occurs for each ReLU neuron, if a neural network has n neurons, then the number of cases to explore is in the order of 2^n, if done naively. Without guiding our solvers, we are bound to fail against the curse of dimensionality.

Key point

Verifying deep neural networks is difficult because of the high number of non-linear case splits their structure generates.

This is the second issue we will try to tackle: *the combinatorial problem of machine learning verification*

To go further A survey on transfer learning is available at [Tan+18]. For a comprehensive survey on the definitions, metrics and algorithms of algorithm fairness, see [Meh+19]. A survey on explainable AI is available at [Sam+21].

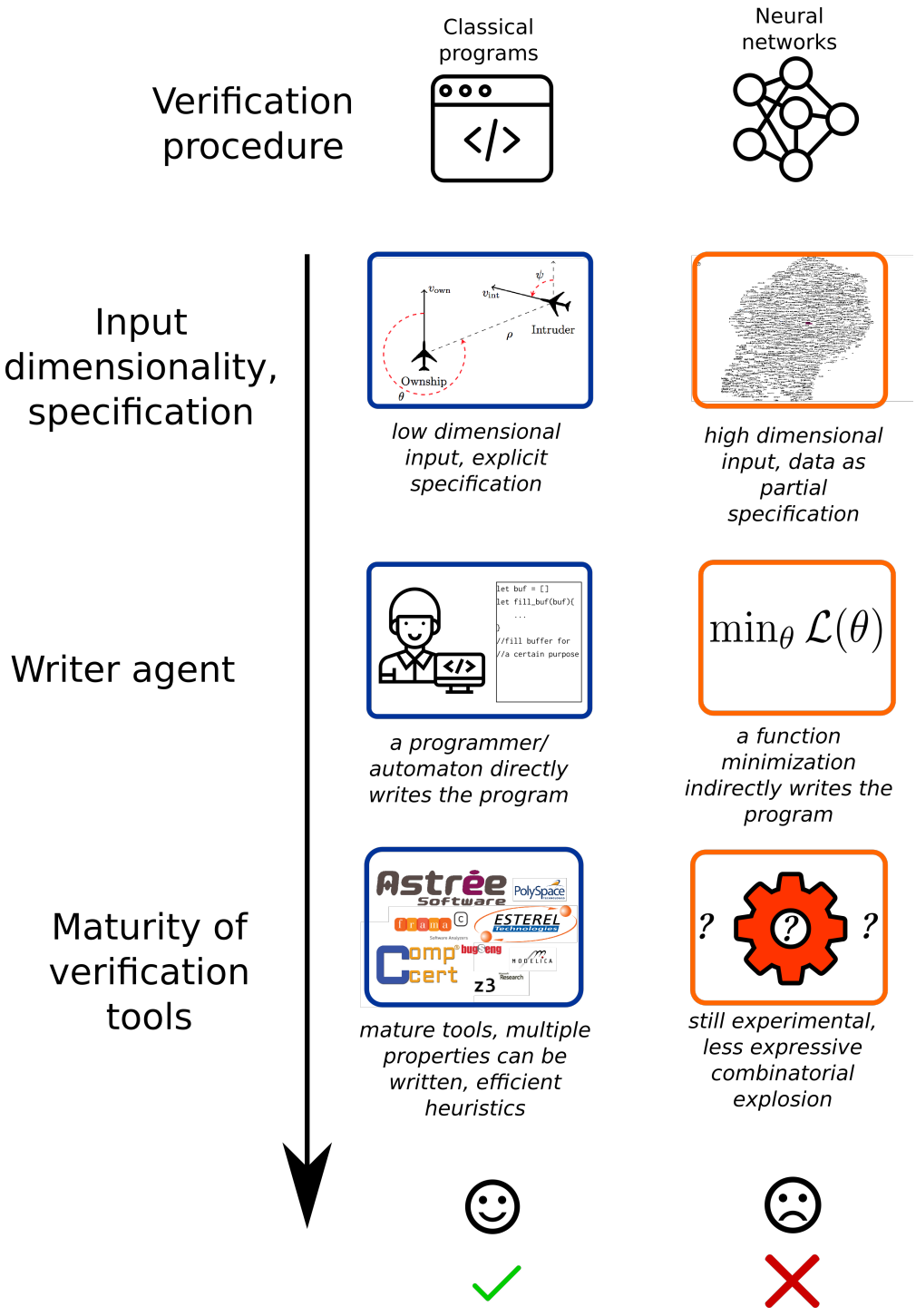


Figure 4.7: main differences between classical programs and neural networks
Icons made by Becris, Freepik, Google and Icongeek26 from www.flaticon.com

Part II

The Specification Problem

Chapter 5

The specification problem

Summary of the chapter

Most research on formal verification of deep neural networks has focused on adversarial robustness, which studies the robustness of perceptive models in the neighbourhood of particular samples. However, other works have proved global properties of smaller neural networks. Yet, formally verifying perception remains insufficiently charted. This is due notably to the lack of relevant properties to verify, as the distribution of possible inputs cannot be formally specified. We propose to take advantage of the simulators often used either to train machine learning models or to check them with statistical tests, a growing trend in industry.
Our formulation allows us to formally express and verify safety properties on perception units, covering all cases that could ever be generated by the simulator, to the difference of statistical tests which cover only seen examples.

What do deep learning programs learn? Asking this question in an assembly of scientists will probably lead to a number of interesting and enthusiast hypotheses. However, verifying experimentally those hypotheses is a difficult task. Obtaining a certainty on what is "actually learned" by a deep learning algorithm is a complicated matter, for several reasons.

The first one is that "actually learned" is somehow ill-defined, or rather, that what one would expect from a deep neural network is actually misaligned with how we make them learn. For a cancer detector applied to medical scans, or a predictor of financial trends, how do we ensure the deep learning algorithm learned what it needs to perform his task reasonably well *according to our standards*? This is tangentially linked with the "AI goal alignment issue" stated by Stuart Russel: the way we phrase objectives to artificial intelligence programs is of crucial importance, since they lack most of the cultural background, social clues and human *habitus*: *the way we phrase*

our goals shape the task much more than we expect.

The second one is that there is actually very few constraints on what the neural network learns. Data powered programming leaves all the semantic to the dataset, the labelling and the gradient descent mechanism. Thus, what the neural network learns will heavily rely on the data, their labelling and a mathematical formalism that puts very few constraints other than minimizing an error function, leaving to analyse quite alien behaviours (high dimensional spaces that we, as humans, fail to grasp intuitively, for instance). This may lead to unwanted behaviour, such as overfitting. An overfitting machine learning model maximizes its prediction score on the training set, which leads to poor generalization outside of this set. Learned features are usually not those a human would use to characterize a sample, given a similar task.

The third one is the one we will further study here: the act of training programs that learn on data and labels relies on the implicit assumption that the dataset and labelling encompass the full semantic of the concepts we are manipulating. Phrased otherwise, a dataset of collected and annotated images of birds is trusted to yield a sufficiently "well-defined idea" of a bird. We call this paradigm *data as specification*. Why use a proxy such as instances of a concept rather than the concept itself? The crux of the matter is that, sometimes, there exist *no formal characterization* of the concepts we manipulate. By formal characterization, we mean logical statements that we can input to a mechanized verification software with relative ease in order to formulate and check a verification problem. This lack of explicit formal specification is especially conspicuous with high dimensional data that we cannot reduce to a value. We define the space where such data live as the following:

Definition 13. A *perceptual space* is a high dimensional space of data sensed by humans. For instance: the space of 256×256 RGB images, the space of all possible sounds, or the set of all possible sentences in a given language.

The following questions, inspired by some of the most prominent use cases of deep learning, illustrate the difficulty of extracting a formal characterization from perceptual space:

- What are the spatial properties of a sample of hate speech? How can we isolate them from "normal" speech, while taking into account cultural differences?
- What are the geometrical features of a cat, independently of weather and lightning conditions?
- What are the features of an image containing a pedestrian? How to include people in wheelchairs in specifications alongside pedestrians?
- How can we ensure a cancer detector on radio prints will scan the actual radio and not overfit to some innocuous feature [Dou21]

Those difficulties to define formal specifications for complex concepts in the perceptual space are inherently why we use machine learning in the first place. Nonetheless, they severely limit our ability to formulate and verify formal properties in the usual way. This chapter is a scientific contribution, that was accepted at the European Conference of Artificial Intelligence (ECAI 2020), that aim to address the following question:

Research questions

How can we formulate properties on data that represent a complex concept on a perceptual space? How can we verify global properties on perceptual inputs?

5.1 Contextualization and motivation

In most deep learning application domains, such as image classification [He+15], object detection [Cha+17], control learning [Boj+16], speech recognition [RPB18], or style transfer [KLA18], there exists no formal definition of the input. As a motivating example, let us consider the (theoretical) software of an autonomous car. A visualisation of what this software does is available figure 5.1. Its goal is to detect various objects on the sensor, and output driving directives. A desirable property would be not to run over pedestrians. This property can be split in two:

1. all pedestrians are detected
2. all detected pedestrians are avoided

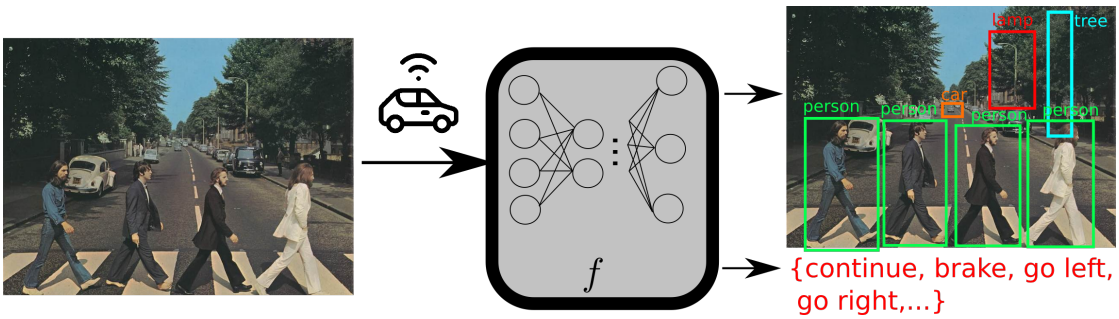


Figure 5.1: A theoretical software embedded in an autonomous vehicule. One property we would like to prove is the following: *how can we ensure that the software does not output a "continue" directive for all images with a pedestrian?*

For a formal certification, the property should be expressed in the form "For any image containing pedestrians, whatever the weather conditions or camera angle, all

pedestrians present in that image are detected and avoided". Such a formulation supposes one is able to describe the set of all possible images containing pedestrians (together with their location). However, there exists no exact characterization of what a pedestrian is or looks like, and certainly not one that takes into account weather condition, camera angle, input type or light conditions. Any handmade characterization or model would be very tiresome to build, and still incomplete.

On the upside, machine learning has demonstrated its ability to make use of data that cannot be formally specified, yielding impressive results in all above-mentioned application domains, among others; on the downside, it has also been demonstrated that ML models can easily fail dramatically, for instance when attacked with adversarial examples. Thus, manufacturers of critical systems need to provide elements that allow regulators, contractors and end-users to trust the systems in which they embed their software.

Usually, car manufacturers rely on test procedures to measure their system's performances and safety properties. But testing can, at best, yield statistical bounds on the absence of failures: the efficiency of a system against a particular situation is not assessed before this situation is actually met during a real-world experiment. As the space of possible situations is enormous (possibly infinite) and incidents are rare events, one cannot assess that an autonomous vehicle will be safe in every situation by relying on field tests alone.

A possible workaround is to use artificial data that is used to build the tests scenarios. Those artificial inputs are usually generated by a simulation software. In the context of this thesis, a simulation software is understood as a software designed to generate inputs in the perceptive space according to a certain parametrization. One of the first attempts to use simulated images to make a neural network "drive" is ALVINN [Pom89]. Multiple simulated sources were used to provide inputs and training objectives to deep neural networks, most notably from video games. AlphaStar is a deep-learning based program aiming to play at a competitive level the real-time strategy game Starcraft 2 [Vin+19]. The rich criminalistic open world video game Grand Theft Auto V was used as a source for semantic segmentation tasks [Ric+16]. For autonomous driving, simulators like Carla [Dos+17] are openly available. Notably, CARLA features multiple perceptive inputs, climate and brightness control, scripting scenarios, and provide a vast number of options for simulating an urban environment, such as the modelling of other cars and pedestrians. Using a simulation software comes with several benefits:

1. it reduces the overall cost of setting up and experimenting with autonomous driving;

2. it enables quick experimentation and reproducibility of experiments;

3. it makes it possible to generate potentially hazardous scenarios precisely, like a child running on the road or a car crash;

However, even if it is possible to artificially generate corner cases more easily, the space of possible scenarios is still enormous, and some accidents remain completely unpredictable *a priori* by human test designers. For instance, in a car accident involving partially self-driving technology, the manufacturer admitted that the camera failed to distinguish a white truck against a bright sky [Haw19], causing the death of the driver. Such a test case is difficult to come up with for a human, because it is the conjunction of specific environmental conditions and specific driving conditions.

Our motivation is to bring an additional layer of trust, not relying on statistical arguments, but rather on formal guarantees. Our long term objective is to be able to formalize a specification and to provide guarantees on every possible scenario, automatically finding violations of the specification. Because practitioners are relying more and more on simulators, we propose as a first step to study such simulated setting. More precisely, we aim to *formalize* it. The idea is to rephrase the verification problem in order to include both the deep learning model *and* the simulator software within the verification problem. As said earlier, a simulator offers more control on the learning data by providing explicit parameters (for instance: number and positions of pedestrians on the image).

Related work

Adversarial robustness: a local property

Most of the literature work has focused on *local adversarial robustness*, defined in section 4.3. Recall that adversarial perturbations are small variations of a given sample in the input space that have been deliberately crafted so that the network misclassifies the resulting noisy example. The crucial part to note here is that this property is *local*, tied to the particular sample we consider. A global adversarial robustness property could be phrased as the following:

Definition 14. A deep neural network f is *globally* ε-robust if $\forall (x_1, x_2) \in \mathcal{X}^2$ such that $\|x_1 - x_2\|_p \leqslant \varepsilon$, $f(x_1) = f(x_2)$.

Verifying this global property is intractable for several reasons:

1. *for any sample* $x \in \mathcal{X}$ where the input space is a subset of $\mathbb{R}^{256\times256\times3}$ requires to search a space of more than 195000 dimensions. Even with relatively large discretization step (0.1), for values in $[0, 1]$, this gives about $8.5e52$ possible images;

2. *for any pair of sample* requires combining two samples pairwise, further increasing the complexity;

3. if one of the samples is too close to the boundary decision ($\varepsilon > \|x - C_l\|_p$ where C_l is the boundary for class l), this property will not hold, since any sample on the other side of C_l will be classified otherwise;

For all of those reasons, most of the work on adversarial robustness has focused on local adversarial robustness.

Proving global properties in non-perceptive space

Sometimes, the neural network is used on simpler problems that allow the formulation of formal properties. By simpler, we mean two main characteristics: (i) the dimensionality of the input is much lower than in typical perception cases, where most of adversarial examples occur, and (ii) the problem the program aims to solve provides an explicit description of the meaning of the inputs and outputs, making a formulation of safety property much simpler. Rephrased otherwise, the program is working on inputs whose semantics are (at least partially) defined. Provided the inputs are sufficiently well-defined, it is then possible to encode safety properties as relationships between inputs and outputs, such as inequality constraints on real values.

An example of such setting can be seen in the Anti Collision Avoidance System for Unmanned aircraft (ACAS-Xu), that we defined in section 4.3. Inputs correspond to aircraft sensors, and outputs to airplane commands. In such case, specifications can be directly encoded as a set of constraints on the inputs and outputs. We emphasize here that the inputs of the program are *not perceptual*: intruder distance, relative angle with the aircraft and relative speed vector... all the data were already obtained by the sensors and pre-processed to give a low-dimensional input to the program. In other words, an ACAS software can be seen as taking the output of a perception unit, and assumes that those outputs are correct. In [Kat+17], the authors proposed an implementation of ACAS-Xu as a deep neural network, and they were able to formally prove that their program respected various safety properties.

Using simulators for verification

Recent work proposed to analyse programs trained on simulators [Dre+19]. Although their motivations are similar to ours, they work on abstract feature spaces without directly considering the perception unit, and they rely on sampling techniques while we aim to use sound, exhaustive techniques. Their aim is to exhibit faulty behaviour in some type of neural network controllers, while we can formally verify any type of perception unit.

On one side, there exist a limited number of *local* verification properties on the *perceptual* space. On the other, we have *global* properties on non-perceptual spaces. With our contribution, we aim to bridge the two worlds by providing a way to express *global* properties on perceptual inputs.

5.2 CAMUS: a new formalism to specify machine learning models

Problem formulation and notations

Let $f : \mathcal{X} \to \mathcal{Y}$ be an algorithm taking a perceptual input $x \in \mathcal{X}$ and yielding a decision $y \in \mathcal{Y}$. The perceptual space $\mathcal{X}$ will typically be of the form $\mathbb{R}^d$ or $[0,1]^d$. f is a program trained with a learning procedure on a finite subset of $\mathcal{X}$ to perform a specific task (e.g., drive the passengers safely home). In our example, the task would be to output a command from an image, in which case, for a given image x, $f(x)$ would be the driving action taken when in environment x.

Let us denote by $g : \mathcal{S} \to \mathcal{X}$ the simulator, that is, a function taking as input a configuration $s \in \mathcal{S}$ of parameters, and returning the result of the simulation associated to these parameter values. A configuration s of parameters contains all the information needed by the simulator to generate a perceptual input; each parameter may be a discrete or continuous variable. Taking for example a simulator of autonomous car scenarios such as CARLA: s would contain the road characteristics, the number of pedestrians and their positions, the weather conditions…, that is, potentially, thousands of variables, depending on the simulator realism.

The problem to solve here is the following: *For a model f trained on data belonging to $\mathcal{X}$ generated by g to perform a certain task, how can we formulate and formally verify practical safety properties for all possible $x \in \mathcal{X}$, including samples never seen during training*?

Including the simulator in the verification

In classical formal verification settings, such as the one schematized in figure 5.2, specifications express relationships from $\mathcal{X}$ to $\mathcal{Y}$, using a formulation of f. But in autonomous driving, $\mathcal{X}$ is such a huge space that formulating properties that are non-trivial, let alone verify these, is prohibitively difficult, especially in the case of perceptive systems where the domain of x cannot be specified: all matrices in $([0,255]^3)^{\#\text{pixels}}$ are images, technically, but few of them make sense, and one cannot describe which ones. Moreover, given an image x, the property to check might be difficult to express, as, to state that all pedestrians were detected and avoided, one needs to know whether there are pedestrians in x and where, which we do not know formally from just the image x. And if one had a way to retrieve such information from x (number and location of pedestrians) without any mistake, one would have already solved the initial problem, i.e., safe self-driving cars.

To summarize, in this setting, it is impossible to express a relevant space for x and

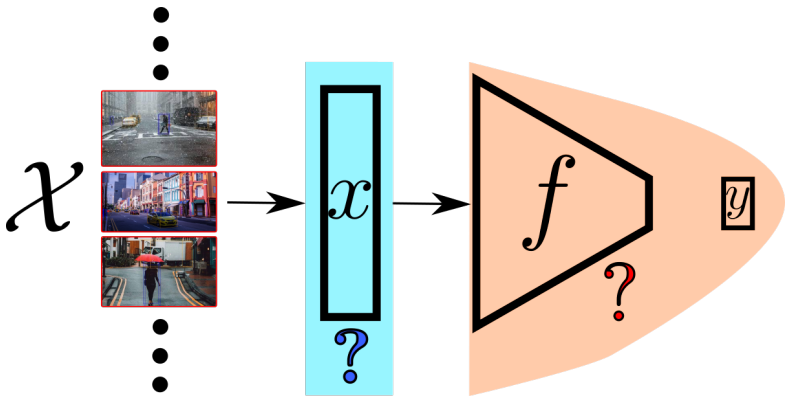


Figure 5.2: Natural inputs with huge perceptual space: no formal characterization of the input can be formulated, preventing formal verification

a property to verify Φ, as expressed in the following equation (5.1).

$$\forall x \in ?, \quad \Phi?\left(?, f(x)\right) \tag{5.1}$$

In the setting of simulated inputs, though it remains difficult to formulate properties on the perceptual space $\mathcal{X}$, we know that this space is produced by g applied to parameters in $\mathcal{S}$. Contrary to $\mathcal{X}$, $\mathcal{S}$ is a space where there exists an abstract, albeit simplistic characterization of entities. This characterization comes from choices made during the conception of the simulator. Indeed, setting on parameter s_p for a pedestrian in the simulated input to appear will effectively make it appear. Thus, the state of s_p effectively holds a specification of what a pedestrian is in $\mathcal{X}$. The procedure g transforms elements $s \in \mathcal{S}$, that represent abstracted entities, into elements $x \in \mathcal{X}$ that describe these entities in the rich perceptual space. To output values in $\mathcal{Y}$, f has to capture the inner semantics contained in $\mathcal{X}$, that is to say, to abstract back a part of $\mathcal{S}$ from $\mathcal{X}$.

The above remark is the key to the proposed framework: if we include $\mathcal{S}$ and g alongside f, $\mathcal{X}$ and $\mathcal{Y}$ in the verification problem, then all meaningful elements of $\mathcal{S}$ are *de facto* included. It then becomes possible to formulate interesting properties, such as the following: "given a simulator that defines pedestrians as a certain pattern of pixels, does a model trained on the images generated by this simulator avoid all pedestrians correctly?".

Key point

Including a simulator capable of generating perceptual inputs inside the verification problem allows to formulate formal properties on perceptual space, provided we limit ourselves to inputs that can be generated by the simulator.

Formally, to ensure that the output $y = p \circ g(s)$ satisfies a property Φ for all examples $x = g(s)$ that can ever be generated by the simulator, the formula to check is of the form described in equation (5.2)

$$\forall s \in \mathcal{S}, \quad \Phi\big(s,\ p \circ g(s)\big) \tag{5.2}$$

It is possible to describe the input space of x, as piloted by certain values of s, and is thus possible to describe the output property $\Phi(f(x))$. The property Φ may depend on s indeed, as, in our running example, s explicitly contains the information about the number of pedestrians to be avoided as well as their locations.

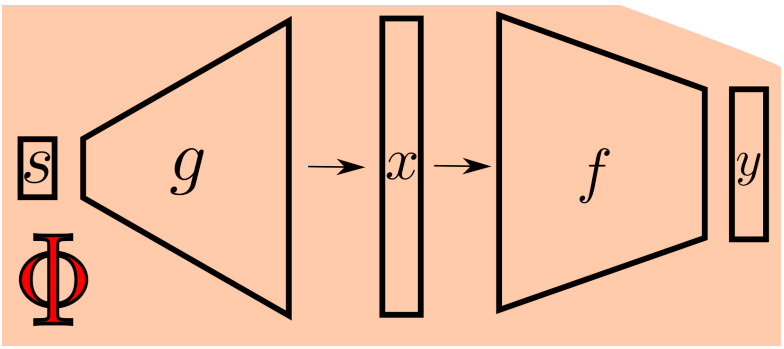


Figure 5.3: Generated inputs with integration of the generation procedure in the verification problem. There are now new properties to check since we have a formal characterization of the perceptual elements.

Including $\mathcal{S}$ and g in a formal property to check requires to formulate at least partially the multiple functions that compose g. Describing precisely these procedures is a key problem, that is also quite difficult. Indeed, if the simulator is a classical program, then the whole set of techniques defined in the introduction is available to verify it. For instance, reachability analysis could be used on the simulator to monitor the effect of a particular parameter on another. It would also be necessary to define properties to check, which requires careful thinking and formalization. It could also be possible to trust not the simulator as a whole, but only key parts, or a modelization of the simulator - that would also require formalization. If the simulator is a machine learning program, the relationships between $\mathcal{S}$ and $\mathcal{X}$ should be carefully controlled and verified. Formalizing simulators is a complex endeavour, that would require expertise in programming language design and simulation software, an expertise that we do not have. Since this thesis aims to cover the different aspects of the formal verification discipline - specification, tooling, heuristics -, we leave the topic of formalizing simulators for future work. For the rest of this thesis, we consider that there exist a sufficient description of g to be included in the whole verification process.

As our framework relies on including the simulator in the verification problem, we call it Certifying Autonomous deep Models Using Simulators (CAMUS).

Separating perception and reasoning

Before the rise of deep learning, the perception function (which recognizes a certain pattern of pixels as a pedestrian) and the control, or reasoning function (which analyses the location of a pedestrian and proposes a decision accordingly) in vehicles were designed and optimized separately. However, work such as [Boj+16] showed that end-to-end learning can in general be a much more efficient alternative; there exist many incentives to adopt an end-to-end architecture, mixing and training jointly the perception and control functions. However, combining perception and reasoning into one model makes the formulation of safety properties more difficult.

Thus, in our description (see figure 5.4), we choose to separate the perception and the reasoning functions, respectively in the components p and r. The perception part p is in charge of capturing all relevant information contained in the image, while the reasoning part r will make use of this relevant information to output directives accordingly to a specification.

One way to make sure that p retrieves *all* relevant information is to require it to retrieve *all* information available, that is, to reconstruct the full simulator parameter configuration s. In this setting, let $s^{'}$ be the output of the perception module p. It lies in the same space as the parameter configuration space $\mathcal{S}$, and the property we would like to satisfy can be written as $p \circ g \approx \mathrm{Id}$, which can be rewritten as:

$$\forall s \in \mathcal{S}, \quad p \circ g(s) \approx s \tag{5.3}$$

This way, we ensure that the perception module p correctly perceives *all* samples that could ever be generated by the simulator. In case some parameters are known not to be relevant (image noise, decoration details, etc.), p can learn to retrieve only the relevant ones (for instance by zeroing out the non-relevant parameters). For the sake of notation simplicity, we will here consider the case where we ask to reconstruct all parameters.

This separation between perception p and further reasoning r brings modularity as an additional benefit: even when dealing with different traffic regulations or specifications, it is only necessary to prove p once; the verification of compliance towards local legislations and specifications by r can be done separately. Indeed, specifications on how to navigate on the road are *more susceptible to change* than specifications on what to perceive. It allows to reuse the complex perception module without needing to prove it again, with different reasoning modules r. Note also that r does not need to be as complex as p, since it will work on much smaller spaces; verifications of r are then easier.

One could argue that this formulation makes the problem more complex, and it indeed may be the case. Our approach only work if the perception and reasoning units are clearly separated as software components, while the trend tend to use end-to-end learning when possible. However, our proposition is aimed at safety, and in

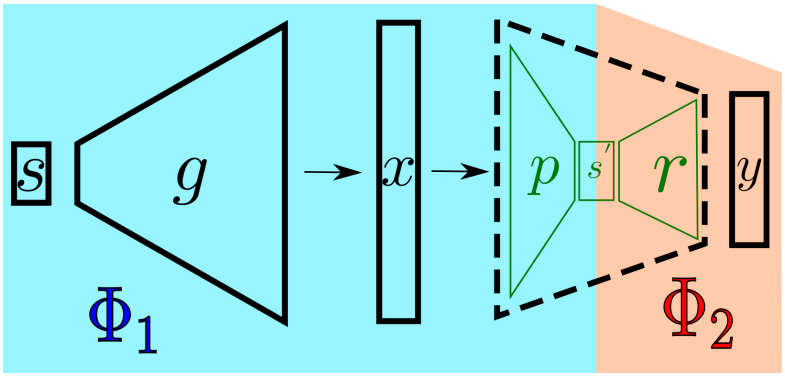


Figure 5.4: Integration of the generation procedure in the verification, with split between perception and reasoning: p learns to capture all the relevant parameters; r learns to respect the specification. Verifying ϕ_1 proves the perception module once and for all; verifying ϕ_2 can be done when the specification changes (e.g., for different driving rules).

order to provide additional trust, it is sometimes necessary to formulate the problem differently. For instance, there are good practices to structure the code to provide some safety guarantees: bounded loops, correctly allocated and de-allocated references, ban of function references and goto statements … are constructs that voluntarily restrain the expressive power of the programming language to ensure a safer behaviour. Hence, although this formulation might seem like a step back with regard to the state-of-the-art, we argue that it provides a new way to formulate safety properties, and hence will be beneficial in the long run.

Properties Formulation

Considering jointly the simulator g and the machine learning model f, split in p and r, two families of properties are amenable to formal checking:

- Φ_1: perception module p has captured sufficient knowledge from $\mathcal{X}$;
- Φ_2: reasoning module r respects a specification property regarding $\mathcal{Y}$.

Families of property ϕ_2 have been addressed in the literature – see section 5.1. The key point of the proposed approach is thus to obtain a representation space that reliably yields semantic meaning, which is the objective of Φ_1. Since the simulator is included in the verification problem, properties of family Φ_1 can be written as relationships between input parameter configurations $s \in \mathcal{S}$ and retrieved parameter configurations $s' \in \mathcal{S}$, outputs of the perception module p. Strict equality between s and s' may be difficult to achieve, and is actually not needed as long as the reasoning module r is able to deal with small estimation errors.

Expressed in the proposed formalism, the perception task is equivalent to finding (a good approximation of) $\mathcal{S}$. Thus, a relaxed version of property equation (5.3) to satisfy could be formalized as some tolerance $\varepsilon > 0$ on the reconstruction error $\|s' - s\|$ (for some metric $\|\cdot\|$):

$$\forall s \in \mathcal{S}, \quad \|s - p \circ g(s)\| \leqslant \varepsilon \tag{5.4}$$

5.3 Discussion

Specification through the simulator puts the burden of trust on the simulator. This therefore require sufficient grounds to trust the simulator software, in particular its ability to sufficiently well approximate the real input space. As companies such as Ansys and Siemens are developping numerical simulation software that are taking part in a wider certification process of systems, it appears that trusting simulators is an existing practice. This trust may not come from formal verification tools however, and it would be an interesting venue to find practical ways to apply formal verification tools on simulators.

Some recent work [FG19; Kat+21] propose to formally verify deep neural networks using deep learning generative models as simulators. Generative models are specific deep neural network programs able to approximate an input distribution and produce synthetic but realistic inputs sampled from this approximated input distribution. Providing the generators can be written as ReLU networks, their work is quite nicely fitting in the CAMUS framework. Still remains the problem of trusting a neural network based simulator.

As stated earlier, it is not always necessary to retrieve all parameters of configuration s. For instance, one could seek to retrieve only the correct number of pedestrians and their locations, from any image generable by the simulator. In this case, the output of p would be just a few coefficients of s, and must consequently be characterized differently (*e.g.,* as belonging to a given subspace of $\mathcal{S}$). This would allow expressing more flexible properties than simply reconstruct all parameters.

For the model f to correctly generalize, the simulated data must have two characteristics:

1. it needs to be sufficiently realistic (that is to say, they should look like real-world images); if not the network could overfit the simplistic representation provided by the simulator;

2. it must be representative of the various cases the model has to take into account, to cover sufficiently diverse situations.

Additional characterization of the simulator would be difficult. For instance, one could suggest requiring the simulator g to be either bijective or injective, in order to

cover all possible cases $x \in \mathcal{X}$, or for parameters to be uniquely retrievable. Yet, the largest part of the perceptual space $\mathcal{X}$ is usually made of nonsensical cases (think of random images in $([0, 255]^3)^{\#\text{pixels}}$ with each pixel color picked independently: most are just noise), and the subspace of plausible perceptual inputs is generally not characterizable (without which the problem at hand would already be solved). Regarding injectivity, being one-to-one is actually not needed when dealing with properties such as equation (5.4).

Finally, let us consider the case where several simulators g_i are available, and where, given a perceptive system p, we would like to assert properties of type Φ_1 for each of them. At first glance, as the output of p consists of retrieved parameters, this would seem to require that all g_i are parameterized exactly identically (same $\mathcal{S}$). However, for real tasks, one does not need to retrieve all parameters but only the useful ones (*e.g.,* number of pedestrians and their locations), which necessarily appear in the configuration of all simulators (at least those generating images of roads).

For a simulator g_i and a simulator g_j with respective parameter spaces $\mathcal{S}_i$ and $\mathcal{S}_j$, it would be possible to write a mapping between relevant parameters in $\mathcal{S}_i$ and $\mathcal{S}_j$. For instance, if the two simulators are detecting pedestrians with a different number of parameters, a linear combination of those parameters could be given to a unique perception unit p, and formal properties for all simulators can thus be expressed.

To go further About the issue of goal alignment, the curious reader is encouraged to read the various work of Stuart Russel on that matter, for instance in [RND10], or [Wor15]. The legend of King Midas, who asked to be able to change everything he touched into gold and perished because he was not able to eat, could be seen as an early example of the alignment problem: asking for something and wanting something are two different things. An instanciation of the goal alignment problem can be seen on model explainability, where what is explained can be nebulous. On this topic, the most excellent paper [Lip16] explain nicely how "explainability" is actually a complex notion that cannot be addressed easily.

Chapter 6

ISAIEH: the Inter Standard AI Encoding Hub

Summary of the chapter

We present ISAIEH, a tool aiming to bridge formal verification and neural network representations on a common ground. We describe the overall software architecture of ISAIEH, and its main features: a modular, extensible intermediate representation, a compiler from ONNX to LP and SMTLIB2 formats. We present a use case of ISAIEH, applied to CAMUS, and show how ISAIEH can be used to ease the verification process on deep neural networks.

A practical problem arose very early in the thesis: it was difficult to use existing formal verification tools on neural networks. Indeed, neural networks are usually saved in formats that are not adapted to the tools we considered. Most of the deep learning frameworks provide mostly an interface with C++ and Python programming languages, other APIs were limited both in functionality and documentation at the beginning of this thesis. At this time, formal verification of neural network was still a nascent field. No standards allowed an easy communication with standard formats used in the formal method community, such as SMTLIB or LP. Phrasing any specification was difficult for the same reason, because deep learning frameworks did not offer ways to express (or prove) properties. Thus easing out the expression of properties and the communication between machine learning and formal verification standards was a necessary first step in order to start comparing techniques on a fair basis. This chapter describes our answer to this issue: the Inter Standard Artificial Intelligence Encoding Hub (ISAIEH).

6.1 Existing tools and formats for formal verification and deep learning

To understand the software context in which our artifact will live, we will describe shortly the tools usually used in deep learning, as well as present a state of the art on deep learning verification tools.

Tools for machine learning practice

Deep learning manipulates high-dimensional data, often represented as multidimensional arrays (or tensors). Efficient manipulation of tensors is crucial. To do so, deep learning needs to make heavy use of parallelism and efficient memory management. Another key point is that deep learning programs need to be differentiable, as stated in chapter 4. Finally, deep learning researchers need to be able to prototype relatively easily their ideas, without needing a PhD on parallel programming or knowing the quirks of memory layouts.

To answer those needs, several deep learning framework were developed over the last decade. The one we mainly used for this thesis is PyTorch [Pas+19], we occasionally used TensorFlow [Aba+16a]. Other include Caffe2 and the Microsoft Cognitive Toolkit. They provide automatic parallelization of tensor computation, a vast collection of deep learning operators and clarify the flow of deep learning programming. Most of those frameworks operate on their own inner intermediate representations, and save the resulting program as blobs (binary large objects), framework-specific format. Because deep learning research and application should not be limited by the kind of framework used during software development, several big players (Microsoft, Facebook…) started to develop a common representation for deep neural networks: the Open Neural Network eXchange (ONNX) format. It provides a unified view of deep learning programs as directed acyclic graphs, where each node represents a computation applied to input tensors. For the rest of this thesis, we almost exclusively worked on ONNX networks. Neural networks were designed, trained and tested using PyTorch, then saved under the ONNX format.

Thus, the question we aim to answer here is the following:

Research questions

How can we express deep learning programs in a way suitable for formal verification?

A standard language for SMT verification: SMTLIB2

Mature SMT solvers are complex engineering artifacts with highly specialized heuristics. Those tools are used in multiple settings by different teams, who need to have a way to collaborate on their research topic with relative ease. To ensure this, an international effort led to the birth of the SMTLIB standard. The latest version (2.6 at the time of redaction) [BFS17] provides the following (citation from the official website of the initiative[1]):

1. Provide standard rigorous descriptions of background theories used in SMT systems
2. Develop and promote common input and output languages for SMT solvers
3. Connect developers, researchers and users of SMT, and develop a community around it
4. Establish and make available to the research community a large library of benchmarks for SMT solvers.
5. Collect and promote software tools useful to the SMT community

We target the SMTLIB2 input language. Its grammar is quite simple, as it is only comprised of two elements: atoms and expressions:

```
type expr = Atom of string | expr list
```

A SMTLIB2 program is a succession of expressions (enclosed within parenthesis, much like Lisp's Symbolic expressions) that encode the various constraints the SMT solver should consider, as well as generic options (such as the theory to consider) and solver-specific options.

6.2 ISAIEH: an encoding hub for neural networks

ISAIEH stands for Inter Standard Artificial Intelligence Encoding Hub. It is written in OCaml. OCaml is a statically and strongly-typed language. Providing we design the type of our data according to a sensible specification, OCaml brings an additional layer of trust in our program by spotting type errors at compile time rather than runtime, which is a plus when building relatively complex tools such as formal verification tools. Additionally, the laboratory in which this thesis was conducted was gifted with several experts in the OCaml programming language, which helped kickstart some ideas and rapid prototyping. Since the flagship formal verification tool designed in the lab,

[1] http://smtlib.cs.uiowa.edu/

Frama-C [Bau+21], was written in OCaml, writting a prototype of a formal verification tool for neural networks that could potentially interface with it seems a sensible thing to do. And finally, the author of this thesis has an aesthetic bias towards functional programming and tends to use it whenever it is reasonable to do. The overall architecture of ISAIEH is available figure 6.1.

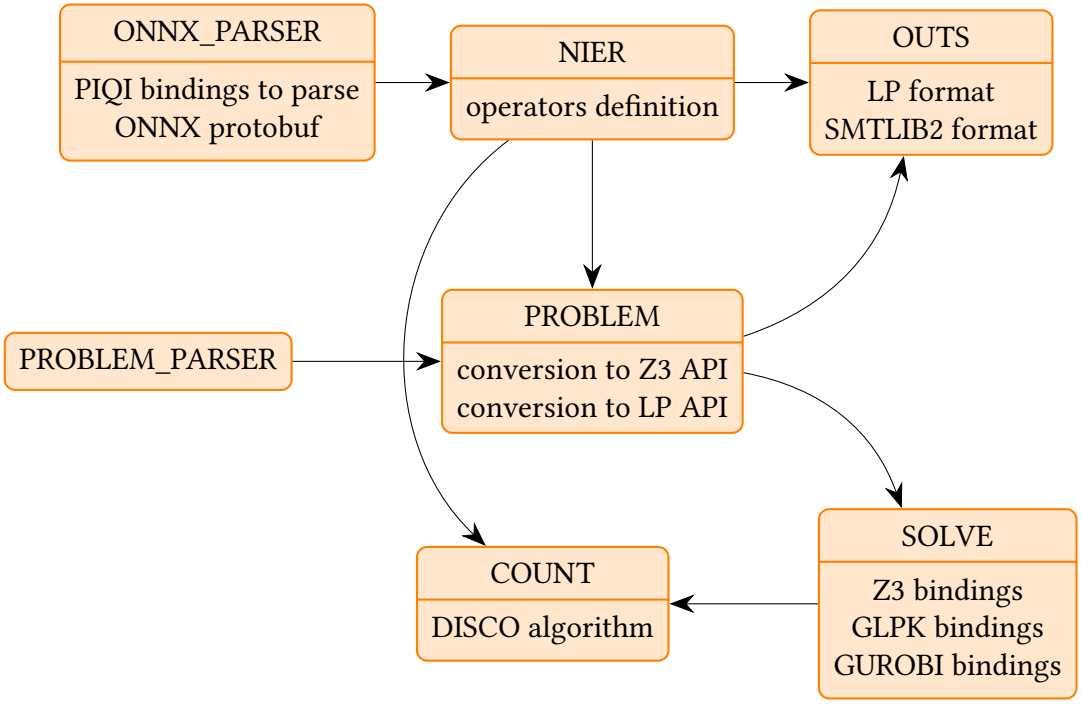


Figure 6.1: ISAIEH architecture. Arrows indicate a dependency.

What need does it fulfill/features

Very early in the thesis, the need to smooth communication between the neural networks world and the formal verification world arose: benchmarking state-of-the-art solvers was the first necessary step, and a tool was already needed for that. Another need was to be able to work at different abstractions level, from a single neuron to the entire architecture, to perform analysis on the neural network. Finally, during the thesis, the need arose to reimplement some state-of-the-art tools and to develop our own verification technique. For all of those reasons, we needed to develop a tool that was:

1. expressive enough to take into account most of the neural networks constructs
2. flexible in order to be able to add different kinds of algorithms

3. able to bind neural networks to formal verifications tools in a streamlined way

The (ongoing) result of this work is the Inter Standard Artificial Intelligence Encoding Hub (ISAIEH). The code is freely available under the LGPLv2 licence[2]. The most prominent features of ISAIEH are the following:

1. support of ONNX standard to produce a Neural IntermediatE Representation (NIER)

2. transcription of the NIER to SMTLIB2 and LP format to be used by state-of-the-art solvers

3. implementation of facets counting

4. bindings to the Z3 SMT solver to perform direct solving of SMT problems, and to the GLPK and GUROBI LP solvers to solve LP problems

Description of the NIER

The NIER is an acyclic directed graph G with vertices V and labelled edges E. Let $v \in V$ be a vertex, let $e_i \subset E$ be the input edges of v. There is only one output for vertices. Each v represents the application of a mathematical operation on the input of the node (determined by the labels of input edges e_i). Supported operations are a subset of the ONNX standard, plus some custom operations used for specific solving strategies, detailed later on. Most of those operations are not commutative, so it is necessary to keep from the ONNX description the operands and their order in a specific field. Finally, some nodes have no operation but store numerical values ("initializers" in the wording of ONNX): those are the parameters of the networks. They are stored in specific vertices, "data vertices". Edges describe the control flow of tensors going through the calculus graph. Each label is labelled with a unique string identifying a tensor. A vertex has the following attributes:

This NIER is mostly descriptive. The main difference with ONNX is the merging between "initializer" nodes and classical nodes. Since it is much smaller in scope, it is also easier to manipulate and extend than ONNX. Applications leveraging this NIER are described in the following section.

The SMTLIB2 compiler and SMTLIB solver API

The first use developped for ISAIEH was to compile down a neural network control flow to a format that was readable by standard SMT solvers; the SMTLIB2 format was

[2] `https://git.frama-c.com/pub/isaieh`

```
type id = int
type shape = int list
type operator = | Add | Mul | Matmul
  | LogSoftmax | ReLu | Transpose
  | Squeeze | MaxPool | Conv
  | Identity | NO_OP | RW_Linearized_ReLu
type ksize = Ksize of shape
type stride = Stride of shape
type pads = Pads of shape
type dilations = Dilations of shape
type operator_parameters =
| Pool_params of (ksize * stride option *
                 pads option * dilations option)
| Conv_params of (ksize * stride option *
                 pads option * dilations option)
| Transpose_params of shape
| RW_Linearized_ReLu_params of ((bool list list) *
                                ((string, float) Hashtbl list *
                                 int))
type t = {
    id: id;
    name: string option;
    mutable shape : shape;
    operator: operator;
    operator_parameters: operator_parameters option;
    pred : string list;
    succ : string list;
    tensor : Tensor.t option
}
```

Figure 6.2: Attributes of vertex

chosen. The main difficulty is that while the semantics of deep learning heavily relies on multidimensional floating-point array computations, such constructs are not supported by the SMTLIB2 standard, nor any state-of-the-art SMT solver. SMTLIB2 standard theories, the Quantifier-Free Non-Linear Arithmetic (QF_NRA) and its linear counterpart (QF_LRA), only work on single numbers, and the expressivity of SMTLIB2 language is not enough to redefine easily those computations. It was then necessary to define a “flattened” version of all neural networks computations. Here, “flattened” means that every single cell of the tensor needs to be registered and calculated as an

independent variable. The SMTLIB2 compiler thus takes all nodes of the NIER, creates individual variables for each underlying tensor, performs the underlying mathematical operation using previously defined variables (symbols), up until the last layer. Then, variables and operations are written according to SMTLIB2 syntax and written to a file.

Key point

ISAIEH compiles a neural network control flow to SMTLIB2 language, the standard input language for SMT solvers.

Using ISAIEH with CAMUS

We used ISAIEH to demonstrate a use case of CAMUS on a simple synthetic problem. Let us consider here the perception module of an autonomous vehicle, whose goal is to output driving directives that result in safe driving behaviour. The perception module is modeled as a deep neural network with one output node, taking as input an image. If an obstacle lies in a pre-defined "danger zone", the network should output a "change direction" directive. Otherwise, it should output a "no change" directive. The "simulator" is here a Python script, taking as input the number and the locations on the image of the one-pixel wide obstacles and generating the corresponding black-and-white images. The verification problem consists in the formulation of the network structure and constraints on the inputs, and in the following properties to check:

1. verify that an input with an obstacle (or several ones) in the danger zone will always lead to the "change direction" directive;

2. verify that an input without obstacle on the danger zone will never lead to the "change direction" directive.

If both properties are verified, our model is perfect for all the inputs that can be generated. If the first one is not verified, our verification system will provide examples of inputs where our model fails, which can be a useful insight on the model flaws. Such examples could be used for further more robust training, *i.e.,* integrated into a future training phase to correct the network misclassifications. Similarly, if the second property is not verified, the solver will provide false positives, that can help designers reduce erroneous alerts and make their tools more acceptable for the end-user. In this toy example, input data are $N \times N$ black-and-white images (see Fig. 6.3 for examples). The space of possible simulated data $g(\mathcal{S}) \subset \mathcal{X}$ can simply be described by the constraint that each pixel can only take two values (0 and 1). In real life, data are much more complex, possibly continuous; such data can also be handled in our framework. The neural network is fully-connected with two hidden layers. The number of neurons

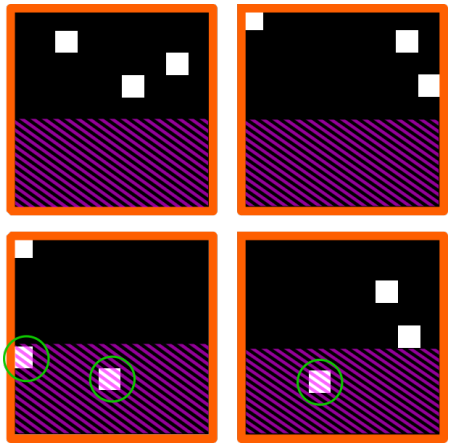


Figure 6.3: Example of inputs for the toy problem. White pixels represent obstacles. If they are in the top half of the image, no alert should be fired (first two examples), while an alert should be fired if at least one lies in the (dashed) bottom half of the image (last two examples). 9x9 picture is depicted here for clarity.

in the first and second hidden layers are respectively one half and one quarter of the flattened size of the input (N^2). The danger zone is defined as the bottom half part of the image. Any image with at least one white pixel in this zone should then yield a "change direction" directive: a binary cross-entropy loss function is used.

We use z3 [dMB08], CVC4 [Bar+11], YICES [Dut14] and COLIBRI [MBC17] SMT solvers as standard verification tools.

Here, constraints on inputs are encoded as statements on the SMT-LIB variables. A fragment of property to check is presented on Figure 6.4. On such a simple problem, the decomposition perception/reasoning is not needed, since there exists a formal characterization of what an obstacle is. All network's parameters were converted using the QF_NRA theory.

The goal was to return UNSAT, meaning no counterexample to the property was found. Under the hypothesis that the space of all possible data is described by the simulator (which is the case here), the model will never fail to detect obstacles. Runtimes are available in Table 6.2. For $N = 5$ and $N = 7$, most of the possible input configurations were not seen in the training or test sets. Yet, the network is proven always correct, which shows interesting generalization abilities. We were thus able to verify quickly that this perceptive module will *never* miss obstacles, at least for inputs generable by the simulator.

N	3	5	7
Train set size	461	500	500
Test set size	51	500	500
Total possible number of samples	512	2672	19650

Table 6.1: Total number of possible samples for each N

N	z3	CVC4	YICES	COLIBRI
3	0.04s	0.08s	TIMEOUT	UNKNOWN
5	30.2s	61.2s	TIMEOUT	TIMEOUT
7	434s	TIMEOUT	TIMEOUT	TIMEOUT

Table 6.2: Runtimes of solvers to answer UNSAT. TIMEOUT is one hour.

6.3 Discussion

The toy example presented last section is of course still simplistic; much work on scalability is needed before real self-driving car simulators can be incorporated into formal proofs.

While we provide a toolkit to translate neural network directly in our framework, a way to easily represent a simulator is yet to be included. It is not an easy task, since the simulator must be describable with sufficient granularity to allow the solver to use the simulator's internal working to simplify the verification problem. A scene description language with a modelling language for simulators is a possible answer to these issues. Further theoretical characterization of the simulator procedure and its link with the perceptive unit will be undertaken, for instance to encompass stochastic processes.

Our current framework checks properties for all possible inputs, including anomalous ones such as adversarial attacks. A possible extension would be to identify "safe" subspaces instead, where perception is guaranteed to be perfect, and "unsafe" subspaces where failures may happen.

Altough our example uses a handcrafted simulator, our framework is not limited to it, provided sufficient trust is put under the simulator design. A recent line of work [Tol+21; FG19; Kat+21] make use of simulators built using generative models trained on a certain input distribution. The noise parameters used to tune the simulators are representative of the input distribution, and can thus be used as proxies for specification using CAMUS.

```
;;;; Automatically generated part
;; Inputs declaration
(declare-fun |actual_input_0_0_0_8| () Real)
                . . .
;; Weights declaration
(declare-fun |l_1.weight_31_4| () Real)
(assert (= |l_1.weight_31_4| (/ -5585077 33554432)))
                . . .
;; An example of encoded calculation
(assert (= |8_0_0_0_39| (* |actual_input_0_0_0_8| (+ |7_80_39| (* |
    actual_input_0_0_0_7| (+ |7_79_39| (* |actual_input_0_0_0_6| (+ |7
    _78_39|
                . . .
;; Outputs declaration
(assert (= |actual_output_0_0_0_1| ( + |16_0_0_0_1| |l_3.bias_1| )))
                . . .
;;;; Handmade annotations
;; Simulator description
;; Input space constraints:
;; inputs between 0 and 1
(assert (or (= actual_input_0_0_0_8 0) (= actual_input_0_0_0_8 1)))
                . . .
;; Property to check
;; ‘‘If at least one input in
;; the danger zone is white...’’
(assert
  (or
    (or (= actual_input_0_0_0_5 1.)
        (= actual_input_0_0_0_6 1.))
    (or (= actual_input_0_0_0_7 1.)
        (= actual_input_0_0_0_8 1.))
                . . .
;; Formulate constraint on outputs
;; ‘‘... then output for detected
;; obstacle always higher
;; than for no obstacle’’
(assert (> actual_output_0_0_0_0 actual_output_0_0_0_1))
```

Figure 6.4: A SMTLIB2 file describing our problem. First part is a full description of the network, automatically produced by ONNX2SMT. Handmade annotations describe the property to check, i.e. there are no false negatives in our network. The goal for the solver is to find a counterexample.

Part III

The Tools and Heuristics Problem

Chapter 7

The tooling problem

Summary of the chapter

The impressive results of modern neural networks partly come from their non-linear behaviour. Unfortunately, this property makes it very difficult to apply formal verification tools, even if we restrict ourselves to networks with a piece-wise linear structure. However, such networks yield subregions that are linear and thus simpler to analyse independently. In this chapter, we present a method to simplify the verification problem by partitioning it into multiple linear sub-problems. To evaluate the feasibility of such an approach, we perform an empirical analysis of neural networks to estimate the number of linear regions, and compare them to the bounds currently known. We also present the impact of a technique aiming at reducing the number of linear regions during training.

In the previous part, we presented a possible way to address the specification problem encountered when performing formal verification on deep neural networks. There is still the need of actually performing the mechanism of verification, by using specific tools that are properly tuned for the problem at hand. For this chapter, we will review the performance of state-of-the-art formal verification tools on deep neural networks and present some insights on why we might need new tools.

7.1 Trying verification with classical solvers

Theoretical results

Before launching verification on neural networks, let us recall a few things.

Recall that neural networks are functions that are composed by linear operations and activation functions. The most common activation function in modern neural

networks is the REctified Linear Unit (ReLU): $x \in \mathbb{R} :\to \max(x, 0)$. This function is *piecewise linear*: it is linear on ranges $]-\infty, 0]$ (where it is equivalent to the constant 0 function) and $[0, \infty[$ (where it is equivalent to the identity function).

When encountering piecewise linear functions, most standard SMT and LP solvers perform what is called a case-split. For a variable $x \in [\underline{x}, \overline{x}]$ and $z = \text{ReLU}(x)$, the following cases are possible:

1. $\underline{x} > 0$, then $z = x$;
2. $\overline{x} \leq 0$, then $z = 0$;
3. $\underline{x} < 0$ and $\overline{x} > 0$, then the solver splits into two subdomains $[\underline{x}, 0]$ and $[0, \overline{x}$ and continue on those two subproblems;

The last case adds another problem for the solver. If no particular optimizations are applied (which is the case for solvers like Z3 and Gurobi) and the range of variables is wide enough, a pessimistic upper bound on the number of possible cases is then $2^{\#ReLU}$, corresponding to the case where every neuron is split in its two possible activation states. Furthermore, in [Kat+17], it is proven that formally verifying a deep neural network is an NP-complete problem.

Experimental assertions

Theoretical results seem to point out that exploiting the linear regions for neural network verification is a dead-end. However, the $2^{\#ReLU}$ upper bound is far from being tight: the hypothesis behind its formulation does not take into account several characteristics of neural networks: namely, their organization in layers creates a dependency between neurons that is not captured with the $2^{\#ReLU}$ bound. Plus, problems that are known to be NP-complete can sometimes be solved practically, providing we use adapted tools and techniques (see for instance SAT solving). In order to assess whether classical tools would be able to work on deep neural networks, we used ISAIEH to translate neural networks and prove linear properties on toy neural networks with varying width and depth.

Experiments made during this thesis show that traditional solvers tend to time out on neural networks with about 10 neurons on an output property that is linear. Overall, Z3 performs the best, but is still unable to scale to realistically sized neural networks, even when limiting to ACAS-like neural networks (with about 300 neurons): modern architectures processing perceptual inputs have multiple order of magnitude more neurons, and are thus even more intractable for classical tools. We also wrote a converter from ONNX to C language, and used Frama-C's EVA abstract interpretation analyzer on the resulting C code. Compared to the lab specialized abstract interpretation analyzer, PyRat, EVA was about 100 times slower.

Without any help, classical SMT or LP solvers are bound to fail; more generally, there is a need to adapt existing techniques to the specificities of neural networks since classical tools tend to be slower.

7.2 Correctly leveraging facets

Recall that ReLU neurons have two linear pieces to consider: when the input is negative, and when the input is (strictly) positive. The linear functions here are quite simple to add into a solver with linear solving capabilities. The issue here is the sheer number of linear problems rather than their complexity.

However, there are elements in the literature that bring hope. Specifically, a few theoretical contributions have brought interesting insights on the *practical* number of possible problems.

In [HR19a], the authors present an upper bound on the number of linear regions that is exponential in the dimension of the input rather than in the number of neurons. For a neural network with N total ReLU, L layers, n_l ReLU for layer l, n_0 for the input dimension and $\#facets$ number of facets:

$$\#facets \leq \frac{N^{n_0}}{n_0!}$$

In the particular case where each n_l has the same value k (equal width for all the layers), another upper bound is proposed in [Rag+17]:

$$\#facets \leq \mathcal{O}(k^{Ln_0})$$

Authors of [STR18] propose another upper bound. Let $j_l \in \mathbb{Z}$ such that $0 \leq j_l \leq \min(n_0, n_1 - j_1 \dots n_{L-1} - j_{L-1}, n_L)$. Then:

$$\#facets \leq \sum_{j_1}^{j_L} \prod_{l=1}^{L} \binom{n_l}{j_l}$$

Finally, the work presented in [Urb+19] uses linear regions to verify fairness properties by assessing them on multiple linear regions at once, leveraging parallelism. If the number of linear regions is far lower than pessimistic theoretical upper bounds, is it possible to use them to change how we formally verify neural networks?

This chapter is a contribution to be submitted, that aims to address the following research questions:

Research questions

Can we leverage the sub-exponential upper bound in the number of facets for formal verification? Is the verification facet by facet faster than verifying them independently? Can we identify features in linear regions that make formal verification more amenable?

7.3 Background

Activation vectors and facets

Let $\mathcal{X} \subset \mathbb{R}^{D_{in}}$ be a multidimensional input space, let $\mathcal{Y} \subset \mathbb{R}^{D_{out}}$ be a multidimensional output space. Let f be a trained neural network of L layers, computing values from $\mathcal{X}$ to $\mathcal{Y}$: $f : \mathcal{X} \to \mathcal{Y}$. Each layer computes a multidimensional input and produces a multidimensional output, both represented as multidimensional arrays (also known as tensors). Each cell of a tensor is called a neuron. A layer l_i has an input in $\mathbb{R}^{D_{(i-1)}}$ and an output in $\mathbb{R}^{D_i}$, for $i = 2..L$, with $D_1 = D_{in}$ and $D_L = D_{out}$. In the rest of this thesis, we will denote a layer by l to avoid cluttering.

We consider here a network for which each layer l is composed of a linear application, followed by a ReLU activation function on all the resulting neurons. Parameter tensors are obtained after training and do not change while using the resulting program. The only variables are the inputs living in $\mathcal{X}$. As a running example for the rest of this chapter, we shall take the neural network described in figure 7.1.

For a given input $x \in \mathcal{X}$, each neuron of the layer l can be either *active*, if their value before the application of ReLU is greater than 0, or *inactive* when this value is stricly lower than 0. We denote by $\mathcal{S}^l_{\mathcal{F}}$ the activation state of ReLU neurons for a given layer l: an active neuron is denoted by 1, an inactive neuron by 0. As an example, for the network in figure 7.2, $\mathcal{S}^1_{\mathcal{F}} = (1, 1, 0)$, and $\mathcal{S}^2_{\mathcal{F}} = (1, 0)$. $\mathcal{S}^l_{\mathcal{F}}$ can also be seen as a vector of dimension $\mathbb{R}^{D_i}$.

We call a *facet* the subset $\mathcal{F}$ of the input space generating a certain activation pattern $\mathcal{S}^l_{\mathcal{F}}$. The network yields the same activation pattern for all inputs within this region. Such a facet describes a linear region, because all ReLU have a fixed behaviour within it; thus the network with inputs reduced to $\mathcal{F}$ is simply a composition of linear applications. In that case, the output vector y is obtained by element-wise multiplication between the pre-activation vector and $\mathcal{S}^l_{\mathcal{F}}$. For example, in the network in figure 7.1, for all inputs in $\mathcal{F}$, y_1 and y_2 are inactive, while y_3 is active. Thus, the final output of the network is $z * \mathcal{S}_{\mathcal{F}} = (0, 0, z_3)$.

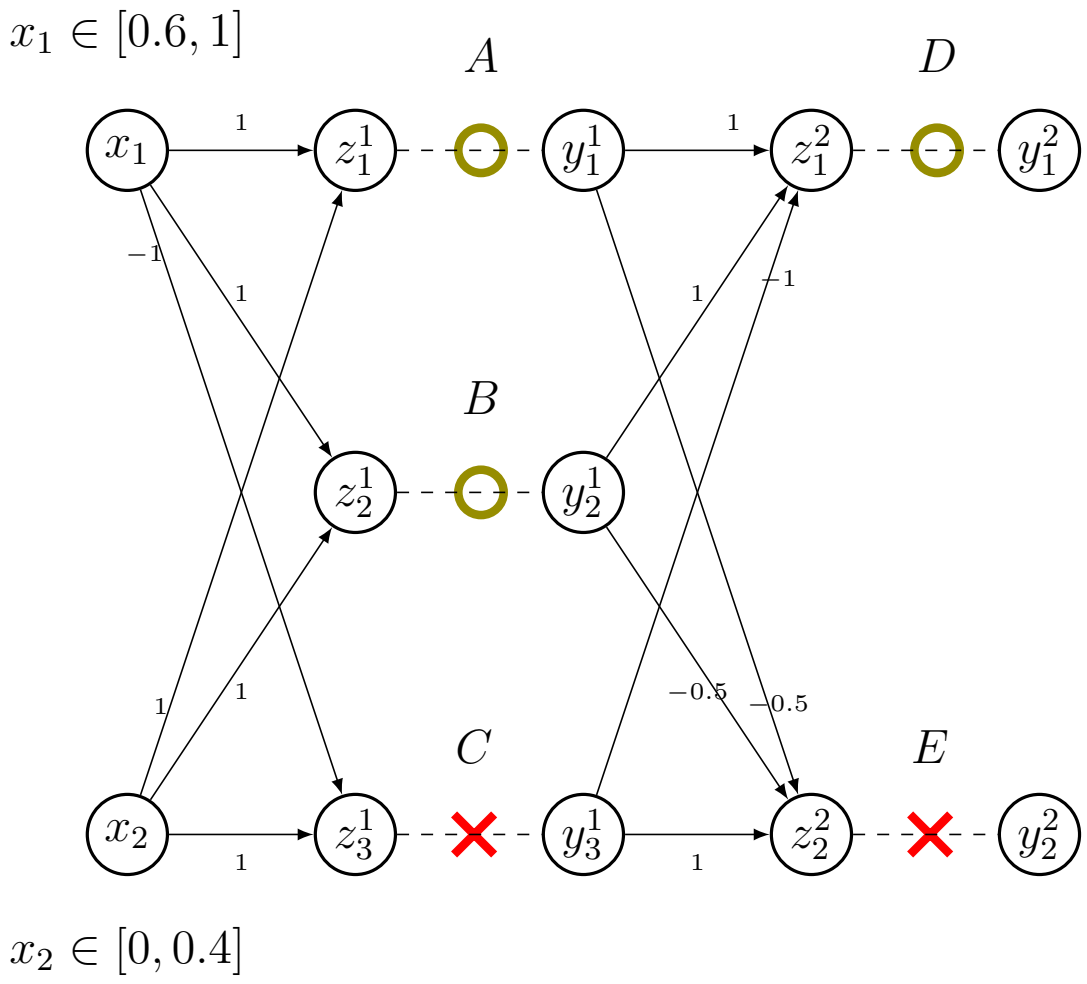


Figure 7.1: A two layered fully-connected network. Weights are indicated on edges. Green circle denotes an active ReLU neuron, while red cross denotes an inactive ReLU neuron. Both inputs x_1 and x_2 are positive. We see that the neuron y_3^1 is inactive, since the weighted sum $z_3^1 = -0.5(x_1 + x_2)$ is negative. Similarly, y_2^2 is inactive since $z_2^2 = -0.5(y_1^1 + y_2^1)$ is negative, y_1^1 and y_2^1 being positive because of being active neurons. The resulting activation states are $\mathcal{S}_{\mathcal{F}}^1 = (1, 1, 0)$ (first layer) and $\mathcal{S}_{\mathcal{F}}^2 = (1, 0)$ (second layer).

Building facets

Let z_i^l be the pre-activation value of a neuron at layer l, y_i^l the post-activation value. The value of this neuron is the result of a composition of affine transformations and ReLUs. If y_i^l is active, by definition of the ReLU, it means that $z_i^l \geq 0$. Similarly, it y_i^l is inactive, it means that $z_i^l < 0$. It follows that the activation state can be expressed as a linear constraint on the different neurons on the previous layer. For example, if the affine transformation in layer l is a matrix multiplication of elements $w_{i,j}^l$ with outputs

y_j^{l-1} of the previous layer, the linear constraint to express that the neuron is active is:

$$z_i^l = \sum_j w_{i,j}^l \, y_j^{l-1} \geq 0 \tag{7.1}$$

$$y_i^l = z_i^l \tag{7.2}$$

Similarly, an inactive neuron yields the constraint:

$$z_i^l = \sum_j w_{i,j}^l \, y_j^{l-1} < 0 \tag{7.3}$$

$$y_i^l = 0 \tag{7.4}$$

Each of the outputs of the previous layer y_j^{l-1} are themselves the result of a composition of affine transformations followed by ReLUs. We can thus write equations (7.1) and (7.3) for every previous layer, up to the input layer. Those equations define boundaries (*hyperplanes*) in the space of the current layer between active and inactive state. Thus, each ReLU neuron generates a constraint on the space of current layer. It is possible to write the activation constraints as depending solely on the input variables. Let us take the network depicted in figure 7.1, with two hidden layers, the first one having three neurons and the second one two. For the first layer, z_1^1, z_2^1 and z_3^1 are directly expressed as functions of the inputs:

$$z_1^1 = w_{1,1}^1 x_1 + w_{1,2}^1 x_2 \tag{7.5}$$

$$z_2^1 = w_{2,1}^1 x_1 + w_{2,2}^1 x_2 \tag{7.6}$$

$$z_3^1 = w_{3,1}^1 x_1 + w_{3,2}^1 x_2 \tag{7.7}$$

Expressing z_1^2 and z_2^2 with inputs depends on the activation $\mathcal{S}_{\mathcal{F}}^1$. First, we have:

$$z_1^2 = w_{1,1}^2 y_1^1 + w_{1,2}^2 y_2^1 + w_{1,3} y_3^1 \tag{7.8}$$

$$z_2^2 = w_{2,1}^2 y_1^1 + w_{2,2}^2 y_2^1 + w_{2,3} y_3^1 \tag{7.9}$$

For instance, if $\mathcal{S}_{\mathcal{F}}^1 = (1, 1, 0)$, then we have

$$y_1^1 = z_1^1$$
$$y_2^1 = z_2^1$$
$$y_3^1 = 0$$

Thus, equation (7.9) becomes

$$z_1^2 = w_{1,1}^2 y_1^1 + w_{1,2}^2 y_2^1 \tag{7.10}$$

$$z_2^2 = w_{2,1}^2 y_1^1 + w_{2,2}^2 y_2^1 \tag{7.11}$$

Reinjecting equation (7.7) in equation (7.11) finally gives

$$z_1^2 = (w_{1,1}^1 w_{1,1}^2 + w_{2,1}^1 w_{1,2}^2)x_1 + (w_{1,1}^2 w_{1,2}^1 + w_{1,2}^2 w_{2,2}^1)x_2 \tag{7.12}$$

$$z_2^2 = (w_{1,1}^1 w_{2,1}^2 + w_{2,1}^1 w_{2,2}^2)x_1 + (w_{2,1}^2 w_{1,2}^1 + w_{2,2}^2 w_{2,2}^1)x_2 \tag{7.13}$$

Note that equation (7.13) is only valid within the facet $\mathcal{F}$ defined by $\mathcal{S}_{\mathcal{F}}^1 = (1, 1, 0)$. A different activation state in the first layer would change the numerical coefficients. Recall that for our particular case, weights $w_{i,j}^l$ are known during analysis: the only variables are the input variables x_i and the result of propagation z_i^l.

More generally, given a facet $\mathcal{F}$, the corresponding activation states $\mathcal{S}_{\mathcal{F}}^{[1..L]}$, and an input vector X, we can obtain the hyperplane equations upto layer l by computing

$$\prod_{k=1}^{l} \mathcal{S}_{\mathcal{F}}^k W^k X \tag{7.14}$$

Geometric interpretation

We just explained how, given a facet, it was possible to express all hyperplanes equations in function of the inputs. Recall that each ReLU neuron generates one hyperplane in the input space; each side of this hyperplane is thus a subspace of the input space; each subspace defines where the neuron is active or inactive. The conjunction of the constraints of all ReLU neurons gives a facet. Since all constraints are linear, the result of this conjunction is a convex polyhedron.

Key point

A facet is the convex polyhedron described by the set of constraints resulting from a given activation pattern. A neural network with inputs restricted to one facet is a linear function.

Changing the activation state of one neuron means crossing a hyperplane on the input space. See figure 7.2 for a geometric illustration of this phenomenon.

7.4 Divide and conquer on linear regions

We aim to *formally verify* a neural network: given a network f, a precondition on the input space $\mathcal{D} \subset \mathcal{X}$ and a postcondition on the output space $\mathcal{P} \subset \mathcal{Y}$, we want to provably ensure that

$$\forall x \in \mathcal{D} \rightarrow f(x) \in \mathcal{P}$$

The form of the pre and postcondition varies according to the property we want to check. For instance, given $x \in \mathcal{X}, \forall \varepsilon < \varepsilon_0$, local adversarial robustness around a

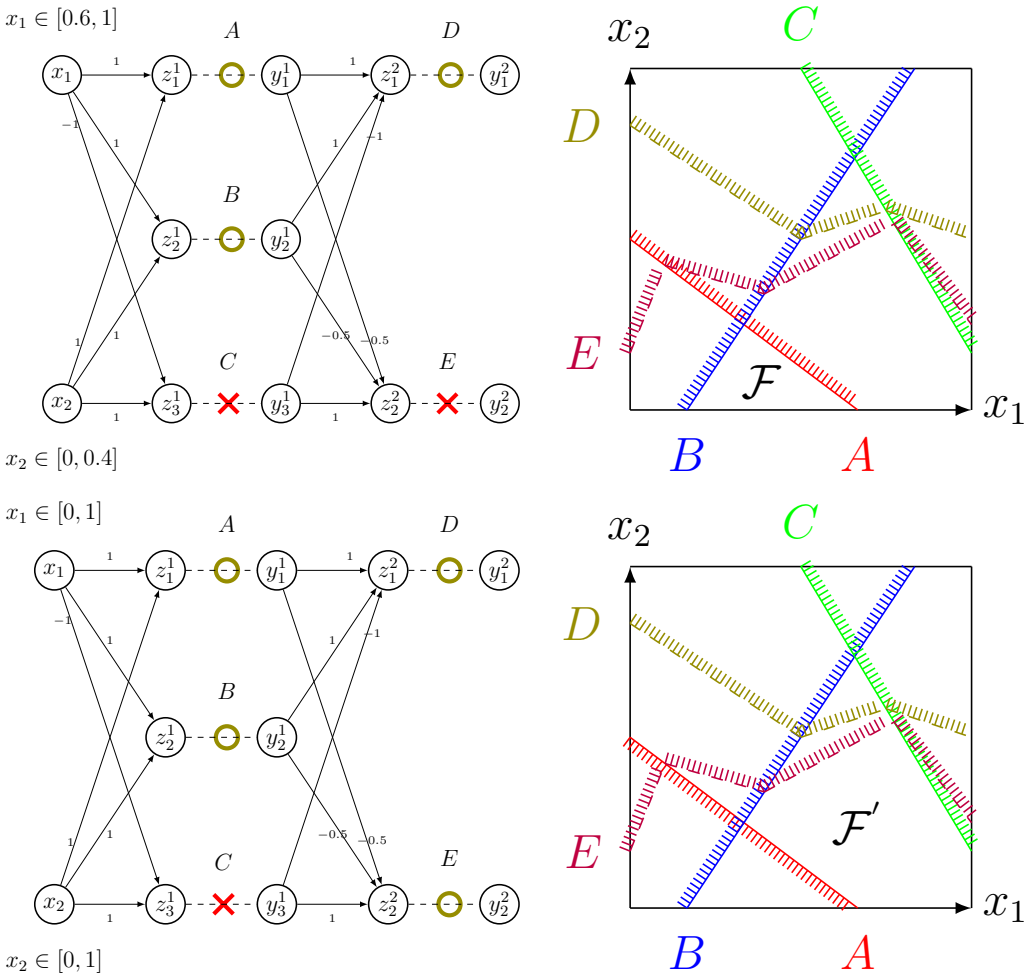


Figure 7.2: On the left: a network with activation states. On the right: the corresponding input space. On top: $\mathcal{S}^1_{\mathcal{F}} = (1, 1, 0)$. On bottom: $\mathcal{S}^1_{\mathcal{F}'} = (1, 1, 1)$. Changing the activation state results in a different linear region in the input space. Also note that all potential facets are convex.

sample would be expressed as $f(x + \varepsilon) = f(x)$ (note that $x + \varepsilon$ might be in a different facet than x). For safety properties of the ACAS benchmark, the preconditions on the inputs and outputs are linear constraints.

Linear operations are easier to verify than networks with ReLU, since they do not produce case splits on solvers. If we somehow have an exhaustive list of *actually reached* facets for our problem at hand, it would be possible to verify each facet independently. The naive estimation of the number of possible facets is exponential in the number of neurons: $2^{\#ReLU}$. This is because for each neuron, there are two possible activation states. However, literature shows that a network does not actually exploit the whole set of possible linear regions. Multiple works on expressivity [Rag+17] and

experimental counting of linear regions [HR19b], [STR18], [SR19] show that the actual number of reached linear regions is not exponential in the number of ReLU neurons. For a simple task, a deep and wide network could partition the input space into a few regions only. Some facets may also have specific characteristics, for instance having a wider support in the input space. Analyzing the number of facets and their characteristics is thus relevant.

On the other hand, we want to perform a sound and complete verification. Sound means that if our method answers that a system is safe, then it is actually safe; complete means that if a faulty behaviour exists for our problem, it will be spotted by our procedure. The key point is thus to exhibit a procedure to enumerate all the facets that are actually within the (constrained) input space, while excluding empty facets, *i.e.,* associated to activation patterns that are not realizable. In other words, we want to find all non-empty $\mathcal{F}_i$ such that $\bigsqcup_i \mathcal{F}_i = \mathcal{X}$.

Enumeration of facets

Our approach is to start from the beginning of the network and proceed neuron by neuron. Using an initial bounding box $\mathcal{D}$ as an initial constraint on the inputs, we iteratively build the linear constraints composing the neural network, as described in section 7.3. When a ReLU neuron y_i is considered, we aim to check whether it can be active, inactive or both, *considering the activation state of its predecessors.* We thus write the constraints corresponding to the activation states, and check if they are consistent with the previous activation states. A *facet* is thus a valid activation pattern for all the network: the conjunction of one possible valid activation for all ReLU neurons.

The active pattern yields the constraint described by equation (7.1):

$$\begin{aligned} z_i^l &= \sum_j w_{i,j}^l \, y_j^{l-1} \geq 0 \\ y_i^l &= z_i^l \end{aligned}$$

Similarly, the inactive pattern yields the constraint described by equation (7.3):

$$\begin{aligned} z_i^l &= \sum_j w_{i,j}^l \, y_j^{l-1} < 0 \\ y_i^l &= 0 \end{aligned}$$

If only one of the two activation states is possible, then the constraints describing this state are added to s, and the algorithm goes through the next neuron. If both activations are possible, then the problem stack is copied. Active constraints are added to the first copy, while inactive constraints are added to the second one. Since the two sub-problems are independent, this algorithm can be parallelized. Let us detail the algorithm on the neural network presented in figure 7.3.

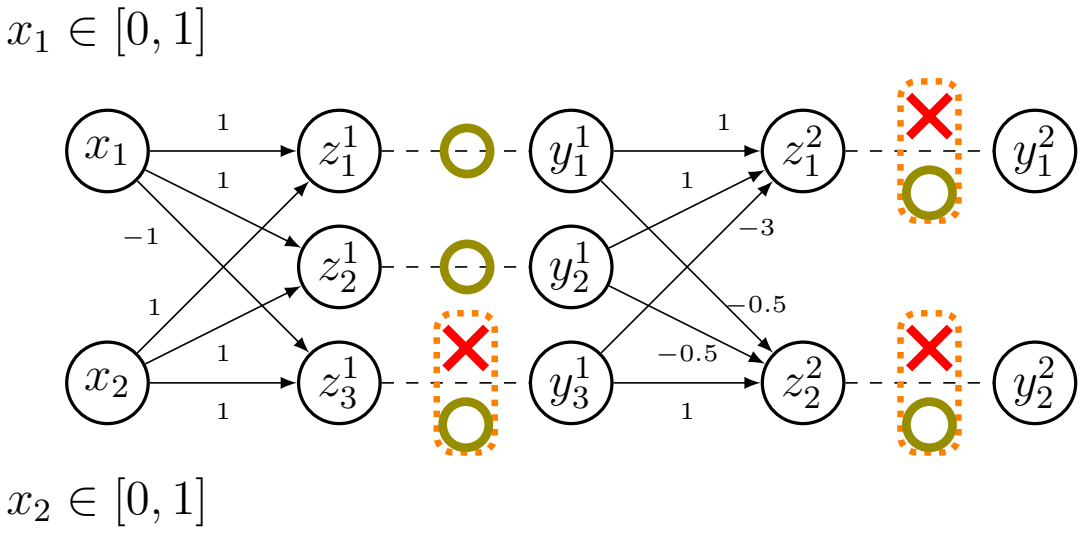


Figure 7.3: The same two-layered network, but with a wider input space. Here, possible activation states are $\mathcal{S}^1_{\mathcal{F}} = (1, 1, 0)$ or $(1, 1, 1)$ (first layer) and $\mathcal{S}^2_{\mathcal{F}} = (1, 0)$ or $(1, 1)$ (second layer).

Both inputs are positive. An initial constraint stack s^1 is filled with the constraints describing the linear operations occuring in the first layer, as well as the input constraints:

$$
\begin{aligned}
x_1 &\geq 0 \\
x_1 &\leq 1 \\
x_2 &\geq 0 \\
x_2 &\leq 1 \\
z_1^1 &= x_1 + x_2 \\
z_1^1 &= x_1 + x_2 \\
z_2^1 &= x_1 + x_2 \\
z_3^1 &= -x_1 + x_2
\end{aligned}
$$

Looking at the first equation, z_1^1 can only be positive. The solver answers negatively when asking it to solve what is currently on s^1, plus the following constraint, corresponding to the case where the neuron y_1^1 is inactive:

$$
\begin{aligned}
z_1^1 &\leq 0 \\
y_1^1 &= 0
\end{aligned}
$$

It means that the opposite constraints (y_1^1 is active) is the only possibility. We thus add to s^1 the following contraints

$$\begin{aligned} z_1^1 >& 0 \\ y_1^1 =& z_1^1 \end{aligned}$$

Similarly, z_2^1 can only be positive: we solve s^1 plus the inactive constraints set, the solver answers negatively so we add to s^1 the active constraints set.

$$\begin{aligned} z_2^1 >& 0 \\ y_2^1 =& z_2^1 \end{aligned}$$

For z_3^1, if $x_1 > x_2$, then the result is negative; and positive otherwise. Since the specification on our inputs does not give that information, both cases are possible. Indeed, the solver answers positively for both cases. The stack s^1 then is copied, obtaining a new stack s^2. We add to s^1 the following constraints, corresponding to the activation of y_3^1:

$$\begin{aligned} z_3^1 >& 0 \\ y_3^1 =& z_3^1 \end{aligned}$$

and we add to s^2 the following constraints, corresponding to the inactivation of y_3^1:

$$\begin{aligned} z_3^1 \leq& 0 \\ y_3^1 =& 0 \end{aligned}$$

We now have two stacks with different contraints: a new process is spawned to deal with stack s^2, while the initial process keeps s^1. On the second layer, we add to both stacks the linear equality constraints:

$$\begin{aligned} z_1^2 =& y_1^1 + y_2^1 - 3y_3^1 \\ z_2^2 =& -0.5(y_1^1 + y_2^1) + y_3^1 \end{aligned}$$

We then proceed to neuron y_1^2. Here, the possibilities depend on the stack we are in:

1. if the current stack is s^2, then $y_3^1 = 0$, hence $z_2^1 > 0$, thus y_1^2 can only be active; we add the corresponding constraints to $s^{'}$;

2. if the current stack is s^1, then $y_3^1 > 0$, thus z_2^1 can be both positive and negative ($z_1^2 = 5x_1 - x_2$); we must then copy the stack s^1 on a new stack s^3. s^1 contains the activation constraints for neuron y_1^2, while s^3 contains the inactivation constraints

We now have three constraint stacks: s^1, s^2 and s^3. Finally, on neuron y_2^2, another choice is possible:

1. if the current stack is s^1, then all neurons are active; z_2^2 can only be negative, thus the inactive constraint is added to s
2. if the current stack is s^2, then neuron y_3^1 is inactive; z_2^2 can only be negative again, thus the inactive constraint is added to s^2
3. since s^3 has the same constraints on previous layer than s^2, the result is the same

We finally have three different constraints stacks, whose content is displayed figure 7.4. See alg. 1 for a pseudo-code description.

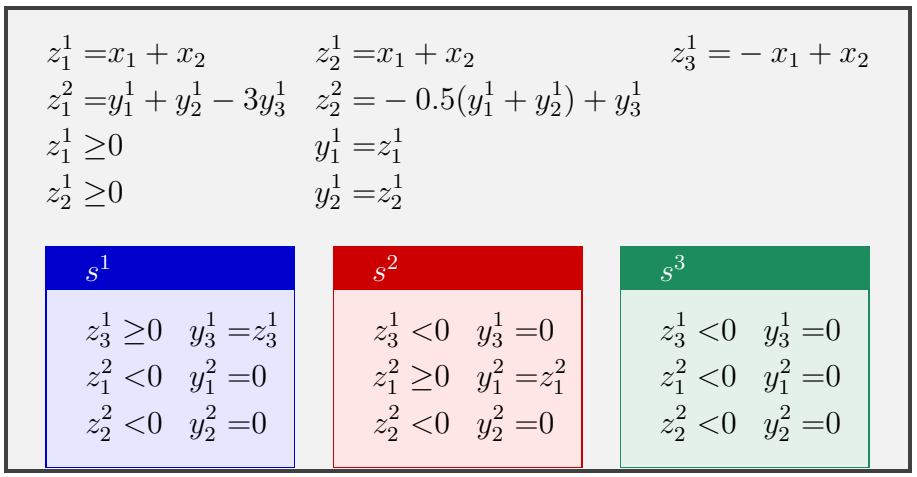


Figure 7.4: Content of constraint stacks; constaints above the three stacks are common to all of them

Once we obtain the set of all relevant facets, it is possible to build the corresponding linear functions. This set of linear functions represent all the possible behaviours of the network on its input space. Verification of the property can then be launched on each linear function; since they are independent problems: parallelization can also be used.

More formally, let us consider a facet set $\bigcup_i \mathcal{F}_i$ for a network f, an input space $\mathcal{X}$, an output space $\mathcal{Y}$, a precondition on the input space $\mathcal{D} \subset \mathcal{X}$ and a postcondition on the output space $\mathcal{P} \subset \mathcal{Y}$. We aim to formally verify that $x \in \mathcal{D} \implies f(x) \in \mathcal{P}$. Partitionning consists on adding to the network's control flow the constraint on the

inputs yielded by $\mathcal{F}_i$, and in enforcing the corresponding activation state for all ReLU neurons. The resulting function is thus a composition of linear operations: original matrix multiplications and active or inactive ReLU (which are diagonal matrices multiplied to the pre-activation inputs). Then, the verification problem becomes $x \in \mathcal{D} \cap \mathcal{F}_i \implies f(x) \in \mathcal{P}$. As facets are Dividing the Input Space into COnvex polytopes, we will be referencing our technique as DISCO in the rest of this thesis (not to be confused with [BKN16]).

7.5 Studies on facets

So far, we presented a methodology to use facets to ease formal verification. Some characteristics of those facets remain however unknown. What is the volume occupied by a facet in the input space? Are all facets activated uniformly? Which parameters influence the number of facets? In this section, we perform an analysis of the facets of several networks. These are fully-connected networks with three hidden layers, with various numbers of inputs. Architecture details are on table 7.1. Those networks are trained to perform multiplication between N floating point numbers sampled randomly according to a normal distribution between 0.5 and 2.

Towards counting facets and beyond

The initial motivation of this work was that the theoretical maximum number of facets was far over the actual number, and that it might be possible to leverage facets for formal verification. We observe that for our problem at least, this seems to be the case. We took the best performing networks in terms of accuracy and reported their number of facets in figure 7.5. The x-axis is the dimension d of the input, y-axis is the number of facets. Note that the number of neurons n directly depends on d as defined in table 7.1, hence the log-linear progression for both the naive and Hanin bounds. Most of the networks have about one or two orders of magnitude fewer facets than the bound proposed in [HR19b]: $K * \frac{n^d}{d!}$. This is an encouraging result, as it indicates that on multilayer feedforward neural networks, this upper bound is not tight and can be further refined. The progression seem sub-exponential on the input dimension; further experimentations are to be conducted to validate our hypothesis.

Not all facets are equal

Reducing the number of facets is a way to reduce the complexity of verification. When starting the verification, the solver will try each facet without prioritizing one over the other. This relies on the assumption that all facets are activated relatively evenly, that is to say, that each achievable facet has an equal chance to be activated by an

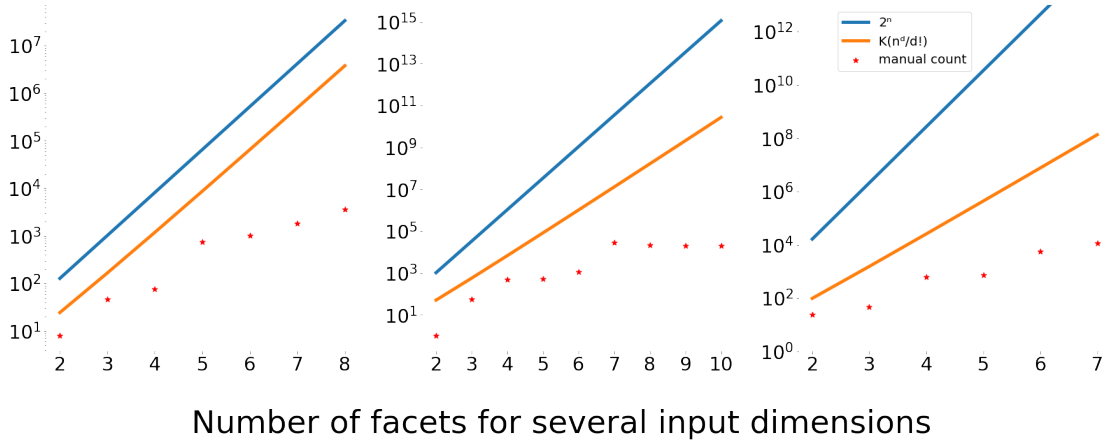


Figure 7.5: x-axis is the input dimension d. Upper orange line with dots is the naive, 2^n bound. Middle blue line with crosses is the bound proposed by [HR19b] $K * \frac{n^d}{d!}$. Red stars are the best performing networks for our experiments. Left figure is for the simple architecture, middle figure is for the big architecture, right figure is for the super architecture. y-scale is logarithmic

input point. However, if some facets were activated more frequently than others, this would mean that the neural network is biased towards certain input subspaces. For instance, in image recognition, the subspace of the inputs that contain images with a specific significant feature would be processed similarly by the network. Another point of interest is that prioritizing verification on those could result in obtaining a meaningful answer faster, since a wider part of the input space is within the facet. Also, the frequency of a facet's occurrence can be a good proxy to estimate the space occupied by the facet in the input space.

Thus, we plotted the distribution of points for each facet, as follow. We built a test set by uniformly sampling 10000 inputs on neural networks (according to a log-normal distribution between 0.5 and 2), and collected the number of points contained in each facet. This can be done without requiring an exhaustive enumeration of facets, as only facets reached by those 10000 points are considered. Results are available on figure 7.6. On this figure, x-axis denotes a unique facet, y-axis denotes the number of points that activated it. The y-scale is logarithmic. We first observe that points are not distributed uniformly on facets. For instance, for two inputs, the first most activated facet is activated by almost 10^4 points, while the second most activated facet is activated by less than 10^3 points, which means that the first facet captures almost all the sampled inputs, thus *the computation of most of the sampled inputs by the neural network can be reduced to the processing of a single linear function.* More generally, we observe that a few facets are activated by a relatively high fraction of the total sampled points. This validates the hypothesis that, at least on our problem, *facets are*

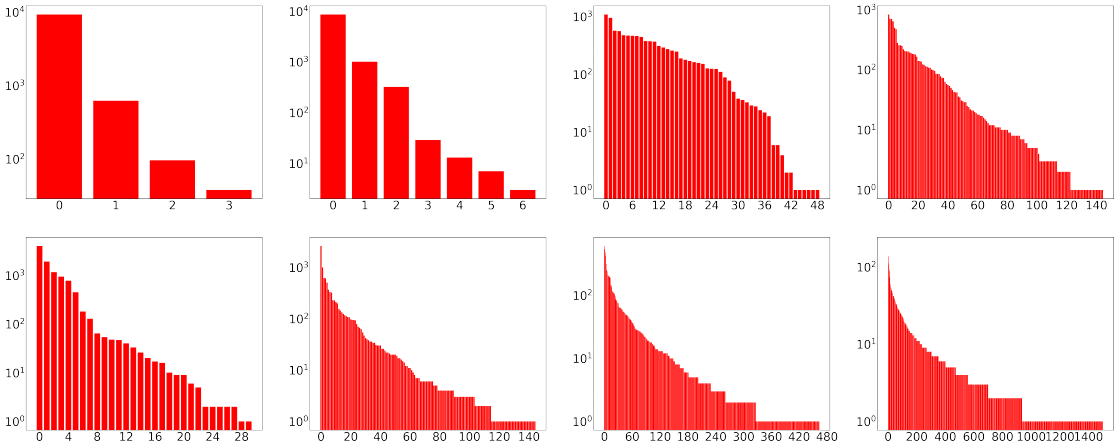


Figure 7.6: Distribution of points in facets for different input dimensions

not activated uniformly by the inputs. This is no surprise: complex non-linear functions approximated by neural networks over an input space $\mathcal{X}$ have no particular reasons to vary uniformly. To elaborate a bit more, let $\bigsqcup_i \mathcal{F}_i = \mathcal{X}$ the decomposition of $\mathcal{X}$ into facets. By slightly abusing notations, we denote by $\mathcal{F}_i$ the linear function represented by its corresponding facet. Finally, let us denote our neural network by $f : \mathcal{X} \mapsto \mathcal{Y}$. Having a uniform distribution of points within facets would mean that the support of f on each $\mathcal{F}_i$ should be of the same volume, which is less and less likely as the input dimension increases.

What makes facets shine?

Neural networks are the result of a complex optimization process. As the weights of a neural network directly influence its possible activation states, one may wonder if this optimization process has an influence on the number of facets. We postulate that, as stated in [Rag+17], a higher number of facets can be interpretated as a higher expressivity for the network, leading to better accuracy. To answer, we trained several neural networks of the same architecture with different seeds. We present on figure 7.7 a summary of all the experiments we made. We observe for instance that for the dataset 5 with 25 total neurons, some neural networks have less than 200 facets, while others have more than the double. More generally, changing the initialization seed greatly modified the number of facets for a given neural network, with extreme consequences on higher input dimensions. In contrast, changing the learning rate and the number of epochs did not result in a significant change in the number of facets compared to the initialization. Our hypothesis is that the training of small, underparametrized neural networks is very sensitive to the initialization, for optimization difficulty reasons. This

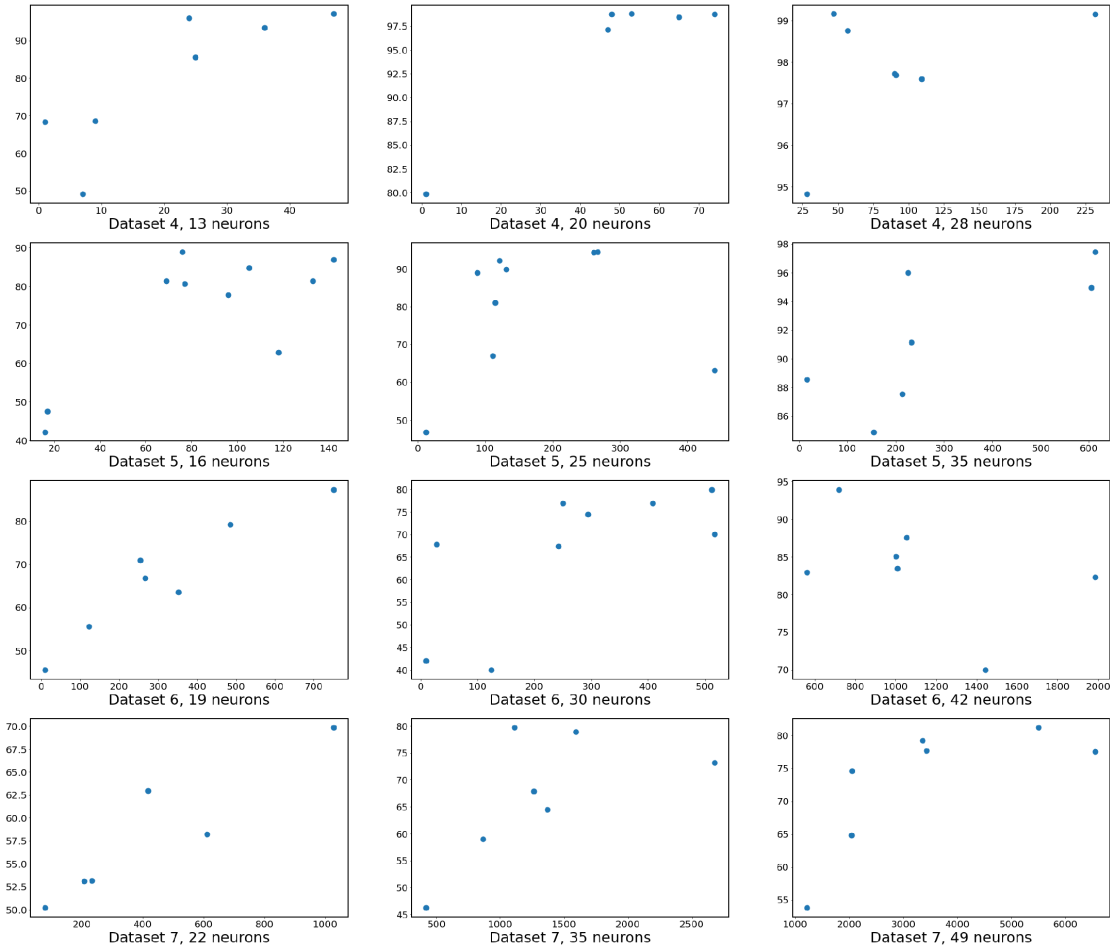


Figure 7.7: x-coordinate denotes the number of facets, y-coordinate the accuracy of the network

may also come from the low complexity of the function we are studying on the input space (multiplication of two real values on $[0.5, 2.0]$ is a saddle with small curvature). Adding more parameters to our neural networks to overparametrize them may lead to a better stability in the number of facets. We observe that a higher number of facets results in a higher accuracy: networks on the right of the x-axis (number of facets) tend to be on the top of the y-axis (accuracy). This is not the case for dataset 5 with 25 neurons. Apart from it, which could be an outlier due to the lack of accurate neural network for this specific dataset, this validates our hypothesis that the number of facets is correlated with the accuracy. To get an intuitive understanding of this fact, let us consider the two extreme cases. If the number of facets is very low compared with the input space dimension, it means that the neural network have very little different possible behaviours on the input space. For a classification, if the number of facets is below the number of classes, the task cannot be achieved. On the opposite, a neural

network with a very high number of facets compared to the input space dimension would theoretically be able to approximate the expected behaviour for each datapoint, which would result in a higher accuracy.

```
Data: An input space domain 𝒟, a list of all neurons in the network 𝒩
Result: A set of linear problems describing all feasible facets for the input
        space
// build the expressions for each neuron
lin_exprs, relu_neurons = BuildExpression(𝒩)
len = Length(relu_neurons)
stack = 𝒟
// the only shared resource between processes
facets = ∅
def SendToNewProcess(stack, index, facets):
    while index < len do
        // linear expressions describing the activation state for a given
           neuron
        neuron = relu_neurons [ index ]
        lin_expr = lin_exprs [ index ]
        stack.push(lin_expr)
        active_expr, inactive_expr = BuildConstraints(neuron)
        // test and propagate only the feasible activations
        if Solve(stack ⋂ active_expr) then
            if Solve(stack ⋂ inactive_expr) then
                stack_copy = stack.copy()
                stack_copy.push(active_expr)
                // send the copied stack to a new instance of the algorithm
                SendToNewProcess(stack_copy, index + 1, facets)
                // proceed in the current process with the other possible state
                stack.push(inactive_expr)
            else
                stack.push(active_expr)
            end
        else
            stack.push(inactive_expr)
        end
        index = index + 1
    end
    // when all neurons have been analysed, add the resulting linear
       constraints to the list of facets
    facets.append(stack)
end
// launch the algorithm on the first ReLU neuron
SendToNewProcess(stack, 0, facets)
return facets
```

Algorithm 1: Counting facets

name	L_1	L_2	L_3	Total number of neurons
simple	$d \times 2$	d	$d/2$	$3.5d$
big	$d \times 3$	d	$d/2$	$4.5d$
super	$d \times 4$	$d \times 2$	d	$7d$

Table 7.1: number of neurons for the different architectures. d denotes the dimension of the input, L_i the $i-th$ layer of the network

Chapter 8

Tackling the combinatorial problem in practice

Summary of the chapter

We present our implementation of DISCO on ISAIEH. To give a proper context, we present some of the most prominent tools of neural network verification, classified under two main categories: exact methods and overapproximative methods. Exact methods are able to provide optimal answers, but are difficult to scale to larger networks. Overapproximative methods provide certified upper and lower bounds on the reachable output for the neural network, which can be used to answer verification queries. Our approach of DISCO is quite simple, without any optimization. We perform faster than traditional SMT solvers on simple problems while LP problems do not seem to benefit from our approach. Implementing optimizations within our implementation is a first research track; an overapproximation scheme reducing the number of linear regions while preserving the semantic of the network is currently planned.

We described why leveraging linear regions for formal verification was a track worth considering. After showing that the number of linear regions was far below the upper bounds found in literature for our problems, we now present our implementation of the DISCO method and the result of our approach compared to state-of-the-art machine learning verification tools.

8.1 Existing tools for machine learning verification

In less than a decade, an impressive amount of research was undertaken to bring formal verification knowledge and tools to the field of adversarial robustness. We shall present

a fraction of those tools, using a taxonomy presented in [Bun+17]. This taxonomy separates existing tools between two main families: *exact* methods and *overapproximative* methods. We will present the tools in those two categories, and further refine this classification. Note that this taxonomy is also closely linked to how we presented formal methods in chapter 3.

Exact methods

The first category regroups all techniques relying on exhaustive exploration of the search space, or *exact* methods. Among such techniques are SMT calculus and LP optimization. Recall that SMT or LP rely on the same basic idea: the verification problem consists in deciding whether, for a given formula, there exists an instantiation of the variables that makes the formula true. Program properties and control flows are encoded as logical formulae, that specialized solvers try to solve. For SMT, it is possible to express precise properties such as elaborate logical constructs (conjunctions of disjunctions) and non-linear properties (multiplication between values). Since most solvers try to be exhaustive over the search space, a careful formulation of the constraints and control flow is necessary to keep the problem tractable.

SMT

Reluplex [Kat+17] and Planet [Ehl17] are the first techniques to aim for exhaustive verification of neural networks, using SMT calculus. They propose a reformulation of the simplex algorithm to lazily evaluate ReLU and branching heuristics such as case-splitting on individual neurons. Their work focus on the algorithmic method used to solve a non-linear, non-convex problem. ReLuPlex is an SMT solver integrating a modified simplex algorithm, that lazily evaluates ReLU, reducing the need to branch on non-linearities. The core idea is that Reluplex stores upper and lower bounds of every variable (inputs and outputs of each layer) at any time. A verification query will add constraints on inputs and outputs: the overall goal is to either reach a possible assignment for each variable regarding the query, or reach a point where no assignment exists. Each step of the algorithm is trying to modify the bounds of variable assignments in order to meet constraints; which leads to tighter bounds and new constraints to take into account. The process is proven to terminate on piecewise linear networks. Reluplex follow-up work, Marabou [Kat+19], improves and extends the tool to support more complex networks. They propose a *divide and conquer* approach where the solver starts with an initial small timeout on an input space: failure to reach a conclusive answer will split the input domain into smaller subdomain. Smaller subdomains reduce the search space and allow for more efficient deductive steps. The original paper does not detail what kind of splitting heuristic is used. Marabou also integrates

network-level reasoning, for instance by tightening the bounds of variables by propagating symbols.

Branch and bound

In [Bun+20], the authors rephrase the problem of adversarial robustness verification as a global optimization problem on which we can apply various solving strategies. The key point of their approach is to split the search space into multiple branches to find optimal bounds with a reasonable time budget: hence the name *branch and bound* (B&B).

In their framework, a ReLU neural network verification problem can be encoded by the following:

$$\underline{x_0} \leq x_0 \leq \overline{x_0} \tag{8.1}$$
$$x_n \leq \overline{x_n} \tag{8.2}$$
$$z_{i+1} = W_{i+1}x_i + b_{i+1} \tag{8.3}$$
$$x_{i+1} = \max(z_i, 0) \tag{8.4}$$
$$\tag{8.5}$$

Their goal is to assess the sign of the output, using linear optimization. To increase the efficiency of this approach, they propose new branching heuristics:

1. on input domains, they propose to split the domain on the highest dimension
2. on ReLU neurons, they propose to prioritize splits that may lead to tighter lower bounds

Their recommendation is to focus on designing new splitting heuristics for both ReLU neurons and input domains, which is exactly what we do with DISCO. We lacked the time to implement their proposal on DISCO; we believe that doing so would lead to even better performances.

Mixed Integer Linear Programming

The verification of adversarial robustness properties on piecewise linear networks can also be formulated as a MILP problem, such as in [TXT19]. This encoding is the one we used for neural networks on the LP side. Encoding the linear operations is similar to equation (8.5). Their ReLU encoding is the only key difference; we reproduce it below. We can encode the ReLU operation $y = max(z, 0)$ like this:

$$y \leq z - \underline{z}(1 - a)) \wedge (y \geq z) \wedge (y \leq \overline{z} * a) \wedge (y \geq 0) \wedge (a \in 0, 1)$$

They make use of interval arithmetic and linear programming to compute tight bounds of the linear relaxations of ReLU neurons, vastly improving the scalability of verification. Adversarial robustness properties were thus checked on ResNets (an architecture considered quite deep, with about 16 layers) with l_∞-bounded perturbations on CIFAR-10.

Performances

Here, we present a synthesis of performances of the selected tools. Performance is evaluated regarding the speed of different methods and, for adversarial robustness, on the accuracy of bounds. A summary is presented table 8.1.

Benchmarks are the following:

1. proving ACAS properties as described in chapter 3; unless stated otherwise, this is the ϕ_2 property, which is one of the hardest;
2. assessing that all inputs of a neural network trained on MNIST are correctly classifed around an l_∞ ball (MNIST);

Apart from B&B, most tools used different input formats and were not easily tailored to work on different networks and benchmarks. Methods worked on other benchmarks than those presented here; we selected results that seemed representative of the performance of the presented tools.

Overall, the MILP approach supports the widest range of threat models, while being able to compute lower bounds and exact values for adversarial robustness. MILP tend to be slightly less expressive than SMT approaches to express properties: for instance, conjunctions of disjunctions like the ϕ_2 property of ACAS are difficult to express, and non-linear properties are impossible to prove directly. However, the authors are able to find the minimal perturbation above which misclassification is possible, where B&B only assess the robustness on existing perturbation settings. The B&B reformulation and subsequent optimizations proved to be the most efficient of approaches, in terms of range of verifiable properties as well as runtimes.

Overapproximation methods

The second set of techniques in formal methods is based on overapproximating the program's behaviour. Indeed, since solving the exact verification problem is hard, some authors worked using techniques building overapproximations of the program, on which it is easier to verify properties. This usually results on guaranteed lower bounds on the perturbation around a set of points (usually the dataset). Most work in this line of work also devised techniques to enhance the adversarial robustness of their networks, based on the bounds they were able to compute. A summary of the performances of those tools is available table 8.1.

Method \ Benchmark	ACAS	MNIST adversarial robustness
Marabou[Kat+19]	1200s (average) for one property	–
Planet[Ehl17]	on par with Marabou on simple instances, ≈ 1000 times slower on difficut instances	106.8s ($l_\infty 0.08$ threat model)
MILP[TXT19]	–	≈ 100s to find the exact minimal distortion
B&B[Bun+20]	98% of properties solved in one hour	1200 cases solved in about 120s

Table 8.1: Runtimes for different benchmarks as reported in the original publications

Abstract interpretation

A line of work from ETH Zurich focused on adapting abstract interpretation to deep neural networks, for instance in [MGV18; Sin+19; SG19] [1]. In those papers, the authors propose and enrich ERAN (ETH Robust Artificial Intelligence), a framework for building abstract interpretations of neural networks, which they use to derive a tight upper bound on robustness for various architectures and for regularization. They also combine with exact methods to compute tighter bounds, such as in [SG19]. They propose abstract transformers for the most common operations of deep learning, and use those to verify local adversarial robustness on CIFAR-10 and MNIST, as well as to prove ACAS properties.

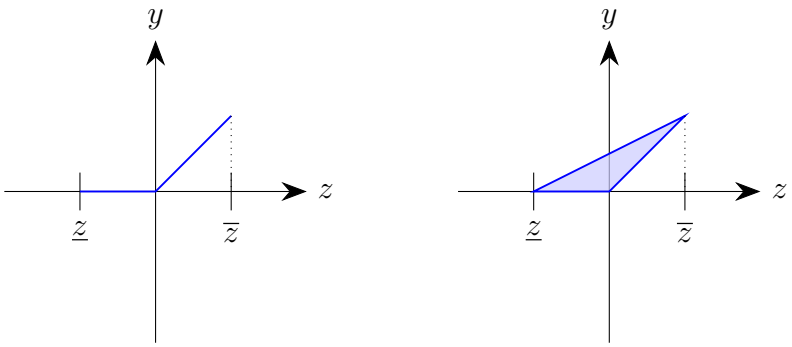
Upper and lower bound propagation

The first instance of specifically-tailored deep learning verification described how to refine non-linear sigmoid activation function to help verification [PT10].

The authors of [WK17] proposes an outer convex envelope for ReLU classifiers with linear constraints, expressing the robustness problem as a Linear Programming (LP) problem. Their approach, shared by most of the authors in this line of work, is to overapproximate the output of a ReLU $y = relu(z)$ neuron by its convex hull (see figure 8.1).

They choose to maximize the dual problem of the original verification problem problem to achieve a lower bound; the dual formulation being easily solvable. Up-

[1] full list of publications available at `https://github.com/eth-sri/eran#publications`, last consulted 2021/07/12



ReLU convex relaxation of reachable set

$$y \geq 0, y \geq z, -\overline{z}z + (\overline{z} - \underline{z})y \leq -\overline{z}\underline{z}$$

Figure 8.1: Linear overapproximation of y =ReLU(z). Note that it includes negative values, which would normally not be possible for the output of a ReLU.

per and lower bounds on each neuron are computed by a backward propagation on the neural network of the original constraint. Once those bounds are computed, the authors use it during training to identify "worst case" perturbations and train neural networks to detect wrong cases.

The authors of the tool Reluval [Wan+18b] introduce symbolic interval analysis and various heuristics applied to it. Inputs of the neural network are symbolic values with bounds, that are propagated through the network. Through careful heuristics (for instance, leveraging parallelism with interval bisection), they are able to derive output bounds on neural networks. The following work, Neurify [Wan+18a], combines this approach with targeted overapproximation to further enhance scalability.

CNN-Cert [Boo+19] is a framework for neural network verification developped at IBM. Its bound propagation follows a forward approach, by proposing tight boundings for every non-linear operation of the neural network (much like abstract interpretation approaches).

Tools leveraging linear regions

Regarding linear regions, a theoretical extension of the universal approximation theorem applied to robustness certification was proposed in [BMV20]. An exact enumeration scheme was proposed by [STR18] using MILP. Our enumeration scheme closely follows theirs, with some additional heuristics; we also leverage the obtained linear regions to perform formal verification, while they do not. They also provide initial insights by showing a correlation between accuracy and the number of facets.

Method \ Benchmark	ACAS	MNIST	CIFAR
Outer Polytope[WK17]	–	5.81%, $\varepsilon = 0.1$ robustness bound	–
ReluVal[Wan+18b]	15632s for ϕ_9	4.4% test error on MNIST for $l_\infty < 1$	
ERAN[SG19]	227s for ϕ_9	15% robust test error, 35s mean runtime, $\varepsilon = 0.03$	$\varepsilon = 0.1$, 79% robust error, 20s mean runtime
CNN-CERT[Boo+19]	–	0.049 certified lower bound, 2.33s for one sample	0.0042 certified lower bound, 15.11s for one sample

Table 8.2: Runtimes for different benchmarks as reported in the original publications

Finally, our work is closely related to [Bak+20], where authors propagate linear constraints within neural networks to check formal properties on fully-connected deep neural networks. They use numerical domains to propagate more information than we do, namely upper and lower bounds of variables within each linear regions. They are also able to overapproximate their propagated set, although this makes their method not complete. In contrast, our path enumeration is always sound and complete, and only needs to be called once to verify any property afterward.

Performances

Most of the following methods are properly scaling to more complex datasets and wider architectures. As such, another comparison can be made on the CIFAR-10 dataset.

Overall, CNN-CERT and ERAN are the tools that support the most operations (recent work on ERAN [Bon+21] even try to tackle transformer architectures), while the outer polytope and ReluVal approaches are behind in terms of support. ERAN is faster on deeper networks. Another tool, [Tra+20], implements various abstract domains, as well as a new star set. They evaluate their work on ACAS-Xu properties ϕ_3 and ϕ_4, and they report results consistently slower than ReluVal on those instances.

8.2 Efficiently implementing DISCO in ISAIEH

To count facets, it was necessary to be able to modify the activation state of a ReLU neuron. To do so, we devised an operator, outside ONNX scope, that replace the ReLU operator with an identity followed by a multiplication by 0 or 1 according to the operator parameters. This binary vector is the direct implementation of the layer-wise activation state $\mathcal{S}_{\mathcal{F}_i}$. Then, a forward propagation is conducted, following the algorithm described in alg. 1 and creating a Problem value. Problems are a list of linear constraints attached to NIER operators. A problem can be solved using LP or SMT

technologies, or directly written down into SMT or LP format for later use. To take into account the various solving technologies, a Solver functor is defined. Functors are OCaml implementation of polymorphism: they provide functions from modules to modules, providing the input modules respect a certain interface specification. Here, the function SolverModule takes a SolverTech module which must implement an instanciation function, as well as a string representation. The resulting module has the ability to solve a problem, as well as to write it down under a string representation. A Problem is encoded as follows:

```
module Constraint : sig
  type variable = Var of string
  type value = float
  type t = Hashtbl (variable, value)
end
module Problem : sig
  type b = ApplyReLU | ApplyRW_ReLU of NIER.operator_parameters | ApplyNothing
  type t = {constraints : (Constraint.t*b) list,
      variables : Constraint.variable set
  }
  val from_nier : NIER.t -> t
  val add : t -> Constraint.t -> t
end
module Solver (Target : SolverTech) : sig
  type res : OK | KO
  val solve : constrs -> Target.t -> Target.opts option -> res
end
module SolverTech : sig
  type t
  type opts
  val instanciate : Problem.t -> opts -> t
  val to_string : Problem.t -> string
end
```

Parallel implementation of DISCO

Since DISCO is implemented in OCaml, direct multicore implementation was difficult, as the language was not straightforwardly supporting parallel implementation at the time: although there exists an implementation of multicore OCaml, preliminary experimentations using Python shown that parallelism brought quantifiable, but limited improvements; the engineering work of implementing DISCO alone was complex enough.

name	L_1	L_2	L_3	Total number of neurons
simple	$d \times 2$	d	$d/2$	$3.5d$
big	$d \times 3$	d	$d/2$	$4.5d$
super	$d \times 4$	$d \times 2$	d	$7d$
perception	$d/2$	$d/4$	–	$0.75d$

Table 8.3: number of neurons for the different architectures. d denotes the dimension of the input, L_i the $i-th$ layer of the network

8.3 Experimentations

This section describes analysis and performance assessments made on our implementation of DISCO. On all settings, neural networks are trained with PyTorch [Pas+19] and exported into ONNX using PyTorch dedicated functions. Unless specified otherwise, neural networks weights are initialized using Glorot initialization with a gain of 1, and trained using the PyTorch implementation of the Adam optimizer. Neural network training took place on the lab GPU cluster, Titanic, a SLURM-powered cluster. Experiments using formal verification tools are conducted on a Dell Precision 5530 with an Intel Core i7-8850H CPU, 2.6Ghz, and Ubuntu 20.04.1 LTS as operating system.

DISCO

We implemented DISCO in OCaml, within ISAIEH. ISAIEH performs symbolic propagation to compute the hyperplanes delimiting facet boundaries during a forward pass, the building of facets is then made according to alg. 1. The linear programming implementation was made with the Python programming language, and Gurobi [Gur21] was used as an LP solver (Version 9.1.1). For the SMT verification, z3 [dMB08] was used (Version 4.8.10).

We consider the two synthetic problems we already used before:

1. detection of the presence of an obstacle within a given area, already presented in chapter 6; this problem will be *N-perception* for the rest of this thesis

2. multiplication between N floating points numbers, already presented in chapter 7 ; this problem will be called *N-multiplication* for the rest of this thesis

For those problems, we study different architectures. All of them are fully-connected networks. $N-multiplication$ networks have three hidden layers, $N-perception$ ones have two hidden layers. Details are on table 8.3.

For each of the two problems, we aim to count the number of facets, then verify whether the network respects its specification. For $N-multiplication$, we check

whether the network can indeed produce multiplication results within the tolerance. As formulating this problem directly is impossible due to linear programming limitations, we instead check if following property is verified:

$$\sum_{k=1}^{N} x_k + 1 - \frac{5}{4}N + \alpha_N \quad \leqslant \quad \prod_{k=1}^{N} x_k \tag{8.6}$$

with $\alpha_N = 0$ if the input dimension N is even and $\alpha_N = \frac{1}{4}$ otherwise. This property should be always true for our input space $[0.5, 2]^N$. A proof of this inequality can be found in part V. For $N - perception$, we check the following two properties:

1. if an input with at least one obstacle (modeled as white pixel) in the lower half of the image is presented to the network, the output will always be over 0
2. if an input with no obstacle on the lower half of the image is presented to the network, the output will always be below 0

See table 8.4 for results. For each network, the first column describes the runtime of verification without our technique, while the second column describes the runtime of verification on all facets. To be fair, the runtime of the enumeration scheme is also noted in the third column. Solving with DISCO or with standard MILP formulation always returns the same result. Note however that the splitting in linear regions is independent of the verification problem: costly enumeration algorithms could be used to obtain the facets of a neural network once, then verification could happen afterward. A violation of property in one facet stops the verification and returns the failure, in a naive "fail-first" heuristic. Preliminary experiments on networks with a high number of facets show that failures are detected early: guiding the search with a fail-first heuristic would prove even more useful. Chosen networks are those with the maximum accuracy, with similar architectures. $N - multiplication$ problems were solved using Linear Programming, while $N - perception$ problems were solved using SMT, QF_LRA theory. We note that the speed-up for the problem verification is much higher with SMT than LP. A possible explanation is that the number of facets with $N - multiplication$ being much lower than in $N - perception$, the additional cost of counting and parallelizing verification on each facet is not worth the effort. It is also likely that Gurobi performs specific heuristics allowing to efficiently deal with piecewise linear problems.

Formally proving a property using DISCO requires to enumerate all possibly reachable facets. Even if their practical number is far below theoretical upper bounds, any existing method reducing it is worth studying. Such a method exists: maximum margin regularization (MMR), presented in [CAH19]. The authors propose to modify the learning objective of the neural network to maximize the distance between a sample and nearby facet boundaries. Neural networks tend to “push away” the boundaries,

Dimension of input	No split	DISCO verification	Facet enumeration	Total time DISCO
3 super	0.769s±0.0205	**0.145s**±0.012	2.69s±0.0596	2.83s
3 super mmr	0.498s±0.00295	**0.184s**±0.0142	1.86s±0.0142	2.05s
4 big	0.25s±0.00423	**0.0972s**±0.00764	0.663s±0.0156	0.76s
4 big mmr	**0.454s**±0.0104	1.43s±0.0444	16.9s±0.0931	18.3s
4 super	5.43s±0.31	**0.71s**±0.0591	13.1s±0.859	13.8s
4 super mmr	3.69s±0.133	**2.77s**±0.174	35.7s±1.41	38.4s
5 simple	**0.0179s**±0.00596	0.0771s±0.0077	0.699s±0.0124	0.776s
5 simple mmr	**0.0204s**±0.00084	0.346s±0.0174	3.75s±0.0581	4.09s
5 big	**0.0279s**±0.00148	1.31s±0.0622	17.4s±0.283	18.7s
5 big mmr	**0.0154s**±0.000531	1.48s±0.0513	18.8s±0.0867	20.3s
6 simple	**0.0264s**±0.00124	0.988s±0.0693	11.6s±0.186	12.6s
6 simple mmr	**0.0291s**±0.00132	1.3s±0.0342	16s±0.149	17.3s
7 simple	**0.0474s**±0.00158	16.8s±0.831	227s±8.51	244s
7 simple mmr	**0.0306s**±0.0016	1.09s±0.0348	15.6s±0.555	16.7s
8 simple	**0.0484s**±0.00551	1.65s±0.113	27.2s±0.576	28.8s
8 simple mmr	**0.12s**±0.00269	1.72s±0.0988	28.9s±0.697	30.6s
5×5 perception	132s	23.7s	0.86s	**24.56s**
7×7 perception	TIMEOUT	1393s	15.38s	**1406.38s**

Table 8.4: Runtime for different problems. TIMEOUT is set at 10000s. Figures are mean taken over 10 runs, standard deviation is reported next to the $\pm$ symbol

resulting in fewer facets for a fixed $\mathcal{X}$. More formally, let us consider a facet $\mathcal{F}_i$. This facet is neighbored by k others, leading to k boundaries. Each of those boundaries are hyperplanes yielded by $\mathcal{F}_i$ and its neighbours, their equation can then be written as $V^k_{\mathcal{F}_i}$. Here, $V^k_{\mathcal{F}_i}$ is the orthogonal vector to the hyperplane constituting the k^{th} boundary with $\mathcal{F}_i$. For any sample s within $\mathcal{F}_i$, the distance between s and a hyperplane defined by $V^k_{\mathcal{F}_i}$ is $\langle V^k_{\mathcal{F}_i}, s\rangle$ (where $\langle\cdot,\cdot\rangle$ denotes the scalar product). In their paper, they compute this distance and aim to maximize it. Another distance towards decision boundaries is also computed, but since we focus on regression tasks, the notion of decision boundaries is not relevant here. The final term added to the cost function of the network is then, with γ_{rb} a parameter and p either 1, 2 or ∞:

$$\max\left(0, 1 - \min_k\left(\frac{\langle V^k_{\mathcal{F}_i}, s\rangle +}{\left\|V^k_{\mathcal{F}_i}\right\|_p} * \frac{1}{\gamma_{rb}}\right)\right) \tag{8.7}$$

We reimplemented their method and applied DISCO to networks trained with MMR, for $N-multiplication$ problems. We trained 10 neural networks with various degrees of MMR, and compared their accuracies and number of facets. Results are available table 8.4 and figure 8.2. We performed an additional experiment by training 30 networks on $4-$multiplication. γ_{rb} was set to 0.01, 0 (no MMR enabled), and 100. 10 networks

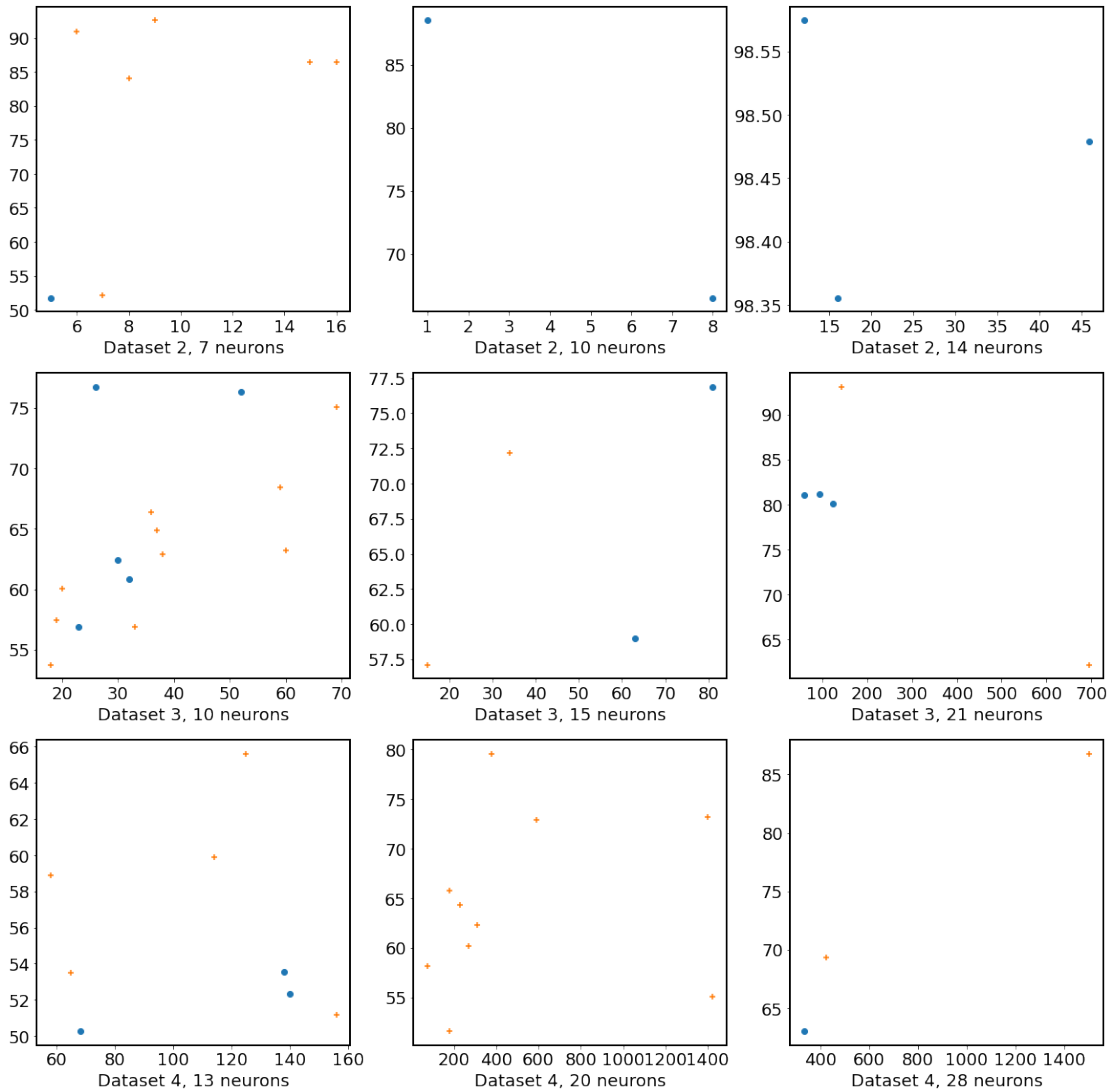


Figure 8.2: Graph summing up the performances of several networks. x-coordinate denotes the number of facets, y-coordinate the accuracy of the network. Orange crosses are networks trained with MMR, blue dots are networks trained without.

were trained for each value of γ_{rb}, with different seeds. Results are available figure 8.3.

We noted no difference between neural networks trained with MMR and without it; a possible explanation is that our neural networks are too small, which leads to a greater variety in initialization states: MMR effects would then be negligible compared to those. On figure 8.3, we observe that neural networks first struggle to perform above 50% of accuracy with less than 50 facets, then their performance enhance dramatically

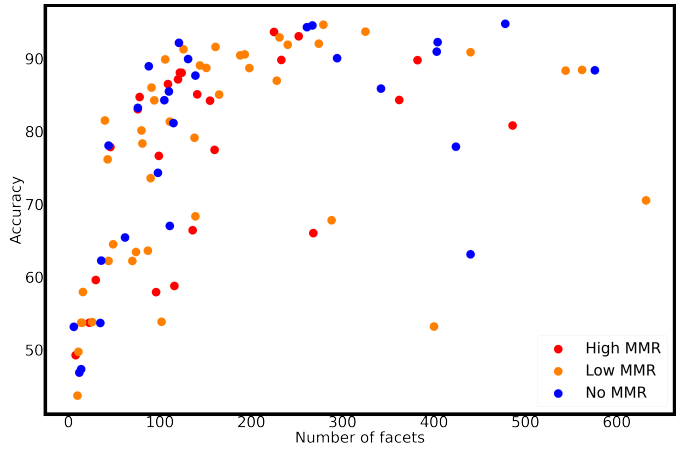


Figure 8.3: Red denotes a high value of parameter γ_{rb}, orange a low value, blue that the parameter is set to 0 value. x-axis is the number of facets, y-axis is the accuracy.

from that point up to 100 facets, where most of neural networks tend to perform above 90% accuracy. Outliers may be due to the high sensibility of optimization to initialization. Most neural networks lie on a Pareto front. This Pareto front enlights us on the relationship between accuracy and number of facets. Indeed, we clearly see that a reduction in the number of facets results in a reduction in accuracy; although accuracy does not fall before reducing the number of facets to less than 100. As said in previous chapter, a wider number of facets leads to a broader diversity of behaviours for the neural network, i.e., expressivity. It is then necessary to find a tradeoff between accuracy and robustness. Such tradeoff could be the transposition of a "No Free Lunch" Theorem for formal verification: it would be impossible to have a network that is both robust and accurate, because of the way we craft our neural networks. This argument is more detailed in the paper [Tsi+18]. Investigating this tradeoff by studying this Pareto front may enhance our understanding of the link between expressivity and accuracy.

8.4 Discussion

Our method provided enhancements for SMT solvers, but not for LP solvers. The piecewise linear structure of ReLU neural network could be much more amenable for the latter: LP solvers may have optimizations for linear problems that SMT solvers lack. However, the range of expressible properties with SMT solvers is far greater than with LP solvers, limited to conjunctions of linear inequalities. If the targetted property is complex and requires high expressivity, our method could be useful.

Our method has limited scalability on LP solvers, but problems with higher input dimensions solved with SMT solvers are benefiting from DISCO. Nevertheless, parallelizing and linear solving will still be prohibitively hard because of the increasing

number of facets for high dimensional inputs. However, there exist industrial cases with low dimensional inputs and relatively small networks that can benefit from this method. Using facets to perform parallel verification has also been used successfully in other works, for instance [Urb+19]; the global idea of considering the piece-wise-linearity of neural networks must then be investigated further.

DISCO does not perform well compared to specialized tools like ERAN or Marabou. Indeed, DISCO was not aimed to be a fully featured solver (ERAN and Marabou are developed in labs with more than 15 members). Rather, it must be seen as another heuristic that can be integrated within more mature tools. Because of engineering issues, we were not able to compare ourselves to MNIST and ACAS benchmarks: launching DISCO on ACAS did not terminate in more than twice the timeout value. Considering those results, we did not tried to test on MNIST, as both input dimension and neural network size are wider than in the ACAS setting.

Using off-the-shelf solvers rather than develop a specialized solver was a pragmatic decision. At the start of the thesis, most of state-of-the-art tools were not easily adaptable to our experiments. LP and SMT solvers have years of expertise of talented researchers and engineers that we wanted to build upon rather than start from scratch. With the rise of multiple tools and instrumenting platforms that integrates LP and SMT solvers, we think it is not a vain approach.

Since the input distribution seems not uniformly spread over facets, a possible enhancement for solving techniques would be to guide solving by focusing on “big” facets as a fail-first heuristic. The current OCaml implementation is not parallel, since the OCaml runtime does not allow straightforward multiprocess. Any multiprocess-capable language could be used to increase performance, since facets are independent. Finally, if the number of facets still remain too high, we could devise an overapproximation scheme to merge them and keep DISCO verification tractable.

Part IV

Perspectives

Perspectives

Summary of contributions

In this thesis, we went through the field of formal verification and tried to apply it on machine learning programs.

In chapter 4, we showed that straightforwardly applying formal verification techniques to machine learning was not possible. We summed up the difficulty under two different problems: the inability to formally define the input space of machine learning programs, and the lack of suitable tools and heuristics to perform efficient formal verification.

In chapter 5, we introduce CAMUS, a formalism describing how to formally express safety properties on functions taking simulated data as input. This first step helps us to consider formalizing high-dimensional inputs, in particular perceptual inputs. Using simulators as a proxy, we may be able to provide sound formalizations of specifications.

In chapter 6, we propose ISAIEH, an open source tool that leverages two standards used by the communities of formal methods and machine learning, to automatically write machine learning algorithms as logical formulae and ease the verification process. We demonstrated the joint use of ISAIEH and CAMUS on a synthetic example mimicking a self-driving car perceptive unit, as a proof of concept of our framework.

In chapter 7, we present DISCO, an algorithm that aims to facilitate the process of formal verification for neural networks. By leveraging their piece-wise linear structure, this algorithm can speed up the verification of problems using off-the-shelf solvers.

In chapter 8, we build an implementation of DISCO in ISAIEH and present our results, as well as the limitations of our approach.

Beyond this thesis

In general, the field of formal network verification is blooming and is attracting numerous scientists. It is birthing several tools and techniques. Its core strength is the interdisciplinarity between formal verification and machine learning, which yields promising advances for the field and the end users in general.

The specification problem of machine learning remains a hard one. Being able to soundly formulate specifications that the user can trust is an open research track. Indeed, perceptual inputs are difficult to qualify precisely; using a proxy that approximates the input distribution (be it a hand-crafted or learned generator) could help, providing some characterization of the quality of the input distribution exists. Any software claiming to formally verify perceptual inputs should transparently expose to all relevant parties the tradeoffs made during its conception, as well as rely on sound premises. Combining explainability approaches with generative models could provide an additional level of trust; and help people link abstractions like Gaussian noise and its actual consequence on the generated data.

The combinatorial problem is heavily researched. During the conduct of this thesis, we witnessed an explosion in the number of tools and techniques, which made keeping up with the last advances sometimes difficult. Most of prominent tools tend to focus on piece-wise linear neural network and exploit this feature to prune the search space, or propagate tight overapproximations. Adapting existing techniques of formal verification to this setting yielded fruitful results, thus advances in abstract interpretation, model-checking and guided testing should be carefully examined. Interestingly, undergoing venues to enhance solvers tend to use machine learning techniques. In a satisfying twist, neural networks could thus be used to ease the verification of neural networks. To the best of our knowledge, we do not know any work using interactive proof assistants to verify deep learning programs. We may only postulate hypothesis on the reasons of this absence. Encoding neural networks under tools like Coq or Isabelle may be too cumbersome. Coq's input language, Gallina, is expressive enough to formulate the properties we verified during this thesis. Proof assistants rely a lot on interaction with the user; if the verification process asks for input at each case split, the tool is not practically usable. In any case, a proof assistant could be useful to verify fragments of properties, or showing inner contradictions of neural networks reasoning in a more fine-grained manner.

The explosion of tools and techniques calls for a unifying way to formulate properties and choose the best (combination of heuristics) for the job to be done. We observe that most maintained tools tend to take features found in others, which leads to some sort of "selection pressure". We observe some tools that start to combine different techniques (for instance, overapproximation and linear programming). The next step would be to further intertwine reasoning techniques, so that one heuristic could inform another. To do so, we will need to unify neural network representations and provide a common reasoning base. Doing so will allow the community to further consolidate its finding while developping new approaches. One may think at the verification platforms Why3 or Frama-C, which allow such communication between methods. CAISAR is a platform currently in development that aims to answer those needs.

Most properties witnessed in the literature are either conjunctions of linear con-

straints, either adversarial robustness ones. The field would benefit from tackling a more diverse set of properties, which would enhance the tools and allow more real-world, complex problems to be tackled. For instance, trying to enforce fairness properties with formal methods could be an interesting research track.

As machine learning programs are deployed within the frames of our society, it is paramount that all members of society are engaged with it; trust is necessary if those tools are to fulfill their purpose while respecting democratic values. As computer scientists, providing ways to formally verify neural networks is a - modest - contribution to bring trust. Crafting definitions of machine learning fairness, to ensure justice against programs, is another. The potentialities of deep learning are however so big that, to paraphrase Georges Clémenceau, "Machine learning is too serious to be left alone to computer scientists". Deep learning technologies have the potential to impact our governments, and the understanding of those effects should be given to the public to conduct a democratic discussion on how we want to use technology.

Part V

Appendix

Additional material

Proof for equation (8.6)

Though $f : x \mapsto x^n$ is convex for any $n \in \mathbb{N}$, the multiplication of n variables $f : (x_1, x_2, \ldots, x_n) \mapsto \prod_k x_k$ is not convex.

For instance for $n = 2$, the surface $f : x, y \mapsto xy$ is a saddle surface.

Formulation

We aim at finding a linear (affine) lower bound and a linear upper bound to the multiplication $\prod_{k=1}^{n} x_k$ of n variables x_k in $[0.5, 2]$.

Upper bound

First, note this inequality between the product and the average:

$$\prod_{k=1}^{n} x_k \quad \leqslant \quad \left(\frac{\sum_{k=1}^{n} x_k}{n} \right)^n$$

Proof

$\log$ is concave; consequently, the average of logs is smaller than the log of the average:

$$\sum_{k=1}^{n} \frac{1}{n} \log(x_k) \quad \leqslant \quad \log \left(\frac{\sum_{k=1}^{n} x_k}{n} \right)$$

hence

$$\sum_{k=1}^{n} \log(x_k) \quad \leqslant \quad n \log \left(\frac{\sum_{k=1}^{n} x_k}{n} \right)$$

and taking the exponential we get the desired result. Note that we use the positivity of all x_k. The average $\frac{\sum_{k=1}^{n} x_k}{n}$ of numbers in $[0.5, 2]$ lies in $[0.5, 2]$ as well.

As the function $f : x \in \mathbb{R}^+ \mapsto x^n$ is convex, one has, for any $0 \leqslant a \leqslant x \leqslant b$, that $f(x)$ is below the line from $f(a)$ to $f(b)$:

$$f(x) \leqslant \frac{f(b) - f(a)}{b - a}(x - a) + f(a)$$

For our case of study, $a = 0.5$ and $b = 2$, this yields:

$$\forall x \in [0.5, 2], \quad x^n \leqslant \frac{2^n - \frac{1}{2^n}}{2 - 0.5}(x - \frac{1}{2}) + \frac{1}{2^n}$$

that is,

$$\forall x \in [0.5, 2], \quad x^n \leqslant \frac{2}{3}(2^n - 2^{-n})(x - \frac{1}{2}) + 2^{-n}$$

$$\prod_{k=1}^{n} x_k \quad \leqslant \quad \left(\frac{\sum_{k=1}^{n} x_k}{n}\right)^n \quad \leqslant \quad \frac{2}{3}(2^n - 2^{-n}) \left(\frac{\sum_{k=1}^{n} x_k}{n} - \frac{1}{2}\right) + 2^{-n}$$

Lower bound

Let us denote by f the product:

$$f : (x_1, x_2, \ldots, x_n) \mapsto \prod_k x_k$$

Then note that at the middle point $(x_1, x_2, \ldots, x_n) = (1, 1, \ldots, 1)$:

$$\forall k, \quad \frac{\partial f(x_1, x_2, \ldots, x_n)}{\partial x_k} = \prod_{j \neq k}^{n} x_j = 1$$

and that consequently around the middle point, the first order approximation of the function is:

$$f(x_1, x_2, \ldots, x_n) = f(1 + (x_1 - 1), 1 + (x_2 - 1), \ldots, 1 + (x_n - 1))$$

$$= f(1, 1, \ldots, 1) + \sum_k \frac{\partial f}{\partial x_k}(x_k - 1) + O\left((x_k - 1)^2\right)$$

$$= 1 + \sum_k (x_k - 1) + O\left((x_k - 1)^2\right)$$

$$= 1 - n + \sum_k x_k + O\left((x_k - 1)^2\right)$$

so that the linear function $(x_1, x_2, \ldots, x_n) \mapsto 1 - n + \sum_k x_k$ looks like a promising approximation of the function. Unfortunately, as said earlier, the multiplication f is not convex nor concave, so some parts of the graph of the function are above it and some other ones below. Let us just remember that the hyperplane direction $\sum_k x_k$ sounds reasonable.

The tautology:

$$\prod_k x_k - \sum_k x_k \quad \geqslant \quad \inf_{y_1,y_2,\ldots,y_n \in [0.5,2]} \left(\prod_k y_k - \sum_k y_k \right)$$

leads to:

$$\forall x_1, x_2, \ldots, x_k \in [0.5, 2],$$

$$\prod_k x_k \geqslant \sum_k x_k + \inf_{y_1,y_2,\ldots,y_n \in [0.5,2]} \left(\prod_k y_k - \sum_k y_k \right)$$

which leads us to a lower bound of the form:

$$\prod_k x_k \geqslant \sum_k x_k + C$$

for some constant C that may depend only on n and the interval chosen $[0.5, 2]$.

Let us study the function:

$$g : (x_1, x_2, \ldots, x_n) \in [0.5, 2]^n \mapsto \prod_k x_k - \sum_k x_k$$

We want to find its minimum over $[0.5, 2]^n$. For each variable x_k: **if the minimum is reached in the interior of $[0.5, 2]$** (i.e. not at $x_k = 0.5$ or 2), then necessarily at that point the derivative is 0:

$$\frac{\partial g}{\partial x_k} = \prod_{j \neq k} x_j - 1 = 0$$

i.e.

$$\prod_{j \neq k} x_j = 1$$

and consequently $\prod_j x_j = x_k$.

Otherwise, *if the minimum is reached on the boundaries of* $[0.5, 2]$*, then either* $x_k = 0.5$ *or* $x_k = 2$.

For each k we consequently have:

- either $\prod_{j \neq k} x_j = 1$
- or $x_k = 0.5$

- or $x_k = 2$

Note that if a variable x_k satisfies the first property then:

$$g(x_1, x_2, \ldots, x_n) = \prod_j x_j - \sum_j x_j = x_k - \sum_j x_j = -\sum_{j \neq k} x_j$$

which does not depend on x_k. Thus in that case one can choose to change x_k for 0.5 or 2 and this will not change the value of g. Thus one can assume that all x_k are 0.5 or 2, that is, the minimum is reached on a corner of the domain $[0.5, 2]^n$.

Let us assume that K variables x_k are 0.5 and the $n - K$ remaining ones are 2. Then:

$$g(x_1, x_2, \ldots, x_n) = 2^{n-K} 0.5^K - ((n - K)\, 2 + K\, 0.5))$$

i.e.

$$g(x_1, x_2, \ldots, x_n) = 2^{n-2K} + \frac{3}{2}K - 2n$$

What is the value of $K \in [[0, N]]$ that minimizes this?

Let is study the function $h : x \in [0, N] \mapsto 2^{n-2x} + \frac{3}{2}x$. If it reaches a minimum strictly inside $[0, N]$ then at that point its derivative is 0:

$$-2 \times 2^{n-2x} + \frac{3}{2} = 0$$

that is

$$\frac{2^{n+2}}{3} = 2^{2x}$$

$$x = \frac{1}{2}(n + 2 - \frac{\log 3}{\log 2})$$

that is

$$x \simeq \frac{n}{2} + 0.2$$

This point is a minimum indeed (and not a maximum) as the second derivative of h is positive. Therefore the K that we are searching for is the closest lower or upper integer to $\frac{n}{2} + 0.2$.

If n is even: these are $\frac{n}{2}$ and $\frac{n}{2} + 1$.

If n is odd: these are $\frac{n-1}{2}$ and $\frac{n+1}{2}$.

By computing the associated values of h, one finds that the minimum in the even case is reached for $K = \frac{n}{2}$ and is $1 + \frac{3}{4}n$, while in the odd case, the same value is obtained for both possible values of K and is $2 + \frac{3}{4}(n - 1)$.

As $g = h - 2n$ at corners, this leads to: - $\inf g = 1 - \frac{5}{4}n$ if n is even - $\inf g = \frac{5}{4} - \frac{5}{4}n$ if n is odd

$$\forall x_1, x_2, \ldots, x_k \in [0.5, 2],$$

$$\prod_k x_k \geqslant \sum_k x_k + 1 - \frac{5}{4}n + \frac{1}{4}\delta_{n \text{ is odd}}$$

with $\delta_{n \text{ is odd}} = 1$ if n is odd and 0 otherwise. The bound is tight and reached on many corners (all the ones with half lowest and half highest coordinates) as well as on the edges linking these corners if n is odd (free variable that can take any value).

Final result:

$$\begin{aligned} &\forall x_1, x_2, \ldots, x_k \in [0.5, 2], \\ \sum_k x_k + 1 - \frac{5}{4}n + \frac{1}{4}\delta_{n \text{ is odd}} \quad &\leqslant \quad \prod_k x_k \\ \prod_k x_k \quad &\leqslant \quad \frac{2}{3}(2^n - 2^{-n}) \left(\frac{\sum_{k=1}^n x_k}{n} - \frac{1}{2} \right) + 2^{-n} \end{aligned}$$

Bibliography

[21a] *Colonial Pipeline Cyberattack*. In: *Wikipedia*. May 24, 2021. URL: https://en.wikipedia.org/w/index.php?title=Colonial_Pipeline_cyberattack&oldid=1024940187 (visited on 05/25/2021) (cit. on p. 26).

[21b] *Simplex Algorithm*. In: *Wikipedia*. June 18, 2021. URL: https://en.wikipedia.org/w/index.php?title=Simplex_algorithm&oldid=1029200726 (visited on 08/10/2021) (cit. on p. 35).

[21c] *Software Crisis*. In: *Wikipedia*. July 7, 2021. URL: https://en.wikipedia.org/w/index.php?title=Software_crisis&oldid=1032500538 (visited on 08/26/2021) (cit. on p. 17).

[Aba+16a] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. "TensorFlow: A System for Large-Scale Machine Learning". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 2016, pp. 265–283. URL: https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf (visited on 03/22/2021) (cit. on pp. 47, 72).

[Aba+16b] Martin Abadi, Andy Chu, Ian Goodfellow, Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. "Deep Learning with Differential Privacy". In: *23rd ACM Conference on Computer and Communications Security (ACM CCS)*. 2016, pp. 308–318. URL: https://arxiv.org/abs/1607.00133 (visited on 05/15/2019) (cit. on p. 49).

[Ara+19] Alexandre Araujo, Rafael Pinot, Benjamin Negrevergne, Laurent Meunier, Yann Chevaleyre, Florian Yger, and Jamal Atif. *Robust Neural Networks Using Randomized Adversarial Training*. Mar. 25, 2019. arXiv: 1903.10219 [cs, stat]. URL: http://arxiv.org/abs/1903.10219 (visited on 04/25/2019) (cit. on pp. 47, 49).

[Bak+20] Stanley Bak, Hoang-Dung Tran, Kerianne Hobbs, and Taylor T. Johnson. "Improved Geometric Path Enumeration for Verifying ReLU Neural Networks". In: *Computer Aided Verification*. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 66–96. ISBN: 978-3-030-53288-8. DOI: 10.1007/978-3-030-53288-8_4 (cit. on p. 109).

[Bar+11] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. "CVC4". In: *Computer Aided Verification (CAV)*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Vol. 6806. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 171–177. ISBN: 978-3-642-22110-1. DOI: 10.1007/978-3-642-22110-1_14. URL: http://link.springer.com/10.1007/978-3-642-22110-1_14 (visited on 05/19/2021) (cit. on pp. 32, 78).

[Bau+21] Patrick Baudin, François Bobot, David Bühler, Loïc Correnson, Florent Kirchner, Nikolai Kosmatov, André Maroneze, Valentin Perrelle, Virgile Prevosto, Julien Signoles, and Nicky Williams. "The Dogged Pursuit of Bug-Free C Programs: The Frama-C Software Analysis Platform". In: *Communications of the ACM* 64.8 (Aug. 2021), pp. 56–68. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/3470569. URL: https://dl.acm.org/doi/10.1145/3470569 (visited on 11/26/2021) (cit. on p. 74).

[Ber15] Gérard Berry, director. *Structures de données et algorithmes pour la vérification formelle*. Collège de France, 2015. URL: https://www.college-de-france.fr/site/gerard-berry/course-2015-2016.htm (visited on 03/04/2021) (cit. on p. 29).

[BFS17] Clark Barrett, Pascal Fontaine, and Aaron Stump. *The SMT-LIB Standard*. July 18, 2017, p. 104. URL: http://smtlib.cs.uiowa.edu/about.shtml (cit. on p. 73).

[BKN16] Diane Bouchacourt, M. Pawan Kumar, and Sebastian Nowozin. "DISCO Nets: DISsimilarity COefficient Networks". In: Oct. 28, 2016. arXiv: 1606.02556 [cs]. URL: http://arxiv.org/abs/1606.02556 (visited on 09/13/2021) (cit. on p. 95).

[BMV20] Maximilian Baader, Matthew Mirman, and Martin Vechev. "Universal Approximation with Certified Networks". In: *International Conference on Learning Representations (ICLR)*. 2020. URL: https://openreview.net/forum?id=B1gX8kBtPr (visited on 01/15/2020) (cit. on p. 108).

[Boj+16] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. *End to End*

Learning for Self-Driving Cars. Apr. 25, 2016. arXiv: 1604.07316 [cs]. URL: http://arxiv.org/abs/1604.07316 (visited on 08/30/2019) (cit. on pp. 42, 59, 66).

[Bon+21] Gregory Bonaert, Dimitar I. Dimitrov, Maximilian Baader, and Martin Vechev. "Fast and Precise Certification of Transformers". In: *International Conference on Programming Language Design and Implementation (PLDI)*. Virtual Canada, June 2021, pp. 466–481. ISBN: 978-1-4503-8391-2. DOI: 10.1145/3453483.3454056. URL: https://dl.acm.org/doi/10.1145/3453483.3454056 (visited on 11/25/2021) (cit. on p. 109).

[Boo+19] Akhilan Boopathy, Tsui-Wei Weng, Pin-Yu Chen, Sijia Liu, and Luca Daniel. "CNN-Cert: An Efficient Framework for Certifying Robustness of Convolutional Neural Networks". In: *AAAI Conference on Artificial Intelligence*. Honolulu, HI, 2019, pp. 3240–3247. ISBN: 978-1-57735-809-1 (cit. on pp. 108, 109).

[Bun+17] Rudy Bunel, Ilker Turkaslan, Philip H. S. Torr, Pushmeet Kohli, and M. Pawan Kumar. *A Unified View of Piecewise Linear Neural Network Verification*. Nov. 1, 2017. arXiv: 1711.00455 [cs]. URL: http://arxiv.org/abs/1711.00455 (visited on 10/24/2018) (cit. on p. 104).

[Bun+20] Rudy Bunel, Jingyue Lu, Ilker Turkaslan, Philip H.S. Torr, Pushmeet Kohli, and M. Pawan Kumar. "Branch and Bound for Piecewise Linear Neural Network Verification". In: *Journal of Machine Learning Research* 21.42 (2020), pp. 1–39. URL: http://jmlr.org/papers/v21/19-468.html (cit. on pp. 105, 107).

[Cae+20] Holger Caesar, Varun Bankiti, Alex H. Lang, Sourabh Vora, Venice Erin Liong, Qiang Xu, Anush Krishnan, Yu Pan, Giancarlo Baldan, and Oscar Beijbom. *nuScenes: A Multimodal Dataset for Autonomous Driving*. May 5, 2020. arXiv: 1903.11027 [cs, stat]. URL: http://arxiv.org/abs/1903.11027 (visited on 03/11/2021) (cit. on p. 41).

[CAH19] Francesco Croce, Maksym Andriushchenko, and Matthias Hein. "Provable Robustness of ReLU Networks via Maximization of Linear Regions". In: *International Conference on Artificial Intelligence and Statistic*. 2019, pp. 2057–2066 (cit. on p. 112).

[CC77] Patrick Cousot and Radhia Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". In: *Principles of Programming Languages (POPL)*. New York, NY, USA, 1977, pp. 238–252. DOI: 10.1145/512950.512973. URL: http://doi.acm.org/10.1145/512950.512973 (visited on 10/25/2018) (cit. on p. 32).

[Cha+17] Florian Chabot, Mohamed Chaouch, Jaonary Rabarisoa, Céline Teulière, and Thierry Chateau. *Deep MANTA: A Coarse-to-fine Many-Task Network for Joint 2D and 3D Vehicle Analysis from Monocular Image*. Mar. 22, 2017. arXiv: 1703.07570 [cs]. URL: http://arxiv.org/abs/1703.07570 (visited on 08/30/2019) (cit. on p. 59).

[Cho+21] Christopher A. Choquette-Choo, Florian Tramer, Nicholas Carlini, and Nicolas Papernot. *Label-Only Membership Inference Attacks*. Jan. 21, 2021. arXiv: 2007.14321 [cs, stat]. URL: http://arxiv.org/abs/2007.14321 (visited on 03/18/2021) (cit. on p. 49).

[Cle] Clearsy. *Analyse des logiciels critiques du CBTC ligne1*. CLEARSY. URL: https://www.clearsy.com/references/saet-l1/ (visited on 08/13/2021) (cit. on p. 17).

[Coo71] Stephen A. Cook. "The Complexity of Theorem-Proving Procedures". In: *ProceedinSymposium on Theory of Computing (STOC)*. Shaker Heights, Ohio, United States, 1971, pp. 151–158. DOI: 10.1145/800157.805047. URL: http://portal.acm.org/citation.cfm?doid=800157.805047 (visited on 08/10/2021) (cit. on p. 30).

[Cop] Copeland. *Artificial Intelligence | Definition, Examples, and Applications*. Encyclopedia Britannica. URL: https://www.britannica.com/technology/artificial-intelligence (visited on 03/08/2021) (cit. on p. 38).

[Cre+18] Antonia Creswell, Tom White, Vincent Dumoulin, Kai Arulkumaran, Biswa Sengupta, and Anil A. Bharath. "Generative Adversarial Networks: An Overview". In: *IEEE Signal Processing Magazine* 35.1 (Jan. 2018), pp. 53–65. ISSN: 1053-5888. DOI: 10.1109/MSP.2017.2765202. arXiv: 1710.07035. URL: http://arxiv.org/abs/1710.07035 (visited on 03/22/2021) (cit. on p. 42).

[CS13] Cristian Cadar and Koushik Sen. "Symbolic Execution for Software Testing: Three Decades Later". In: *Communications of the ACM* (2013), pp. 82–90 (cit. on pp. 34, 35).

[CW16] Nicholas Carlini and David Wagner. *Towards Evaluating the Robustness of Neural Networks*. Aug. 16, 2016. arXiv: 1608.04644 [cs]. URL: http://arxiv.org/abs/1608.04644 (visited on 11/07/2018) (cit. on p. 47).

[CW18] Nicholas Carlini and David Wagner. *Audio Adversarial Examples: Targeted Attacks on Speech-to-Text*. Jan. 5, 2018. arXiv: 1801.01944 [cs]. URL: http://arxiv.org/abs/1801.01944 (visited on 11/21/2018) (cit. on p. 18).

[Den+09] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. "ImageNet: A Large-Scale Hierarchical Image Database". In: *IEEE Computer Vision and Pattern Recognition (CVPR)* (2009), p. 8 (cit. on p. 41).

[DLL62] Martin Davis, George Logemann, and Donald Loveland. "A Machine Program for Theorem-Proving". In: *Communications of the ACM* 5.7 (July 1, 1962), pp. 394–397. ISSN: 0001-0782. DOI: 10.1145/368273.368557. URL: https://doi.org/10.1145/368273.368557 (visited on 03/05/2021) (cit. on p. 30).

[dMB08] Leonardo de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2008, pp. 337–340. ISBN: 978-3-540-78800-3. DOI: 10.1007/978-3-540-78800-3_24 (cit. on pp. 32, 78, 111).

[Dos+17] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. "CARLA: An Open Urban Driving Simulator". In: *Conference on Robot Learning*. Oct. 18, 2017. URL: http://proceedings.mlr.press/v78/dosovitskiy17a.html (visited on 06/03/2021) (cit. on p. 60).

[Dou21] Will Douglas Heaven. *Hundreds of AI Tools Have Been Built to Catch Covid. None of Them Helped.* MIT Technology Review. July 30, 2021. URL: https://www.technologyreview.com/2021/07/30/1030329/machine-learning-ai-failed-covid-hospital-diagnosis-pandemic/ (visited on 08/18/2021) (cit. on p. 58).

[Dre+19] Tommaso Dreossi, Daniel J. Fremont, Shromona Ghosh, Edward Kim, Hadi Ravanbakhsh, Marcell Vazquez-Chanlatte, and Sanjit A. Seshia. *VERIFAI: A Toolkit for the Design and Analysis of Artificial Intelligence-Based Systems*. Feb. 12, 2019. arXiv: 1902.04245 [cs]. URL: http://arxiv.org/abs/1902.04245 (visited on 10/02/2019) (cit. on p. 62).

[Dut14] Bruno Dutertre. "Yices 2.2". In: *Computer Aided Verification (CAV)*. Ed. by Armin Biere and Roderick Bloem. Vol. 8559. Lecture Notes in Computer Science. Springer, July 2014, pp. 737–744 (cit. on p. 78).

[Ehl17] Ruediger Ehlers. *Formal Verification of Piece-Wise Linear Feed-Forward Neural Networks*. May 3, 2017. arXiv: 1705.01320 [cs]. URL: http://arxiv.org/abs/1705.01320 (visited on 06/20/2019) (cit. on pp. 104, 107).

[Eyk+18] Kevin Eykholt, Ivan Evtimov, Earlence Fernandes, Bo Li, Amir Rahmati, Chaowei Xiao, Atul Prakash, Tadayoshi Kohno, and Dawn Song. "Robust Physical-World Attacks on Deep Learning Visual Classification". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Salt Lake City, UT, USA: IEEE, June 2018, pp. 1625–1634. ISBN: 978-1-5386-6420-9. DOI: 10.1109/CVPR.2018.00175. URL: https://ieeexplore.ieee.org/document/8578273/ (visited on 03/18/2021) (cit. on p. 47).

[FG19] Nathanaël Fijalkow and Mohit Kumar Gupta. *Verification of Neural Networks: Specifying Global Robustness Using Generative Models*. Oct. 11, 2019. arXiv: 1910.05018 [cs, stat]. URL: http://arxiv.org/abs/1910.05018 (visited on 06/09/2021) (cit. on pp. 68, 79).

[GGP09] Khalil Ghorbal, Eric Goubault, and Sylvie Putot. "The Zonotope Abstract Domain Taylor1+". In: *Computer Aided Verification*. Ed. by Ahmed Bouajjani and Oded Maler. Vol. 5643. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 627–633. ISBN: 978-3-642-02658-4. DOI: 10.1007/978-3-642-02658-4_47. URL: http://link.springer.com/10.1007/978-3-642-02658-4_47 (visited on 04/02/2020) (cit. on p. 32).

[GSS14] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. *Explaining and Harnessing Adversarial Examples*. Dec. 19, 2014. arXiv: 1412.6572 [cs, stat]. URL: http://arxiv.org/abs/1412.6572 (visited on 10/24/2018) (cit. on pp. 47–49).

[Gur21] LLC Gurobi Optimization. "Gurobi Optimizer Reference Manual". In: (2021). URL: http://www.gurobi.com (cit. on pp. 35, 111).

[Haw19] Andrew J. Hawkins. *Tesla Didn't Fix an Autopilot Problem for Three Years, and Now Another Person Is Dead*. The Verge. May 17, 2019. URL: https://www.theverge.com/2019/5/17/18629214/tesla-autopilot-crash-death-josh-brown-jeremy-banner (visited on 03/29/2021) (cit. on pp. 18, 61).

[He+15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. *Deep Residual Learning for Image Recognition*. Dec. 10, 2015. arXiv: 1512.03385 [cs]. URL: http://arxiv.org/abs/1512.03385 (visited on 04/02/2019) (cit. on pp. 42, 52, 59).

[HR19a] Boris Hanin and David Rolnick. *Complexity of Linear Regions in Deep Networks*. Jan. 25, 2019. arXiv: 1901.09021 [cs, math, stat]. URL: http://arxiv.org/abs/1901.09021 (visited on 03/27/2019) (cit. on p. 85).

[HR19b] Boris Hanin and David Rolnick. "Deep ReLU Networks Have Surprisingly Few Activation Patterns". In: *Conference on Neural Information Processing Systems (NeurIPS)*. 2019, pp. 361–370 (cit. on pp. 91, 95, 96).

[Ily+19] Andrew Ilyas, Shibani Santurkar, Dimitris Tsipras, Logan Engstrom, Brandon Tran, and Aleksander Madry. *Adversarial Examples Are Not Bugs, They Are Features*. May 6, 2019. arXiv: 1905.02175 [cs, stat]. URL: http://arxiv.org/abs/1905.02175 (visited on 05/17/2019) (cit. on p. 49).

[Iva20] Sergei Ivanov. *NeurIPS 2020. Comprehensive Analysis of Authors, Organizations, and Countries*. Medium. Oct. 15, 2020. URL: https://medium.com/criteo-engineering/neurips-2020-comprehensive-analysis-of-authors-organizations-and-countries-a1b55a08132e (visited on 03/22/2021) (cit. on p. 41).

[JJR21] JJRicks Studios, director. *Waymo Self Driving Taxi Fumbles In Construction Zone, Blocks Traffic | JJRicks Rides With Waymo #54*. May 13, 2021. URL: https://www.youtube.com/watch?v=zdKCQKBvH-A (visited on 07/07/2021) (cit. on p. 40).

[Kat+17] Guy Katz, Clark Barrett, David Dill, Kyle Julian, and Mykel Kochenderfer. "Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks". In: *International Conference on Computer Aided Verification*. Springer, Feb. 3, 2017, pp. 91–117. arXiv: 1702.01135 (cit. on pp. 50, 62, 84, 104).

[Kat+19] Guy Katz, Derek A. Huang, Duligur Ibeling, Kyle Julian, Christopher Lazarus, Rachel Lim, Parth Shah, Shantanu Thakoor, Haoze Wu, Aleksandar Zeljić, David L. Dill, Mykel J. Kochenderfer, and Clark Barrett. "The Marabou Framework for Verification and Analysis of Deep Neural Networks". In: *Computer Aided Verification*. Ed. by Isil Dillig and Serdar Tasiran. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2019, pp. 443–452. ISBN: 978-3-030-25540-4. DOI: 10.1007/978-3-030-25540-4_26 (cit. on pp. 104, 107).

[Kat+21] Sydney M. Katz, Anthony L. Corso, Christopher A. Strong, and Mykel J. Kochenderfer. *Verification of Image-based Neural Network Controllers Using Generative Models*. May 14, 2021. arXiv: 2105.07091 [cs]. URL: http://arxiv.org/abs/2105.07091 (visited on 06/08/2021) (cit. on pp. 68, 79).

[KLA18] Tero Karras, Samuli Laine, and Timo Aila. *A Style-Based Generator Architecture for Generative Adversarial Networks*. Dec. 12, 2018. arXiv: 1812.04948 [cs, stat]. URL: http://arxiv.org/abs/1812.04948 (visited on 08/22/2019) (cit. on p. 59).

[KSH17] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Communications of the ACM* 60.6 (May 24, 2017), pp. 84–90. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/3065386. URL: https://dl.acm.org/doi/10.1145/3065386 (visited on 03/09/2021) (cit. on p. 41).

[Le +89] Y. Le Cun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. "Handwritten Digit Recognition with a Back-Propagation Network". In: *Advances in Neural Information Processing Systems (NIPS)*. Cambridge, MA, USA: MIT Press, Jan. 1, 1989, pp. 396–404 (cit. on p. 41).

[Le 86] Yann Le Cun. "Learning Process in an Asymmetric Threshold Network". In: *Disordered Systems and Biological Organization*. Ed. by E. Bienenstock, F. Fogelman Soulié, and G. Weisbuch. NATO ASI Series. Berlin, Heidelberg: Springer, 1986, pp. 233–240. ISBN: 978-3-642-82657-3. DOI: 10.1007/978-3-642-82657-3_24 (cit. on p. 40).

[Ler19] Xavier Leroy, director. *Programmer = démontrer ? La correspondance de Curry-Howard aujourd'hui*. Collège de France, 2019. URL: https://www.college-de-france.fr/site/xavier-leroy/course-2018-2019.htm (visited on 03/03/2021) (cit. on p. 35).

[Les10] Lawrence Lessig. *Code: Version 2.0*. SoHo Books, 2010. ISBN: 978-1-4414-3764-8 (cit. on p. 15).

[Li 12] Li Deng. "The MNIST Database of Handwritten Digit Images for Machine Learning Research [Best of the Web]". In: *IEEE Signal Processing Magazine* 29.6 (Nov. 2012), pp. 141–142. ISSN: 1053-5888. DOI: 10.1109/MSP.2012.2211477. URL: http://ieeexplore.ieee.org/document/6296535/ (visited on 03/11/2021) (cit. on p. 41).

[Lio96] Jacques-Louis Lions. *ARIANE 5 Failure - Full Report*. June 19, 1996. URL: http://sunnyday.mit.edu/accidents/Ariane5accidentreport.html (visited on 03/04/2021) (cit. on p. 28).

[Lip16] Zachary C. Lipton. *The Mythos of Model Interpretability*. June 10, 2016. arXiv: 1606.03490 [cs, stat]. URL: http://arxiv.org/abs/1606.03490 (visited on 11/07/2018) (cit. on p. 69).

[LSP07] Thierry Lecomte, Thierry Servat, and Guilhem Pouzancre. "Formal Methods in Safety-Critical Railway Systems". In: Brazilian Symposium on Formal Methods. 2007, p. 9 (cit. on p. 17).

[Mad+17] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. *Towards Deep Learning Models Resistant to Adversarial Attacks*. June 19, 2017. arXiv: 1706.06083 [cs, stat]. URL: http://arxiv.org/abs/1706.06083 (visited on 10/24/2018) (cit. on p. 47).

[Mak] A. Makhorin. "GLPK (GNU Linear Programming Kit)". In: () (cit. on p. 35).

[Mar+96] Jo Marques, Joao P. Marques Silva, Joao P. Marques Silva, Karem A. Sakallah, and Karem A. Sakallah. "GRASP—A New Search Algorithm for Satisfiability". In: *In Proceedings of the International Conference on Computer-Aided Design*. 1996, pp. 220–227 (cit. on p. 30).

[Mat+16] Surya Mattu, Julia Angwin, Jeff Larson, and Lauren Kirchner. *Machine Bias*. ProPublica. 2016. URL: https://www.propublica.org/article/machine-bias-risk-assessments-in-criminal-sentencing?token=nEh5WNViIayEtqf96qVA8Dp-s2YDMY-f (visited on 03/12/2021) (cit. on p. 44).

[MBC17] Bruno Marre, Francois Bobot, and Zakaria Chihani. "Real Behavior of Floating Point Numbers". In: *15th International Workshop on Satisfiability Modulo Theories*. Heidelberg, Germany, 2017. URL: https://smt-workshop.cs.uiowa.edu/2017/papers/SMT2017_paper_21.pdf (cit. on p. 78).

[Meh+19] Ninareh Mehrabi, Fred Morstatter, Nripsuta Saxena, Kristina Lerman, and Aram Galstyan. *A Survey on Bias and Fairness in Machine Learning*. Sept. 17, 2019. arXiv: 1908.09635 [cs]. URL: http://arxiv.org/abs/1908.09635 (visited on 03/12/2021) (cit. on p. 53).

[MGV18] Matthew Mirman, Timon Gehr, and Martin Vechev. "Differentiable Abstract Interpretation for Provably Robust Neural Networks". In: *International Conference on Machine Learning (ICML)*. 2018 (cit. on p. 107).

[Min17] Antoine Miné. "Tutorial on Static Inference of Numeric Invariants by Abstract Interpretation". In: *Foundations and Trends® in Programming Languages* 4.3-4 (2017), pp. 120–372. ISSN: 2325-1107, 2325-1131. DOI: 10.1561/2500000034. URL: http://www.nowpublishers.com/article/Details/PGL-034 (visited on 03/04/2021) (cit. on p. 35).

[Mit21] Melanie Mitchell. *Why AI Is Harder Than We Think*. Apr. 28, 2021. arXiv: 2104.12871 [cs]. URL: http://arxiv.org/abs/2104.12871 (visited on 05/25/2021) (cit. on p. 40).

[MJ16] Guido Manfredi and Yannick Jestin. "An Introduction to ACAS Xu and the Challenges Ahead". In: *IEEE/AIAA Digital Avionics Systems Conference (DASC)*. Sacramento, CA, USA, Sept. 2016. ISBN: 978-1-5090-2523-7. DOI: 10.1109/DASC.2016.7778055. URL: http://ieeexplore.ieee.org/document/7778055/ (visited on 03/29/2021) (cit. on p. 49).

[Mos+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. "Chaff: Engineering an Efficient SAT Solver". In: *Design Automation Conference (DAC)*. New York, NY, USA: Association for Computing Machinery, June 22, 2001, pp. 530–535. ISBN: 978-1-58113-297-7. DOI: 10.1145/378239.379017. URL: https://doi.org/10.1145/378239.379017 (visited on 03/05/2021) (cit. on p. 30).

[MP43] Warren S. McCulloch and Walter Pitts. "A Logical Calculus of the Ideas Immanent in Nervous Activity". In: *The bulletin of mathematical biophysics* 5.4 (4 Dec. 1, 1943), pp. 115–133. ISSN: 1522-9602. DOI: 10.1007/BF02478259. URL: https://link.springer.com/article/10.1007/BF02478259 (visited on 03/10/2021) (cit. on p. 39).

[MP72] Marvin Minsky and Seymour A. Papert. *Perceptrons: An Introduction to Computational Geometry*. 2. print. with corr. Cambridge/Mass.: The MIT Press, 1972. 258 pp. ISBN: 978-0-262-63022-1 (cit. on p. 39).

[Owe21] Therese Poletti Owens Jeremy C. *Opinion: It's Time for Elon Musk to Start Telling the Truth about Autonomous Driving*. MarketWatch. May 29, 2021. URL: https://www.marketwatch.com/story/its-time-for-elon-musk-to-start-telling-the-truth-about-autonomous-driving-11621958761 (visited on 07/07/2021) (cit. on p. 40).

[Pas+19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2019. URL: https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html (cit. on pp. 47, 72, 111).

[PMG16] Nicolas Papernot, Patrick McDaniel, and Ian Goodfellow. *Transferability in Machine Learning: From Phenomena to Black-Box Attacks Using Adversarial Samples*. May 23, 2016. arXiv: 1605.07277 [cs]. URL: http://arxiv.org/abs/1605.07277 (visited on 12/18/2018) (cit. on p. 47).

[Pom89] Dean A Pomerleau. "ALVINN: An Autonomous Land Vehicle in a Neural Network". In: *Proceedings of Advances in Neural Information Processing Systems (NIPS)* (Dec. 1989), pp. 305–313 (cit. on p. 60).

[PT10] Luca Pulina and Armando Tacchella. "An Abstraction-Refinement Approach to Verification of Artificial Neural Networks". In: *Computer Aided Verification (CAV)*. 2010. DOI: 10.1007/978-3-642-14295-6_24 (cit. on p. 107).

[Rag+17] Maithra Raghu, Ben Poole, Jon Kleinberg, Surya Ganguli, and Jascha Sohl-Dickstein. "On the Expressive Power of Deep Neural Networks". In: *International Conference on Machine Learning*. International Conference on Machine Learning. PMLR, July 17, 2017, pp. 2847–2854. URL: http://proceedings.mlr.press/v70/raghu17a.html (visited on 07/02/2021) (cit. on pp. 85, 90, 97).

[RHW86] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning Representations by Back-Propagating Errors". In: *Nature* 323.6088 (6088 Oct. 1986), pp. 533–536. ISSN: 1476-4687. DOI: 10.1038/323533a0. URL: https://www.nature.com/articles/323533a0 (visited on 03/09/2021) (cit. on p. 40).

[Ric+16] Stephan R. Richter, Vibhav Vineet, Stefan Roth, and Vladlen Koltun. *Playing for Data: Ground Truth from Computer Games*. Aug. 7, 2016. arXiv: 1608.02192 [cs]. URL: http://arxiv.org/abs/1608.02192 (visited on 06/03/2021) (cit. on p. 60).

[Ric53] Henry Gordon Rice. "Classes of Recursively Enumerable Sets and Their Decision Problems". In: *Transaction of American Mathematical Society* 74 (1953), pp. 358–366. DOI: 10.1090/S0002-9947-1953-0053041-6 (cit. on p. 28).

[RND10] Stuart J. Russell, Peter Norvig, and Ernest Davis. *Artificial Intelligence: A Modern Approach*. 3rd ed. Prentice Hall Series in Artificial Intelligence. Upper Saddle River: Prentice Hall, 2010. 1132 pp. ISBN: 978-0-13-604259-4 (cit. on pp. 38, 69).

[Ros58] Frank Rosenblatt. "The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain." In: *Psychological Review* 65.6 (1958), pp. 386–408. ISSN: 1939-1471, 0033-295X. DOI: 10.1037/h0042519. URL: http://doi.apa.org/getdoi.cfm?doi=10.1037/h0042519 (visited on 03/09/2021) (cit. on p. 39).

[RPB18] Mirco Ravanelli, Titouan Parcollet, and Yoshua Bengio. *The PyTorch-Kaldi Speech Recognition Toolkit*. Nov. 18, 2018. arXiv: 1811.07453 [cs, eess]. URL: http://arxiv.org/abs/1811.07453 (visited on 09/07/2019) (cit. on p. 59).

[Sam+21] Wojciech Samek, Gregoire Montavon, Sebastian Lapuschkin, Christopher J. Anders, and Klaus-Robert Muller. "Explaining Deep Neural Networks and Beyond: A Review of Methods and Applications". In: *Proceedings of the IEEE* 109.3 (Mar. 2021), pp. 247–278. ISSN: 0018-9219, 1558-2256. DOI: 10.1109/JPROC.2021.3060483. URL: https://ieeexplore.ieee.org/document/9369420/ (visited on 08/10/2021) (cit. on p. 53).

[SCA15] SCADAStrangeLove. *32C3 Slides*. Dec. 27, 2015. URL: http://www.scada.sl/2015/12/32c3-slides.html (visited on 05/25/2021) (cit. on p. 26).

[Sch] Jürgen Schmidhuber. *Who Invented Backpropagation?* URL: https://people.idsia.ch/~juergen/who-invented-backpropagation.html (visited on 03/09/2021) (cit. on p. 41).

[SG19] Gagandeep Singh and Timon Gehr. "Boosting Robustness Certification of Neural Networks". In: *International Conference on Learning Representations (ICLR)*. 2019, p. 12 (cit. on pp. 107, 109).

[She+19] Li Shen, Laurie R. Margolies, Joseph H. Rothstein, Eugene Fluder, Russell McBride, and Weiva Sieh. "Deep Learning to Improve Breast Cancer Detection on Screening Mammography". In: *Scientific Reports* 9.1 (1 Aug. 29, 2019), p. 12495. ISSN: 2045-2322. DOI: 10.1038/s41598-019-48995-4. URL: https://www.nature.com/articles/s41598-019-48995-4 (visited on 06/22/2021) (cit. on p. 18).

[Sho+17] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. "Membership Inference Attacks Against Machine Learning Models". In: *2017 IEEE Symposium on Security and Privacy (SP)*. 2017 IEEE Symposium on Security and Privacy (SP). San Jose, CA, USA: IEEE, May 2017, pp. 3–18. ISBN: 978-1-5090-5533-3. DOI: 10.1109/SP.2017.41. URL: http://ieeexplore.ieee.org/document/7958568/ (visited on 11/16/2018) (cit. on p. 49).

[Sig18] Julien Signoles. "From Static Analysis to Runtime Verification with Frama-C and E-ACSL". July 9, 2018. URL: http://julien.signoles.free.fr/publis/hdr.pdf (visited on 03/04/2021) (cit. on p. 28).

[Sin+19] Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin Vechev. "An Abstract Domain for Certifying Neural Networks". In: *Proceedings of the ACM on Programming Languages (POPL)*. Vol. 3. 2019, pp. 1–30 (cit. on p. 107).

[SR19] Thiago Serra and Srikumar Ramalingam. *Empirical Bounds on Linear Regions of Deep Rectifier Networks*. Dec. 14, 2019. arXiv: 1810.03370 [cs, math, stat]. URL: http://arxiv.org/abs/1810.03370 (visited on 04/03/2020) (cit. on p. 91).

[Sta21] Nick Statt. *Hackers Tampered with a Water Treatment Facility in Florida by Changing Chemical Levels*. The Verge. Feb. 8, 2021. URL: https://www.theverge.com/2021/2/8/22273170/hackers-water-treatment-facility-florida-hacked-chemical-levels-changed (visited on 05/25/2021) (cit. on p. 26).

[STR18] Thiago Serra, Christian Tjandraatmadja, and Srikumar Ramalingam. *Bounding and Counting Linear Regions of Deep Neural Networks*. Sept. 15, 2018. arXiv: 1711.02114 [cs, math, stat]. URL: http://arxiv.org/abs/1711.02114 (visited on 03/26/2020) (cit. on pp. 85, 91, 108).

[Syn20] Synced. *Exploring Gender Imbalance in AI: Numbers, Trends, and Discussions | Synced*. Mar. 13, 2020. URL: https://syncedreview.com/2020/03/13/exploring-gender-imbalance-in-ai-numbers-trends-and-discussions/ (visited on 03/22/2021) (cit. on p. 41).

[Sze+13] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. *Intriguing Properties of Neural Networks*. Dec. 20, 2013. arXiv: 1312.6199 [cs]. URL: http://arxiv.org/abs/1312.6199 (visited on 10/24/2018) (cit. on p. 47).

[Tab+20] Mark Tab, Kent Sharkley, David Coulter, and C.J. Gronlund. *Azure AI Guide for Predictive Maintenance Solutions - Team Data Science Process*. Oct. 1, 2020. URL: https://docs.microsoft.com/en-us/azure/machine-learning/team-data-science-process/predictive-maintenance-playbook (visited on 06/22/2021) (cit. on p. 18).

[Tan+18] Chuanqi Tan, Fuchun Sun, Tao Kong, Wenchang Zhang, Chao Yang, and Chunfang Liu. "A Survey on Deep Transfer Learning". In: *Artificial Neural Networks and Machine Learning – ICANN 2018*. Ed. by Věra Kůrková, Yannis Manolopoulos, Barbara Hammer, Lazaros Iliadis, and Ilias Maglogiannis. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, pp. 270–279. ISBN: 978-3-030-01424-7. DOI: 10.1007/978-3-030-01424-7_27 (cit. on pp. 44, 53).

[Tol+21] Felipe Toledo, David Shriver, Sebastian Elbaum, and Matthew B Dwyer. "Distribution Models for Falsification and Verification of DNNs". In: *The 36th IEEE/ACM International Conference on Automated Software Engineering*. July 14, 2021, p. 13 (cit. on p. 79).

[Tra+16] Florian Tramèr, Fan Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. "Stealing Machine Learning Models via Prediction APIs". In: 25th USENIX Security Symposium (USENIX Security 16). 2016, pp. 601–618. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/tramer (visited on 11/16/2018) (cit. on p. 49).

[Tra+20] Hoang-Dung Tran, Xiaodong Yang, Diego Manzanas Lopez, Patrick Musau, Luan Viet Nguyen, Weiming Xiang, Stanley Bak, and Taylor T. Johnson. "NNV: The Neural Network Verification Tool for Deep Neural Networks and Learning-Enabled Cyber-Physical Systems". In: *Computer Aided Verification*. Ed. by Shuvendu K. Lahiri and Chao Wang. Vol. 12224. Lecture

Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 3–17. ISBN: 978-3-030-53287-1. DOI: 10.1007/978-3-030-53288-8_1. URL: http://link.springer.com/10.1007/978-3-030-53288-8_1 (visited on 07/13/2021) (cit. on p. 109).

[Tri19] Gilles Triolier. *Frappé par une cyberattaque massive, le CHU de Rouen forcé de tourner sans ordinateurs*. Le Monde.fr. Nov. 18, 2019. URL: https://www.lemonde.fr/pixels/article/2019/11/18/frappe-par-une-cyberattaque-massive-le-chu-de-rouen-force-de-tourner-sans-ordinateurs_6019650_4408996.html (visited on 05/25/2021) (cit. on p. 26).

[Tru+19] Stacey Truex, Ling Liu, Mehmet Emre Gursoy, Lei Yu, and Wenqi Wei. *Towards Demystifying Membership Inference Attacks*. Feb. 1, 2019. arXiv: 1807.09173 [cs]. URL: http://arxiv.org/abs/1807.09173 (visited on 03/18/2021) (cit. on p. 49).

[Tsi+18] Dimitris Tsipras, Shibani Santurkar, Logan Engstrom, Alexander Turner, and Aleksander Madry. *Robustness May Be at Odds with Accuracy*. May 30, 2018. arXiv: 1805.12152 [cs, stat]. URL: http://arxiv.org/abs/1805.12152 (visited on 10/24/2018) (cit. on p. 115).

[Tub21] Paola Tubaro. "Disembedded or Deeply Embedded? A Multi-Level Network Analysis of Online Labour Platforms". In: *Sociology* (Jan. 31, 2021), p. 003803852098608. ISSN: 0038-0385, 1469-8684. DOI: 10.1177/0038038520986082. URL: http://journals.sagepub.com/doi/10.1177/0038038520986082 (visited on 08/10/2021) (cit. on p. 43).

[Tur50] A. M. Turing. "I.—COMPUTING MACHINERY AND INTELLIGENCE". In: *Mind* LIX.236 (Oct. 1, 1950), pp. 433–460. ISSN: 1460-2113, 0026-4423. DOI: 10.1093/mind/LIX.236.433. URL: https://academic.oup.com/mind/article/LIX/236/433/986238 (visited on 03/09/2021) (cit. on p. 39).

[TXT19] Vincent Tjeng, Kai Xiao, and Russ Tedrake. "Evaluating Robustness of Neural Networks with Mixed Integer Programming". In: International Conference on Learning Representations (ICLR). 2019. URL: https://openreview.net/pdf?id=HyGIdiRqtm (visited on 06/19/2019) (cit. on pp. 105, 107).

[Urb+19] Caterina Urban, Maria Christakis, Valentin Wüstholz, and Fuyuan Zhang. *Perfectly Parallel Fairness Certification of Neural Networks*. Dec. 5, 2019. arXiv: 1912.02499 [cs]. URL: http://arxiv.org/abs/1912.02499 (visited on 03/26/2020) (cit. on pp. 85, 116).

[Vas+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. *Attention Is All You Need*. Dec. 5, 2017. arXiv: 1706.03762 [cs]. URL: http://arxiv.org/abs/1706.03762 (visited on 04/02/2020) (cit. on p. 42).

[Vin+19] Oriol Vinyals, Igor Babuschkin, Wojciech M. Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H. Choi, Richard Powell, Timo Ewalds, Petko Georgiev, Junhyuk Oh, Dan Horgan, Manuel Kroiss, Ivo Danihelka, Aja Huang, Laurent Sifre, Trevor Cai, John P. Agapiou, Max Jaderberg, Alexander S. Vezhnevets, Rémi Leblond, Tobias Pohlen, Valentin Dalibard, David Budden, Yury Sulsky, James Molloy, Tom L. Paine, Caglar Gulcehre, Ziyu Wang, Tobias Pfaff, Yuhuai Wu, Roman Ring, Dani Yogatama, Dario Wünsch, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy Lillicrap, Koray Kavukcuoglu, Demis Hassabis, Chris Apps, and David Silver. "Grandmaster Level in StarCraft II Using Multi-Agent Reinforcement Learning". In: *Nature* 575.7782 (7782 Nov. 2019), pp. 350–354. ISSN: 1476-4687. DOI: 10.1038/s41586-019-1724-z. URL: https://www.nature.com/articles/s41586-019-1724-z (visited on 08/24/2021) (cit. on p. 60).

[Vin21] James Vincent. *Canon Put AI Cameras in Its Chinese Offices That Only Let Smiling Workers Inside*. The Verge. June 17, 2021. URL: https://www.theverge.com/2021/6/17/22538160/ai-camera-smile-recognition-office-workers-china-canon (visited on 06/21/2021) (cit. on pp. 18, 42).

[Wan+18a] Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. "Efficient Formal Safety Analysis of Neural Networks". In: *Advances in Neural Information Processing Systems 31*. Ed. by S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett. Curran Associates, Inc., 2018, pp. 6369–6379. URL: http://papers.nips.cc/paper/7873-efficient-formal-safety-analysis-of-neural-networks.pdf (visited on 12/17/2018) (cit. on p. 108).

[Wan+18b] Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. *Formal Security Analysis of Neural Networks Using Symbolic Intervals*. Apr. 28, 2018. arXiv: 1804.10829 [cs]. URL: http://arxiv.org/abs/1804.10829 (visited on 06/21/2019) (cit. on pp. 108, 109).

[Wer90] Paul John Werbos. "Backpropagation through Time: What It Does and How to Do It". In: *Proceedings of the IEEE* 78.10 (Oct. 1990), pp. 1550–1560. ISSN: 00189219. DOI: 10.1109/5.58337. URL: http://ieeexplore.ieee.org/document/58337/ (visited on 03/09/2021) (cit. on p. 41).

[WH60] Bernard Widrow and Ted Hoff. *An Adaptative "Adaline" Neuron Using Chemical "Memistor"*. Technical report 1553-2. Oct. 17, 1960. URL: https://isl.stanford.edu/~widrow/papers/t1960anadaptive.pdf (visited on 03/10/2021) (cit. on p. 40).

[Win92] Patrick Henry Winston. *Artificial Intelligence*. 3rd ed. Reading, Mass: Addison-Wesley Pub. Co, 1992. 737 pp. ISBN: 978-0-201-53377-4 (cit. on p. 38).

[WK17] Eric Wong and J. Zico Kolter. "Provable Defenses against Adversarial Examples via the Convex Outer Adversarial Polytope". In: *Proceedings of the 35th International Conference on Machine Learning* (Nov. 2, 2017). URL: https://arxiv.org/abs/1711.00851v3 (visited on 04/12/2019) (cit. on pp. 107, 109).

[Wor15] World Economic Forum, director. *Value Alignment | Stuart Russell*. May 24, 2015. URL: https://www.youtube.com/watch?v=WvmeTaFc_Qw (visited on 08/24/2021) (cit. on p. 69).

[Ye+21] M. Ye, J. Shen, G. Lin, T. Xiang, L. Shao, and S. C. H. Hoi. "Deep Learning for Person Re-identification: A Survey and Outlook". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2021), pp. 1–1. ISSN: 1939-3539. DOI: 10.1109/TPAMI.2021.3054775 (cit. on p. 42).

List of Figures

List of Tables

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

université PARIS-SACLAY

Titre: Vérification et validation de techniques d'apprentissage machine

Mots clés: Réseaux de neurones, méthodes formelles

Résumé: L'apprentissage machine, en particulier au moyen des réseaux de neurones artificiels, connaît depuis une dizaine d'année une expansion impressionnante. détecteurs de collision d'aéronefs, aide au diagnostic pour différents cancers, aides aux décisions de justice, véhicules autonomes et capteurs d'anomalies d'ancrage sur des plateformes offshores sont autant d'applications faisant intervenir les technologies d'apprentissage profond au sein de systèmes critique; ouvrant des perspectives inexplorées pour les sociétés humaines. Bien que bénéfique en apparence, cette révolution a de quoi inquiéter à mesure qu'elle se concrétise: la fragilité de ces techniques d'apprentissage est désormais un fait scientifique établi. La taxonomie des vulnérabilités, qu'elles soient accidentelles ou malicieuses, ainsi que leur caractère imprévisible remet en question la possibilité d'intégrer des réseaux de neurones dans des domaines critiques qui pourraient pourtant en bénéficier. A l'heure actuelle, peu de méthodes permettent de démontrer formellement la fiabilité d'un réseau de neurones. Par contraste, le domaine du logiciel critique, quant à lui, jouit d'une multitude de méthodes et techniques: model checking, simulation, interprétation abstraite, tests dirigés, etc.

L'objectif de cette thèse est de réconcilier l'abondance des techniques de vérification de programmes classiques et l'absence de garanties sur les réseaux de neurones, ce pour permettre aux logiciels critiques de conserver le haut niveau de confiance qu'ils ont atteint quand ils seront inévitablement modifiés avec des mécanismes d'apprentissage machine.

Title: Verification and validation of machine learning techniques

Keywords: Neural Networks, Formal Verification

Abstract: Machine Learning techniques, Neural Networks in particular, are going through an impressive expansion, permeating various domains, becoming the next frontier for human societies. Aircraft collision avoidance, cancer detection, justice advisors, autonomous vehicles, or mooring line failure detection are but a few examples of Neural Networks applications. This effervescence, however, may hold more than benefits, as it slowly but surely reaches critical systems. Indeed, the remarkable efficiency of neural nets comes at a price, more and more underlined by the scientific consensus: weakness to environmental or adversarial perturbations, unpredictability... which prevents their full-scale integration into critical systems. While the domain of critical software enjoys a plethora of methods that help verify and validate software (abstract interpretation, model checking, simulation, bounded tests...), these methods are generally useless when it comes to Neural Nets.

This thesis aims at bridging formal software verification and machine learning, in order to bring trust in critical systems incorporating Neural Networks elements.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France