Formal Verification of Neural Networks

Bunel Rudy

Lady Margaret Hall
University of Oxford

A thesis submitted for the degree of
Doctor of Philosophy

Trinity 2019

Abstract

Machine learning models and in particular Deep Neural Networks are being deployed in an ever increasing number of applications, making it crucial for methods capable of verifying their behaviour to be developed. In order to go beyond treating them as black boxes, we study the feasibility of verifying that certain properties on the mapping that the models encode should always hold and propose algorithms to perform this verification. We advocate the developments of these methods through the lens of optimisation.

By reformulating the verification of properties as optimisation problems over Neural Networks, we introduce a unified framework to reason about algorithms and identify the necessary components of verification systems. An attempt at verifying a property either results in a formal proof that the property holds or in concrete examples of cases where it is violated.

Current state of the art in verification technology of machine learning artifacts is constrained by both the restricted applicability of existing techniques and by their limited scalability. With the aim of making it more feasible to prove statements over widely used models, we present general methods that increase the diversity of verifiable properties and architectures amenable to verification. In order to get closer to the scale of models that power industrial applications, we refine heuristics used during the proving process and introduce algorithmic improvements to the optimisation algorithms underlying the methods, leading to order of magnitude speedups. Benchmarks empirically validate our runtime improvement claims, showing our contribution to addressing the problem of limited scalability of formal methods in the context of Neural Networks.

Formal Verification of Neural Networks



Bunel Rudy

Lady Margaret Hall

University of Oxford

A thesis submitted for the degree of

Doctor of Philosophy

Trinity 2019

Acknowledgements

They say it takes a village to raise a child, but it takes almost as much to finish a DPhil. I would never have accomplished this endeavour without the support from a lot of people, first of all my supervising team. I am extremely thankful for the freedom they granted me, always supporting me through my frequent changes of subject, while still encouraging me to actually finish things. **M. Pawan Kumar**, more than anyone else, was responsible for me reaching the end of this PhD. He provided daily mentorship at the beginning and constant support throughout. Even before me reaching Oxford, he is the person that convinced me that I was able to do a PhD and that I should. Four years later and here we are. Starting from the classes he taught during my master, in every conversation we had whenever I felt stuck, in every reading group, I always came out with a better understanding of the problem and a renewed enthusiasm for research. I also learned immensely from him about scientific writing and how to run presentations. **Philip Torr** was a great mentor to have. He was relentlessly insisting that I should be making the most of my time in Oxford, regularly feeding me Catz lunches, and always reminding me to not lose track of the problems we were trying to solve rather than getting bogged down on the niche technical problems that I found interesting. **Pushmeet Kohli** was instrumental in broadening my perspective. He instilled in me that one of the most important part of science lies in communicating your results and making them useful for other peoples. He also repeatedly opened new doors for me to experience research and seminars in different places.

I was indeed lucky to intern with great people. At Microsoft Research, **Jacob Devlin**, **Rishabh Singh** and **Matthew Hausknecht** gave me my first taste of large computational budgets. They shared their language model and program synthesis experience with me and gave me my first (and luckily only) machine learning projects. The Robust AI team at Deepmind (**DJ**, **Sven Gowal**, **Robert Stanforth**, **Jonathan Uesato**, **Chongli Qin**, **Chenglong Wang**) had the same interest I had in Neural Network verification, and shared the vast amount of work they were doing on it with me, making my internship there one of my best summer. I can not wait to go back to working with them.

Anyone that ever met me during my time in Oxford would know the importance that college rowing had in my life. It's unclear what the net effect of my **LMHBC** involvement on my DPhil has been. Despite the tired brain and shortened lab hours caused by training, the increased productivity from relieving all stress and the gained discipline to structure my life around probably balanced it out. Beyond all this, during my time here, LMHBC has been my social circle, my family, my bedrock. I wouldn't have lasted as long as I did in Oxford without them. I want to start by thanking the coaches that gave me the gift of rowing: **Tom Middleton** that taught me to row, **Sammi Chekroud** that taught me to row better, the Brookes gang of **Susie Dear**, **Grace MacDonald** and **Brendan Edwards** for pushing me to push hard and **Anna Carbery** for transforming our crew into a division 1 contender. Beyond everything, I am grateful to my crewmates with

whom I had the pleasure to share boats, from the 2015 novice crew that went on to earn blades to the history making Summer Eights crew of 2019. In particular, **Chris Liang** and **Sam Rowan** for always sharing their drive to improve and never accepting subpar rowing, **Rob Dickens** for being so easy to follow, **Ollie Watson** for the example he was setting and his dedication that often kept the crew running, **Giacomo Arrighini** for being a voice of reason in the boat, **Ben Moore** and **Jimmy Brebner** for always lightening up the mood on and off the water, **Owen Sweeney** for his relentlessly positive attitude and for laying down the rhythm with me, **Harry Ward** and **Ben Gregson** for so often sharing my training sessions, whether on the erg or the double, **Emily Cook** and **Esther Rollinson** for steering our boats even at the cost of a few fins, **Jack** and **Sean Kelly** for their contribution in making the boat club such a welcoming place. The morning were early, the ergs gruesome but I'd do it all again in a heartbeat.

My time at the Engineering Department was made infinitely better by all the friends. The French mafia of the Oval Group (**Alban Desmaison**, **Leonard Berrada**, and for a time **Diane Bouchacourt**) was always a little bit of the homeland with which to share a chat. I also enjoyed working in partnership with **Jingyue Lu**, **Florian Jaeckle** and **Alessandro de Palma** at different times and took great pleasure in starting my PhD with **Pankaj Pansari**. A discussion with people from the Torr Vision Group (particularly **Nantas Nardelli**, **Siddharth Narayanaswamy**, **Jimmy Shi**, **Thalaiyasingam Ajanthan**) always resulted in learning something or getting my spirit lifted, often both.

The Union Street house (**Alban Desmaison**, **Alice Stelfox** and **Sam Watson**) was always a pleasure to return to, the formal and home-cooked dinners we regularly had never failed to make life more enjoyable. Frequent restaurants companions and drinks sharers **Bea Rea**, **Tariq Hussain**, **Ollie Ballinger** and for a time **Keitumetse-Kabelo Murray** are also to be thanked for so often improving my mood after a day of work. The service of **Steven the sourdough** should also not be neglected, for all the time he fed me.

Outside of Oxford, I also benefited from the remote support of a large amount of people. From France, **Paul Dib** and **Karine Monpravad** always opened their home to me whenever I was in Paris, the Abitur gang (**Perrine Picard**, **Sophie Delahaye**, **Agnes Blesch**, **Nelson Dijoux Coquillas**) welcomed me as if we never separated each time we reunited. With **Marine Schweitzer**, I never had to reunite because she was always available for a call or a visit, despite the Channel separating us. Old buddies **Marwan ElBouss** and **John Reen**, as well as new found friends **Aimée Plutsick** and **Maud Stiegler** played a massive role in making me get out of this little Oxfordshire country town and seeing the beauty and culture of London. **Nefeli Custer** should be lauded for putting up with me in all circumstances, encouraging me in all occasions and pushing me to actually start to write this thesis. My parents, **Nicole and Claude Bunel** offered unwaivering support for the four years it took me to graduate, only occasionally inquiring when would I get a real job.

I gratefully acknowledge the Microsoft Research studentship that funded my first three years and the Santander scholarship that supported my last year.

Abstract

Machine learning models and in particular Deep Neural Networks are being deployed in an ever increasing number of applications, making it crucial for methods capable of verifying their behaviour to be developed. In order to go beyond treating them as black boxes, we study the feasibility of verifying that certain properties on the mapping that the models encode should always hold and propose algorithms to perform this verification. We advocate the developments of these methods through the lens of optimisation.

By reformulating the verification of properties as optimisation problems over Neural Networks, we introduce a unified framework to reason about algorithms and identify the necessary components of verification systems. An attempt at verifying a property either results in a formal proof that the property holds or in concrete examples of cases where it is violated.

Current state of the art in verification technology of machine learning artifacts is constrained by both the restricted applicability of existing techniques and by their limited scalability. With the aim of making it more feasible to prove statements over widely used models, we present general methods that increase the diversity of verifiable properties and architectures amenable to verification. In order to get closer to the scale of models that power industrial applications, we refine heuristics used during the proving process and introduce algorithmic improvements to the optimisation algorithms underlying the methods, leading to order of magnitude speedups. Benchmarks empirically validate our runtime improvement claims, showing our contribution to addressing the problem of limited scalability of formal methods in the context of Neural Networks.

Contents

1 Introduction **1**

1.1 From Programs to Models 1
1.2 Success of Machine Learning 2
1.3 A Need for Verification 3
1.4 Formalism of the Verification Task 5
1.5 Challenges of Verification in Machine Learning 6
1.6 Summary of Thesis Contributions 7
1.7 Contributions to other subjects 8
1.7.1 Neural Network for Program Synthesis 8
1.7.2 Optimization of dense conditional random fields 9

2 Literature Review **11**

2.1 Unsound verification 11
2.1.1 Adversarial Attacks 12
2.1.2 Probabilistic Verification 13
2.2 Incomplete Verification 14
2.2.1 Linear relaxation of networks 15
2.2.2 Semidefinite Programming bounds 18
2.2.3 Lipschitz Bounds 19
2.3 Complete 20
2.3.1 Enumerative Methods 20
2.3.2 Mixed Integer Programming 21
2.3.3 Satisfiability Modulo Theory 22
2.3.4 Branch and Bound 22
2.4 Specification 24

3 A Unified View of Piecewise Linear Neural Network Verification **25**

3.1 Preamble 25
3.2 Introduction 26
3.3 Problem specification 27
3.4 Verification Formalism 28
3.4.1 Verification as a Satisfiability problem 28

3.4.2 Mixed Integer Programming formulation 29
3.5 Branch and Bound for Verification 30
3.5.1 Reluplex 32
3.5.2 Planet 33
3.5.3 Potential improvements 35
3.6 Experimental setup 36
3.6.1 Methods 37
3.6.2 Evaluation Criteria 38
3.7 Analysis 38
3.8 Conclusion 41
3.9 Additional Notes 42
3.9.1 Toy Problem example 42
3.9.2 Link between the MIP formulation and the Planet relaxation . . 44
3.9.3 MaxPooling 46
3.9.4 Mixed Integer Variants 47
3.9.5 Additional performance details 51
3.9.6 PCAMNIST details 52

4 Knowing When to Stop:
Evaluation and Verification of Conformity to Output-size Specifications 55
4.1 Preamble 55
4.2 Introduction 56
4.3 Related Work 58
4.4 Modulating Output-Size 59
4.4.1 The Output-Size Modulation Problem 61
4.4.2 Extending Projected Gradient Descent Attacks 62
4.4.3 Continuous relaxation for discrete inputs 62
4.5 Verified Bound on Output Length 63
4.6 Target Model Mechanism 65
4.6.1 Image captioning models 65
4.6.2 Machine translation models 66
4.7 Experiments 66
4.7.1 Details of models and datasets 67
4.7.2 Adversarial Attacks 68
4.7.3 Verification 71
4.8 Conclusion 72
4.9 Additional Notes 73
4.9.1 NMT Examples 73
4.9.2 Toy Example 74

4.9.3 CNN Embedding Encoding 74
4.9.4 RNN Cell Encoding 76
4.9.5 Argmax Encoding 76
4.9.6 Vocabulary Embedding 77

5 A Proximal Solver for Neural Network Bounds **79**
5.1 Preamble 79
5.2 Introduction 79
5.3 Related Work 81
5.4 Preliminaries 82
5.5 Algorithm 83
5.5.1 Lagrangean Decomposition of the Problem 84
5.5.2 Proximal Method for Optimisation of the Bound 85
5.5.3 Frank-Wolfe Algorithm 86
5.5.4 Alternative Method 89
5.6 Experiments 89
5.6.1 Implementation Details 89
5.6.2 Experimental setting 90
5.6.3 Results 91
5.6.4 Sensitivity to hyperparameters 93
5.7 Discussion 95
5.8 Additional Notes 95
5.8.1 Network Architecture 95
5.8.2 Sigmoid Activation function 95
5.8.3 Algorithm summary 99
5.8.4 Comparison between the duals 100
5.8.5 Initialisation of the dual 103

6 Discussion and Future Work **107**
6.1 Methodology 107
6.2 Scalability 108
6.3 Generality 109
6.4 Incorporating Properties into Training 110
6.5 Theoretical understanding 111

References **113**

1 Introduction

1.1 From Programs to Models

Since the first developments of computers and information systems, the application domains of automation have kept expanding. Early on, machines such as Babbage's "Difference Engine" or the first electromechanical analog computers had as their only job to evaluate mathematical formulas, hence the name "computer" that has lasted to this day. Centuries later, new application domains keep being tackled and this departure from well defined mathematical problems has lead to a requirement to be able to handle ever less well defined specifications.

Nowadays, through scientific improvements in algorithms, organizational developments to tame large scale complexity, and increased computing power, tasks that would have been once considered infeasible are now regularly attempted.

If initially, using a computer to solve a problem required having human experts capable of specifying a procedural way of accomplishing the target goal, that is no longer the case. Declarative programming and the advent of logic solvers and program synthesizers lead to a first breakthrough, making it possible to generate programs based on a complete, formal description of the task to accomplish. Using this specification, solvers were capable of providing an implementation satisfying the requirements. Users still had to specify the description of their objective, but got freed from having to think about how the solution should be reached [109]. Widely used, successful application of this can be found in regular expression engines, or SQL query planners. Going one step further, machine learning is promising to do away with having to provide a complete specification of the desired behaviour, only requiring examples of desired

behaviour or a way of scoring results, and using *learning algorithms* to generate a model producing the desired behaviour.

The way this is accomplished is by generating a mapping from the task inputs to its outputs or to actions to take. These mappings are parametric functions, with the parameters completely controlling the behaviour of the mapping. By adjusting them, the outputs corresponding to each input are modified. The learning process amounts to estimating an optimal set of parameters according to some pre-determined criterion. The role of this criterion is to measure the difference between the outputs currently being generated and the desired one as given by the user, as well as potentially biasing the search towards values of parameters more likely to generalize, that is producing similar performance on data yet unseen.

One particularly successful way to achieve these results in recent years has been the methodology of Deep Learning [103]. The mapping from input to output is composed of a succession of differentiable building blocks. The criterion used, designated as "objective function" is usually the sum of a differentiable loss function and a regularization term. Taking advantage of the differentiability of the criterion with regards to the parameters, a gradient can be obtained so that improvement directions can be estimated, leading to an efficient search for the set of optimal parameters.

1.2 Success of Machine Learning

Thanks to several structural advantages, the machine learning approach to solving problem has been quite successful.

The learning algorithms can be applied to most objective functions as they rely on very few assumptions. Differentiability is one of the only requirements, but the use of surrogate loss makes tractable to solve even problems that are hard to formulate in differentiable terms.

Advances in GPU hardware and computing power in general led to very efficient and scalable implementations of the learning process, making it easy to tackle large problems.

The requirement on only data, rather than on a formalized specification also allowed to drive down the amount of effort required to solve a problem. Indeed, while providing a procedural solution or a formal specification of a problem can often only be achieved by domain experts, annotated data can be cheaply obtained from human annotators on websites like Mechanical Turk [47, 142], by crawling large internet archives [34, 115, 148] or even directly through simulation [149].

As a result, machine learning has achieved great success in a wide variety of domains. Computer vision tasks such as handwriting recognition [102], image classification [100],

object detection [67], semantic segmentation [111], stereo matching [113] are on their way to be considered essentially solved. Speech recognition [72] and synthesis [127] systems using Neural Networks are nowadays available on all smartphones. Translation [13] or text understanding [82] systems are constantly getting improved. Performance beyond the capability of the best humans were achieved in a wide collection of games such as backgammon [156], go [148], chess, shogi [149], video games [122] or poker [21]. Machine learning-based recommender systems [38] power the recommendations of most online websites. Even in domains populated by experts such as health care, deep learning methodologies have shown that in some cases they could contribute by analyzing results from test and triaging patients at a performance comparable to experts [46]. Due to those successes, there are attempts to bring those methods to an ever growing diversity of domains, such as protein folding [60], self driving cars [31], robotics [106, 104].

A traditional hallmark of the area where Machine learning methods have proven useful is in perception[72, 100]. Typical input data in this context is high dimensional, such as images or sounds captured by sensors. Humans will naturally understand what they see or hear as something semantically meaningful, but are not able to explicitly describe their perceptual reasoning. The flexibility of machine learning methods is here an advantage as technical experts are neither able to write programs to make sense of this data, nor to provide proper specification. Nevertheless, even non experts can easily give the answers that the model should produce, thereby allowing the easy collection of data to learn from.

1.3 A Need for Verification

At the same time, not requiring a specification and relying only on data for specification is dangerous. Different solutions giving the same results on the training data will be considered equivalent from the point of view of the learning algorithm but might be of completely different values when deployed into the real world and encountering previously unseen data.

As no behaviour was ever fully specified, unexpected outcomes can happen. Discriminatory or unfair predictions [9], and widely varying classifications [155] are examples of failures resulting from the lack of specification imposed. The learned model satisfies the requirement of behaving correctly on the data used for training but has no notion of fairness [15] or of the implicit expectation to produce consistent classification.

In addition, the artifacts produced by learning algorithms, and especially Deep Learning methods are not easy to understand, consisting only of the numerical values of all parameters. They are notorious for being “black boxes”, not open to interpretability. In order to be able to be deployed in actual systems, it is important for them to be able

to pass certifications procedures and for their designer to have the possibility of looking at what's happening under the hood. This is particularly true if the target applications are interfacing with users and might have life-critical implications.

Beyond the usefulness of verifying models to ensure that they are suitable for deployment, the ability to make definitive statements is also beneficial for research. The evolution of the status of adversarial examples in the machine learning community gives an example why. After [155] discovered the expectation defying fact that imperceptible perturbations to an image could fool state of the art models, a large part of the community attempted to resolve this problem. [129] introduced the defensive distillation method, claiming to solve the problem with his method diminishing the success probability of adversarial attacks down to 0.5%. However, it was later shown that by using stronger attacks [29], adversarial samples could still be found with a perfect success rate. The machine learning community developed several schemes [19, 69, 73, 81, 120] to detect adversarial samples and therefore prevent the model getting fooled. However, it was shown later that using attacks aware of the detection mechanism [28] could circumvent them. In order to solve the problem, some proposed combining multiple defenses in order to obtain higher robustness [1, 178] and claimed success. This lasted until it was shown that these defenses could also be circumvented [79]. At the ICLR 2018 conference, several new defenses were proposed[22, 51, 76, 105, 114, 153, 176], based on techniques such as randomization or generative models. But by the next conference, it was shown that all of them could be broken by using stronger adversaries [11, 160] using approximation of the backward pass or gradient-free optimization methods.

For several years, the cycle repeated itself with some researchers claiming that their methods eliminated this class of problem due to existing methods not being able to surface any, followed up by other work showing that tweaking and improving the attack methods resulted in the discovery that these problems still existed. This wasteful cat and mouse game was caused by the fact that due to the lack of methods allowing to formally verify their claims, scientists proposing new defense mechanisms could not be sure that their proposed strategies were actually effective. This highlights the need for methods capable to generate guarantees on the behaviour of systems, so that researchers can have the ability to make definite statement about their results.

Similar problems are not new and have already been tackled in the context of software engineering. Based on a formal specification of what a program should be doing, formal methods verify that the specification is met. Their use has helped to ensure safe deployment of systems, and guarantee properties such as the security of cryptographic protocols [119], the correctness of firmware implementations [78], the safety and compliance to requirements of avionics system [124]. The goal of this thesis is to advance the feasibility of achieving similar results when deploying machine learning powered systems.

1.4 Formalism of the Verification Task

Let us now clarify what we mean by verification of neural networks. As we discussed earlier in section 1.2, machine learning models are employed in situation where it is not feasible to describe formally what the correct behaviour should be. Therefore, we will not be able to prove that the model always produces the right results. Furthermore, this would be a futile endeavor, as in most cases, it is well accepted that neural networks are not able to achieve perfect results. Indeed, simply evaluating on the test set often reveal some incorrect decisions.

Even if we can not verify the full correctness of the neural network, it is worthwhile to deal with simpler sanity checks. When experts are not able to specify desired behaviors fully, they are often capable of defining classes of errors that should be avoided or particular cases in which the decision to take is evident. For example, while formulating a specification for an image classifier is impossible, it makes sense to stipulate that imperceptible perturbation to a known image should not change its classification. For a model implementing a controller, it is often possible to define a range of signals in which the output should lie.

A verification task is defined by a trained neural network, implementing a certain mapping from inputs to outputs, as well as a property to verify. The property represents the specific behaviour that we want to ensure the network has. It is defined in two parts. First, it contains a set of constraints on the input to define a domain of interest. This allows to restrict the applicability of the property to the part of the input space on which some knowledge about desired behaviors is available. Second, it also comprises a Boolean function of the network output that indicate if the prediction made by the network is compatible with what is desired. Based on these elements, the property to verify is that for any sample belonging to the input domain, the prediction made for it by the network should satisfy the Boolean function. The contrapositive of it is that there exist no sample in the input domain for which the prediction of the model lead to the Boolean function evaluating to false.

The verification task consists in proving this property, by showing that no input contained in the domain of interest will lead to a prediction that doesn't satisfy the Boolean function. If the property to verify is not true, then this goal is not achievable and a counterexample proving it should be exhibited to show why the property can not be verified.

1.5 Challenges of Verification in Machine Learning

There are however challenges that lies in the verification of Deep Neural Networks. Some of them are similar to those that can be found in classical verification using formal methods, while others are novel.

One common difficulty is the scalability problem. As in software verification, the statements that we want to establish are absolute and need to account for every possible input. This tends to introduce a combinatorial explosion on the number of configurations that need to be analyzed if smart methods are not employed. In verification of machine learning, this combinatorial explanation is usually dependent on the number of hidden units whose activation can not be pre-determined.

One significant challenge also lies in the flexibility and the diversity of networks that can be considered. It does not matter how efficient the verification algorithms are if they are only applicable to models that do not reflect the ones used by practitioners. This is a frequently leveled criticism towards verification algorithms that can only handle simple feed forward networks with ReLU activation function. There exists a trade-off between losing generality and leveraging more structure of the problem in order to gain efficiency.

Similarly, the limited ability of verification procedures to accommodate more diverse types of properties is at the moment the largest limiting factor to more widespread adoption. While significant progress has been made on training for and proving robustness to adversarial attacks limited to small noise perturbation bounded in magnitude, machine learning users are requesting the ability to deal with more semantically meaningful attacks models. For example, in computer vision applications, robustness to small rotations or translation are basic specifications that existing methods struggle to handle. The ambition would be to have the ability to prove that a driving scene understanding model is robust to change of weather or lighting conditions. However, doing so would require being able to specify the transformation that a change in weather would do to the captured image, which is not something that even experts are capable of formally specifying. Until we develop better way of specifying complex semantic restrictions on the input domain, the only solution available is to vastly overapproximate the domains in which we want to be robust, thereby creating a much more complex task for the neural network.

The common theme of the solution that we propose is to rephrase the verification problem as a search problem. We formulate an optimization problem to maximize the violation of the property to verify and then reach from the existing wide literature on optimization to develop practical solutions. Flexibility in network architecture and in specification will be achieved by expanding on the type of constraints that our optimization algorithms are able to handle, and scalability will be achieved by employing

and developing efficient optimization algorithms. We think that this framing of the problem offers a conceptual framework that can be used to better understand existing methods and improve upon them.

1.6 Summary of Thesis Contributions

We now outline the contributions in this thesis that attempt to solve the challenges of verification of neural network models outlined above.

We propose a unified framework, based on optimization, for neural network verification (Chapter 3). We formalize the verification problem for a wide class of properties on neural networks and show that formal verification of those properties can be achieved by performing global optimization on a non-convex problem. This problem is equivalent to finding the inputs leading to the minimal value for the output of a neural network of a similar structure to the one needing to be verified. We introduce a generic framework to be able to solve the optimization problem and show that existing work on neural network verification can be interpreted as specific instances of this framework. We also leverage this insight to develop other specific algorithms and show that we achieve significantly improved results on benchmarks. At the same time, we propose a comparison between already published methods that was so far lacking in the literature and attempt at identifying the factors causing hardness of verification. This work has appeared at NeurIPS 2018 [25].

We show how to perform verification on Recurrent Neural Networks with variable output lengths (Chapter 4). We study the problem of verifying that Neural Network models producing variable-length output will generate an output of bounded length, rather than never reaching termination. We formally specify the problem and build an algorithm inspired by adversarial attacks to highlight the weaknesses of model trained with standard methods. This allows us to show that they can be made to generate input significantly longer than what would be expected. We demonstrate how to encode robustness to these weaknesses as an optimization problem, and how it can be made feasible to encode variable length outputs. Compared to existing methods, this is feasible even when the model studied are autoregressive and therefore have activation dependent on previously made predictions. We conducted experiments on image captioning and machine translation to demonstrate the working of our methods. This work was done in collaboration with Chenglong Wang. This work has appeared at CVPR 2019 [163].

We show how to scale up the computation of bounds underlying Neural Networks verification methods (Chapter 5) Verification algorithms such as the ones described in previous chapters rely on the ability to compute bounds on the value than neural network outputs can take. We introduce an optimisation algorithm that provides several advantages over existing methods by being easily parallelizable, leveraging the structure of typical convolutional neural networks, producing anytime solutions and being applicable to various types of activations and input domain. This work is at the moment under review.

Additional Contributions In addition to the works discussed in this thesis, other contributions were made to published papers on the subject of Neural Network verification. Work lead by Qin et al. [135], accepted at ICLR 2019, extended the type of verifiable properties to allow for non-linear specifications, making it possible to specify properties over the probability distributions generated by a network or over quadratic formulas over the network outputs, therefore allowing the encoding of properties such as conservation of energy for network modeling physical systems. Work lead by Gowal et al. [71], accepted at ICCV 2019, showed that incorporating even simple bounding techniques into training procedures could lead to state of the art robustness against adversarial examples.

1.7 Contributions to other subjects

1.7.1 Neural Network for Program Synthesis

While not directly related to the topic of this thesis, several contribution were made to the topic of using neural networks in the context of program synthesis. If interpretability and safety concerns are critical, verifying the machine learning models used to make prediction is one option. Another possibility lies in using machine learning models to generate programs and then use existing software verification methodologies to perform verification and achieve interpretability.

We showed how to compile programs down to a neural network representation and how it is then possible to attempt their runtime optimization. We described a simple programming language whose execution can be mapped to the evaluation of a recurrent neural network and explain how simple programs can be compiled down to the weights of a neural network such that evaluating the network is equivalent to running the program. We showed that by optimizing a cost function including a penalty for the execution time over the network, we can recover data-dependent optimization of the initial programs. This work has appeared at NIPS 2016 [23].

We showed that machine learning can be used to improve the heuristics of existing superoptimizers. Existing superoptimizers operating on assembly language representations operate by performing a stochastic search over source code modification in order to find more efficient versions of existing programs. We show that by learning the probability distribution from which program transformations are sampled during the search, we can reach more efficient solutions with less iterations. This work has appeared at ICLR 2017 [24].

We showed that incorporating information about the target programming language structure and about the semantic correctness of generated programs improve the performance of neural program synthesizers. We identified a problem with existing methods for neural program synthesis[50] that resulted in penalizing programs that were not the reference programs from the dataset, even though they were semantically equivalent. We introduced a reinforcement learning method to allow for the optimization of the probability of generating a correct program, rather than the probability of matching a reference program. We also showed how the grammar of the target programming language could be included in the structure of the network or even jointly learned so as to improve results. This work appeared at ICLR 2018 [26]. At the same time, I also helped in categorizing and framing existing neural program synthesis and induction literature for the work lead by Devlin et al. [49].

1.7.2 Optimization of dense conditional random fields

I also contributed improvements to optimization algorithms for inference on dense conditional random fields[99]. Jointly with Alban Desmaison, we showed [48] that convex relaxations could be used for large scale problems and provided advantages when compared to other methods based on meanfield approximations, while still retaining their efficiency. I contributed as a non-primary author in extensions to our method to improve the solver of the Linear programming relaxation [2] and to add support for higher order potentials [90].

2
Literature Review

In this chapter, we aim to provide an overview of the literature on formal verification of neural network.

While the problem of neural network verification is quite general, a lot of the research has focused on the specific case of robustness to adversarial examples perturbations. Despite their narrow focus, these problems are still very relevant to the general task. By evaluating the property to verify over the outputs of the network, we can perceive the joint system to be a network outputting a classification on whether the property is violated or not. Verifying that there is no perturbation leading to a change of prediction is therefore equivalent to verifying that there is no counterexamples to the property so the general problem can be reduced to a robustness to adversarial examples problem.

More generally, as we will be discussing in Chapter 3, a very large class of verification properties can be reformulated as finding the optimal value taken by a neural network, also known as the bound computation problems. Methods used for proving robustness to adversarial examples can be applied to computing bounds on values taken by units of the network. In this literature review, we will therefore be discussing both the generic problem of verification and the one of certifying robustness to misclassification around a sample without establishing a distinction, chosing instead to structure our discussions around the properties and guarantees of each algorithm.

2.1 Unsound verification

A verification algorithm is deemed *sound* if the only property that it accepts as true are indeed true. An *unsound* verification algorithm may classify some properties for

which counterexamples exist as true. Soundness is a very desirable property, but it might make sense to trade it off in some cases, for example in order to allow the tackling of problems that are order of magnitudes larger, even if the results will not be able to be trusted to the same extent.

2.1.1 Adversarial Attacks

The most common examples of unsound algorithms in Machine Learning is the use of adversarial attacks on a model. A search algorithm, based on optimization, attempts to discover a counter-example to the property. If none are found, the property is assumed to be true.

The field of adversarial attacks has been widely studied, mainly in the context of proving robustness to small perturbations of the inputs. In this settings, the attacks attempt to find an input within a norm ball where the classification generated by a network differs from a reference one. The initial paper identifying the existence of adversarial examples [155] proposed to use the L-BFGS [108] algorithm to discover them. Goodfellow et al. [70] proposed a fast heuristic named the Fast Gradient Sign Method based on taking steps according to the sign of the coordinates of the gradient and discarding their magnitude. More complex methods were later proposed, achieving higher success rate in discovering counterexamples, such as Deepfool [123] based on iteratively linearizing the objective function around the current point and solving the resulting linear problem. Madry et al. [116] formalized the use of the standard method projected gradient descent as an adversarial attack and conjectured that it represents an optimal first order attack. This was later disputed by Wang et al. [163]. By using gradients of a different objective function based on a bounding over-approximation of the network rather than simply on an encoding of the property, they showed that more adversarial examples could be discovered.

Carlini and Wagner [29] introduce attacks specifically tailored to different types of input domain restriction, but also provide a survey and experimental comparison of the earliest methods proposed in the literature. A more recent survey was published by Yuan et al. [182]. In addition, high quality software packages [126, 128] exist, providing implementations for most frameworks. Outside of gradient based adversarial attacks, some other alternatives with similar properties were proposed. Papernot et al. [130] attempt to generate counterexamples by reusing counterexamples generated on other networks, while Chen et al. [33] use coordinate descent for its search. These methods have the particularities that they don't require the ability to obtain gradients of the network or even to observe anything beyond the predictions.

All these methods suffer from the same problem: due to the non-convexity of neural networks, these optimization algorithms have no guarantees to converge to their optimal solution. As a result, it is impossible to distinguish between a network satisfying the property to verify, for which no counterexample exist and a network for which counterexample have not been found, despite them existing.

By restricting the search to a region of the space in which the network is convex, it is possible to guarantee the optimality of the counterexample found [17]. This approach is, however, still unsound because it only proves the property on a small restriction of the whole space, ignoring the rest. It is therefore possible that counterexample exists outside of the restriction.

The methods discussed in this section will possibly return counterexamples to the property, thereby proving it to be false. Conflating the absence of generated counterexamples and a proof that property is correct results in an unsound verification algorithm. While the failure to invalidate the property should make us update our beliefs on the result, it does not provide any guarantees.

2.1.2 Probabilistic Verification

Another type of verification algorithms which lead to unsound results are verification methods for which the results hold only with a certain probability.

CLEVER [168] uses sampling of the gradient for points in the region of interest and use maximum likelihood estimation to obtain the parameters of a reverse Weibull distribution. A Lipschitz constant is derived from it, which can then be used to verify properties by bounding the possible variation around sample points. The method was later extended to recurrent neural networks by Ko et al. [97]. Another sampling based approach [167] used a Monte-Carlo algorithm named adaptive multi-level splitting [77] to estimate the probability of a counterexample existing. Wong et al. [172] presented an extension to their existing sound bound computation method [171] that replaced the computation of some expensive terms with an estimation based on Cauchy random projections. Because these methods are sampling based, they can not provide any guarantees.

Randomized smoothing based models are a specific type of models that loan themselves well to verification. The prediction is based on a smoothed version of another model, obtained by taking a majority vote in a neighbourhood around the sample to evaluate with regards to the prediction to make [43]. In order to estimate the results of this majority vote, sampling is a requirement, both for evaluating the prediction of the neural network and for verifying any property. Better bounds can be obtained if adversarial training has been incorporated at learning time, as shown by Salman et al. [144], but the verification procedure remain dependent on sampling.

While most of the sampling based methods provide probabilistic guarantees, or a way to estimate the probability of an incorrect prediction depending on the number of samples used, they are still not able to provide completely sound verification.

Outside of sampling based probabilistic bounds, there also exist deterministic bounds that only hold in expectation. A modified version of Interval Bound Propagation [71] follows this strategy [5] and advances some theoretical results that for network whose weights have gaussian independent and identically distributed values, the resulting bounds are in expectation valid bounds of the neural network, while at the same time being tighter than rigorous, over-approximation based bounds. Despite experimental signs that these bounds might be useful, the analysis of their correctness holds only in expectation, which means that it is possible to exhibit examples where they are incorrect. In addition, the analysis only holds under extremely strong conditions which might not be realistically applicable.

Unsound verification methods are often significantly more scalable than other algorithms, but this comes at the cost of not being able to provide any strict guarantees. While this trade-off might be occasionally worthwile, for example to use as a training signal [116], it exposes the user to sometimes draw incorrect conclusions [11, 160]. Using unsound methods lead to overly optimistic estimates of the verified status of a property. Another approach to achieve scale is the pessimistic approach, based on *sound, but incomplete* methods.

2.2 Incomplete Verification

A verification algorithm is deemed *complete* if it can prove any true property. This is a very strong requirement, as certain properties might require almost exhaustive enumeration to prove. Most complete methods are computationally expensive. *Incomplete but sound* methods will be able to prove some, but not all of the true properties. All properties that are false, as well as some true properties that the algorithm is not strong enough to prove, will have an unknown status.

One example of incomplete verification is the work of Croce et al. [45]. Their approach shares strong similarities with the unsound method of Bastani et al. [17]. The difference lies in the fact that Croce et al. [45] not only measure the closest distance from the sample to the decision boundary, but also measure the distance to the boundary of the convex region where the restriction of the network to a linar function is valid. As a result, their method can only provide proofs for properties where analyizing this small region is sufficient (so it's an incomplete method, not capable of proving all true properties). It has however the advantage of being sound. Combined with training improvements

that lead to increasing the size of the region where the training applies, this allows to verify a large number of robustness properties.

Most incomplete methods in neural network verification are based on the principles of bounding. The set of possible outputs of the neural network is soundly over-approximated and this set is compared to the property to be validated. On the one hand, if there is no intersection between the superset of the values that the neural network output can take and the values that violate the properties, this implies that no counterexample can exist and the property is therefore verified. On the other hand, if those two sets have a non-empty intersection, it is not possible to determine if the elements of that set are actual counter-examples which the network could generate, or if they are simply consequences of the over-approximation of reachable values. Incomplete methods differ by how they generate the over-approximation. The best algorithms are those generating as small as possible of an over-approximation set as possible.

An example of this approach is the computation of an over-approximation to the output reachable set [173]. Layer by layer, an upper bound on the difference between activations reachable due to a perturbation and a reference activation is obtained. Combining the bounds for all the layer, this results in a bound on the value that the output of the network can take, which can then be compared to the property to verify. In order to avoid too loose of an over-approximation, Xiang et al. [173] propose to tile the input space and propagate bounds separately for each tile, thereby limiting the over-approximation.

Another strategy to limit the over-approximation, and therefore boosting the success rate of the verification method, is to use tighter bounds even though they might be more expensive to derive. One large class of bounds are those derived from linear programming.

2.2.1 Linear relaxation of networks

For a neural network, fixing the input results in fixing all activations and outputs, as they will be constrained by the forward propagation rules. In the linear programming relaxation of the network, a linear program is built where each variable will model an input, an activation or an output. Based on the structure and weights of the network, constraints are established that limit the values that variables can take. For example, the activations before and after a linear layer are linked by linear equality constraints. Non-linearities of the network can not be exactly encoded in the formalism of a linear program, but it is possible to provide a tractable relaxation.

Originally proposed by Ehlers [58] as a component of its satisfiability modulo theory based solver, the Planet relaxation replaces the non-linear constraints binding together the input and output variables of ReLU non linearities by a set of linear constraints encoding the convex hull of the non linearity. Because the convex hull of the ReLU is larger than

the actual feasible set, some assignments to the variables of the resulting linear program are feasible that could not be obtained by the activations of the network. On the other hand, given that the convex hull is a super-set to the feasible set, any activation pattern of the neural network is necessarily a feasible solution of the linear program.

As a consequence, the resulting linear program has all possible assignments to the activation of the neural network as feasible solutions. As a linear program is necessarily convex, we can always find the optimal solution when maximising for the largest value reachable by a variable. The resulting value provides us with a guaranteed bound for this variable. Indeed, if there was an input of the neural network for which a larger value could be reached, this would represent a feasible solution that the linear program could reach.

Optimizing over the linear programming relaxation of the network therefore allows us to obtain bounds over the values reachable by the network. These bounds are not tight due to the looseness introduced by the convex hull approximation of the ReLU, which means that we will sometime be unable to prove a property based on the relaxation.

Another inconvenience is that to define the linear program, we require bounds on all the activations of the network. While the optimization methods described provide a way to obtain them, it is required to perform two optimization passes per hidden unit of the network. As modern networks can have more than tens of thousands of hidden units, this is computationally expensive. Several looser but faster versions were introduced to alleviate this.

One approach is based on Lagrangian duality. All linear programs admit a dual linear program. When working in the dual formulation of a problem, any assignment to the dual variables results in a valid bound on the value of the optimization problem, with the optimal choice of dual variables resulting in a bound equivalent to the one given by solving the original primal problem optimally. By taking the dual of the linear problem of the Planet relaxation, it is possible to obtain bounds on the value taken by the neural network without having to solve any optimization problem. Wong and Kolter [171] introduced a heuristic, non-optimal solution that could be obtained in closed-form and performed extremely well in practice. It was further refined to handle more complex neural network architectures [172]. Similarly operating on the dual of the optimization problem, Dvijotham et al. [53] performed optimization using a fixed number of steps of an iterative algorithm to derive good solutions. Due to the fact that it operates on the dual problem, all iterates of their method generate valid bounds, allowing them to balance tightness with computation time. An alternative method by the same authors attempted to use an auxiliary verifier network to predict a good choice of dual variables [54].

Rather than tackling the complex problem and limiting ourselves only to suboptimal solutions, another approach consists in simplifying the problem until it can be solved

without requiring optimization. The FastLin algorithm [169] is based on replacing the convex hull of the ReLU in the Planet relaxation by pairs of parallel linear constraints. While this is a looser approximation, leading to potentially less tight bounds, this results in a problem that can be solved in closed form. Interestingly, this method is equivalent to the heuristic solution to the dual problem proposed by Wong and Kolter [171]. This method was subsequently superseded by Crown [184], which used the same type of relaxation but without requiring the RelU-approximating constraints to be parallel, thereby allowing the obtention of tighter bounds. Crown also discussed how to handle additional activation functions such as sigmoid or hyperbolic tangents. CNN-Cert [20] is a method based on the same principles but explicitly supporting elements such as convolutional layers, residual connection or pooling layers. Ko et al. [97] also showed how this procedure could be applied to recurrent neural networks.

Abstract interpretation methods [64] create abstract element of various type representing the reachable states after each layer of the network. For each type of layer, they describe the transformation required to emulate the forward pass through the network. Abstract elements vary in complexity, ranging from the simplest Box domain, for which propagation is equivalent to simple interval bound propagation [71], to the polyhedra domain, which would be capable of being as tight as the Planet relaxation. One specific abstract element that proved particularly useful has been the Zonotope domain. Zonotope are a restricted class of polyhedra whose special structure allowed efficient computation of bounds. Several iterations of improvements to the transformers employed with zonotopes were published, introducing new methods leading to tighter approximations of the reachable states. Mirman et al. [121] initially presented several transformers, which were superseded by the ones proposed in the DeepZ systemSingh et al. [150], based on over-approximating activation functions by two linear functions with parallel slopes. Unsurprisingly, these transformers generated a relaxation exactly equivalent to the FastLin relaxation of [169]. A finer relaxation, equivalent to the relaxation of Crown [184] was obtained [151] by using linear functions with different slopes to approximate the activation functions, at the cost of requiring a domain more complex and therefore more memory intensive than the Zonotope.

All the methods described in these section are looser version of the Planet relaxation introduced by Ehlers [58]. Salman et al. [143] attempted to provide a detailed analysis on the limitations of this approach. To do so, they implemented the relaxation of Ehlers [58], which is the tightest relaxation for ReLU non-linearities, based on some of the tightest bounds possible, the ones derived from the linear programming relaxation. The goal was to evaluate what could be achieved if computational cost was not a consideration and the "optimal" approach was employed. Despite using this tight and expensive to compute

relaxation, they showed that some of the properties that they were considering still could not be proven true. They used this result to claim that there was "an inherent barrier to tight verification" for methods based on convex relaxations. This result is expected as convex relaxations rely on over-approximation, and therefore can not be expected to be complete.

It is also worth noting that despite some of their claim, their suggested method is not optimal. Other methods to handle non-linearities, such as the relaxation of the ideal formulation of Anderson et al. [8] or SDP-based relaxations [136] can provide tighter bounds than the "optimal" relaxation that Salman et al. [143] consider. Even amongst purely linear-based methods, techniques performing convex-hull relaxations on groups of ReLU rather than on each ReLU independently, like the work of Singh et al. [152], allow to obtain better results.

2.2.2 Semidefinite Programming bounds

Semidefinite Programming (SDP) relaxations can be thought of as an extension of Linear Programming, where in addition to linear inequality constraints applied to vectors, it is also possible to use matrix inequalities to enforce positive semidefiniteness of some matrix of variables. Semidefinite programs usually produce tighter bounds than linear programs because they are able to capture correlations between different activations that linear programming does not. Raghunathan et al. [136] was the first to propose a semidefinite programming relaxation, applicable only to networks with a single hidden layer. This restriction was later lifted [137] to allow standard multi layer neural networks.

In addition to producing tighter bounds, semidefinite programming relaxation of neural networks have the advantage that, contrarily to linear programming based approaches, they theoretically do not require the computation of intermediate bounds to formulate the relaxations. In practice, these intermediate bounds are always computed and incorporated through additional constraints, resulting in tighter bounds.

Despite these advantages, semidefinite relaxations are not often employed. Even though the problem is convex and polynomial time algorithms exist, they are prohibitively expensive to solve for instances of the size of standard neural network. A specialised algorithm for efficiently deriving bounds based on the dual formulation of the relaxation of Raghunathan et al. [137] was proposed by Dvijotham et al. [56]. By restricting the search for dual variables to a specific subspace, they make the problem amenable to projected gradient based optimisation, allowing them to tackle instances several order of magnitude larger.

Outside of producing tighter bounds, semidefinite programming relaxation are also more expressive. They have been employed to encode a richer type of constraints, such as one involving quadratic terms of the output of the network [135]. This was for example

used to verify that a network modelling the evolution of a physical system had learned to respect the conservation of energy.

The incomplete method that we have presented so far all rely on the same principle: encode some constraints that assignments to the activations of the network should respect and perform optimization to derive bounds on the output of the network. This approach forces us to explicitly model all intermediate activations of the network. An alternative consists in bounding the rate of variation of the network output. By combining this limit and observed output values, output bounds can also be derived.

2.2.3 Lipschitz Bounds

As neural network are composition of Lipschitz continuous functions, they are Lipschitz continuous themselves. If we can identify the Lipschitz constant of the network, we can use it to prove statements in the neighbourhood around samples. The smaller the Lipschitz constant we can derive, the larger the neighbourhood we can make statements for will be. In their seminal paper identifying the presence of adversarial examples, Szegedy et al. [155] also showed that a simple way to derive a Lipschitz constant of the network was to simply multiply the Lipschitz constant of each layer. Similar layerwise decomposition were employed by others[42, 132, 159] to obtain guaranteed lower bounds on the minimum perturbation necessary for samples to be misclassified.

Rather than computing a global Lipschitz constant holding for the whole of the input space, it is often beneficial to compute only a local Lipschitz constant restricted to the region of interest which might be a lot tighter. How to evaluate this local bounds was first shown for neural networks with a single hidden layer [80], but leveraging the linear relaxation described in Section 2.2.1 allowed Weng et al. [169] to obtain local Lipschitz bound for multi-layer neural networks with their FastLip method. They later extended the method [185] by showing that a recursive procedure could be employed to obtain significantly tighter bounds on the Lipschitz constant at the cost of some additional computation.

All the methods mentionned in this section compute conservative upper bounds of the Lipschitz constant. Raghunathan et al. [136] showed that even in the case of a simple neural network with a single hidden layer, evaluating the optimal Lipschitz constant was equivalent to solving the MaxCut problem, which is known to be NP-hard [91]. Overestimating the Lipschitz constant leads to more conservative bounds being produced.

Rather than having to estimate the Lipschitz constant at verification time, an alternative is to learn models for which it known by construction. By using a specific type of activation function and an adapted training procedure, Anil et al. [10] obtained networks

with a Lipschitz constant of 1. As a result, they don't need to perform any evaluation of the Lipschitz constant and can easily derive bounds from observed margins.

In order to guarantee soundness while still maintaining computational tractability, the incomplete methods that we have been describing in this section have to sometimes be overly conservative, leading them to be unable to prove certain properties. As a result, using them may lead to pessimistic estimates of the status of properties.

While the unsound methods we described in Section 2.1 can prove certain properties to be false, and incomplete methods described in Section 2.2 can prove certain properties to be true, there remain properties for which none of these methods can identify the status. In order to resolve the ambiguity, *complete* methods are required.

2.3 Complete

Because they are required to always return a solution, complete method needs to handle the whole region on which properties needs to be verified. Because they need to be able to prove any true property, they can not rely on overapproximation to maintain soundness and need to model the neural network accurately enough. For these reason, they are extremely computationally expensive and are rarely applied beyond small neural network or extremely restricted input domains.

This intuition is formally backed by theoretical complexity analysis. Katz et al. [94] shows that the verification problem is NP-hard by reducing it the 3-SAT problem. Furthermore, Weng et al. [169] showed by reduction to the set-cover problem that even computing bound with a bounded approximation ratio was not possible in polynomial time.

2.3.1 Enumerative Methods

One feasible approach to guarantee completeness is through exhaustive enumerations. Huang et al. [85] proposes to discretize the space of possible inputs and perform a finite exhaustive search over the possible neural network inputs. This is however limited in terms of applicability because it is dependent on being able to enumerate the input space.

A more general method consists in observing that by construction, neural networks with ReLU activation function implement piecewise linear function. These piecewise linear function are composed of a necessarily finite set of linear function defined over polytopes. This can be shown by the fact that if we fix the state of each ReLU in the network, the network implement a linear function, and that all constraints necessary to force the ReLU to their desired states are also linear constraints. It is therefore possible to enumerate those pieces and perform the verification of any desired property for each piece separately [174]. Once dealing with a single linear piece, verification is easy due

to the convex structure of the problem. Jordan et al. [89] leverages the same insight but rather than performing an enumeration and then verifying all components independently, they grow the verified region iteratively, adding polytopes one at a time.

2.3.2 Mixed Integer Programming

While an explicit enumeration of the pieces of the neural network is a valid approach, it is also possible to rely on tools to enact the required systematic search. In this case, the enumeration is implicit. One flexible formalism in which we can encode the problem is the one of Mixed Integer Programming (MIP). A MIP is an optimization problem, similar to a linear program in that it is composed of a linear objective functions and linear constraints, but with the addition that some of the variables can be restricted to be integers.

The first encodings of neural network into the MIP format were proposed by Cheng et al. [36] and Lomuscio and Maganti [110]. Compared to the linear programming formulation that we discussed in Section 2.2.1, the ReLU constraints are not approximated but encoded exactly, with the help of an auxiliary binary variable and the so-called big-M formulation [74]. The big-M encoding of the MIP was also used by Fischetti and Jo [62] even if not explictly, as they relied on high-level features of their solver to generate the big M encoding.

The big-M formulation allows to encode the non-linearity of the activation function, but requires the computation of intermediate bounds, whose tightness will determine how easy to solve the MIP is. While the first methods[36, 110] generated bounds using interval bound propagation, Tjeng et al. [157] showed that refining some of the intermediate bounds using the linear programming relaxation of Ehlers [58] was beneficial. Singh et al. [151] made a similar observation. Their method achieves good performance by combining zonotope based incomplete bounds computation methods to obtain initial intermediate bounds [150] and refining some of them further based on a heuristic. The refinement is achieved by either solving the full linear programming relaxation of Ehlers [58] or the MIP formulation of a subset of the network.

In addition to the tightness of the intermediate bounds [151, 157], another factor which was observed to strongly influence the speed of operation of the solver was the way the model had been trained [157, 175], with network trained to be robust being more easily verifiable.

In an attempt to speed up the search based on the MIP solver, Sherlock [52] combines it with gradient based search. Optimization is performed using gradient descent as much as possible, but a MIP solver is employed to find if some additional improvements are possible over the best solution found, which would then be used to restart the gradient search.

Despite its simplicity, the big-M formulation is not necessarily the best choice for encoding the neural network verification problem. Anderson et al. [8] highlighted its looseness with simple examples and described a criterion to evaluate if a formulation of the non-linearity is ideal. They proposed several different formulations addressing it. While some are only of theoretical interest as they require a quadratic number of variables or an exponential number of constraints, they also introduce a simple cutting plane algorithm to iteratively add useful constraints. Their approach compares favorably to the commercial solver Gurobi, that most other method we discussed rely on.

2.3.3 Satisfiability Modulo Theory

Verification tools built on satisfiability modulo theory also follow the strategy of implicitly enumerating the possible linear pieces of the neural network. A formula in the theory of linear arithmetic encoding the existence of a counterexample to the property is defined. As opposed to incomplete methods, this encoding is exact and does not introduce any relaxation. The aim of the Satisfiability Modulo Theory (SMT) solver is to determine whether this formula is satisfiable and if so, find an assignment satisfying it. Compared to simpler enumerative methods, the difference lies in the fact that the SMT will detect infeasible partial assignments and therefore prune large regions of the search space, leading to improved running times.

Reluplex [94] was the first method to follow this approach. Throughout the search procedure, Reluplex maintains an assignment to all the variables representing the activation of the network, and attempts to resolve unsatisfied constraints. If no progress is being made, it splits the search space in two along the phase of a chosen ReLU unit. Impossibility of partial assignments is detected by simple bound propagation.

Planet [58] works in a similar manner except that it doesn't maintain an assignment to all the activation variables, operating directly over a binary representation of the pattern of ReLU phases. Satisfiability of partial assignments is measured thanks to the linear relaxation of the network described in Section 2.2.1.

As will be further discussed in Chapter 3, these SMT-based methods can also be interpreted as specific cases of the branch and bound framework.

2.3.4 Branch and Bound

Branch and bound is an optimization method to obtain the global optimum, even of non-convex functions. By applying it to the bound computation problem, we can obtain the exact set of reachable outputs and therefore perform complete verification. The approach consists in combining a bounding procedure, which is an incomplete verification method

and a branching procedure. The bounding procedure attempts to verify the property. In case of success, the property is proven. If the incomplete verification method is unable to verify the property, the branching procedure will divide the search space into smaller elements, for which verification will be attempted. By recursively dividing the search space, the overapproximation made by the bounding procedure is diminished, until the domains handled are small enough that the property can be proven on them. Internally, MIP solvers also use branch and bound algorithms. By designing solvers specialized to neural network verification, researchers hope to obtain performance benefits.

In addition to the algorithm we introduce in Chapter 3, other methods can be easily explained as instanciation of the branch and bound framework. The methods introduced by Rubies-Royo et al. [141] follows the same approach we do of using the Planet relaxation as an incomplete verification method and splitting over the inputs, but use information extracted from the dual variables to prioritise between possible branching decisions.

ReluVal [166] is based on an incomplete bounding method relying on interval analysis. Compared to traditional interval bound propagation [71], their method keeps track of the dependency on the input variables while going through ReLUs whose phase is fixed, leading to tighter bound estimate. This bounding method is combined with an refinement strategy consisting of splitting the input dimension into several subdomains, so as to guarantee completeness. The same authors later proposed another branch and bound approach named Neurify [165]. The incomplete bounding method used by Neurify corresponds to the one of Weng et al. [169], and the branching method splits over the ReLU non linearities, prioritising splits based on gradient information.

Another branch and bound method was suggested by Anderson et al. [6]. They used the zonotope based abstract interpretation incomplete verification method [64] for the bounding procedure, and perform branching over the input dimensions. One novelty of the work is that the choices made by their branching procedure were done by a learned policy.

Ruan et al. [140] also use a branch and bound based method, with a bounding procedure based on obtaining the Lipschitz constant of the network, similar to the ideas of Section 2.2.3 and a branching procedure based on recursively splitting the input domain so as to progressively obtain tighter and tighter bounds. It should however be noted that the bounding method that they use requires the method they use to obtain the Lipschitz constant is based on evaluating the rate of variation at a set of discrete points, which is not correct, which indicate that their method might not be sound.

The tool Never [134] does not strictly perform branch and bound but presents a lot of similarities with it. It relies on a incomplete verification method based on abstract interpretation [44], and iteratively refines the fineness of the over-approximation, until the property is verified or a counterexample is exhibited. While this does not present the

recursive splitting typical of branch and Bound methods and therefore does not benefit from coarse pruning of the search space, the rest of the method is extremely similar.

In addition to improving the scalability of methods or the tightness of incomplete methods, a productive area of research lies in proposing ways to encode more interesting verification problems.

2.4 Specification

Most methods described so far make the assumption that the restriction of the input domain on which we are trying to perform verification is easily represented in an accessible format, but some regions are not necessarily easy to encode. Balunovic et al. [14] tackled the problem of verifying robustness of image classification model to small geometric transformations. Their strategy is based on first bounding the reachable values for the perturbed input thanks to geometric analysis, and then leveraging other existing solvers.

Similarly, a weakness that most verification methods presented here presents is that they assume that the property is essentially something static, against which the output of the network can be compared. Akintunde et al. [3] extended the formalism to deal with the verification of properties over a system comprising a neural network based agent and an environment. By either assuming that the environment can be encoded as a MIP or approximating it by a feedforward network encodable as a MIP, their tool NSVerify is able to verify properties expressible in linear temporal logic [133] about the behaviour of an agent interacting with its environment. This was further extended by the toolkit RNSVerify [4] to be able to handle agents with a memory, implemented using recurrent neural networks.

Another innovation in the domain of specification can be attributed to Dvijotham et al. [55]. They define a formalism of verification for probabilistic model. As most probabilistic model operates partially through sampling distributions with potentially unbounded supports, standard verification can not provide meaningful results. Rather than providing guarantees that a property always hold, the objective becomes to verify the certain property will hold for any conditioning input and with high probability over the sampling of the latent variables. This requires the combination of bounding values taken by the network but also of bounding probability of unlikely events.

3

A Unified View of Piecewise Linear Neural Network Verification

3.1 Preamble

This chapter contains a paper originally published at NeurIPS 2018 [25]. As the field has evolved since the writing and submission of this paper, some newer references are relevant that are not discussed in this chapter. We refer the reader to Section 2.3 for a more up-to-date discussion of complete verification methods for neural network verification and more specifically to subsection 2.3.4 for an overview of all methods fitting in the framework of branch-and-bound verification.

The success of Deep Learning and its potential use in many safety-critical applications has motivated research on formal verification of Neural Network (NN) models. Despite the reputation of learned NN models to behave as black boxes and the theoretical hardness of proving their properties, researchers have been successful in verifying some classes of models by exploiting their piecewise linear structure and taking insights from formal methods such as Satisifiability Modulo Theory[58, 94]. These methods are however still far from scaling to realistic neural networks. To facilitate progress on this crucial area, we make two key contributions. First, we present a unified framework that encompasses previous methods. This analysis results in the identification of new methods that combine the strengths of multiple existing approaches, accomplishing a speedup of two orders of magnitude compared to the previous state of the art. Second, we propose a new set of benchmarks which includes a collection of previously released testcases. We use the benchmark to provide the first experimental comparison of existing algorithms and identify the factors impacting the hardness of verification problems.

3.2 Introduction

Despite their success in a wide variety of applications, Deep Neural Networks have seen limited adoption in safety-critical settings. The main explanation for this lies in their reputation for being black-boxes whose behaviour can not be predicted. Current approaches to evaluate trained models mostly rely on testing using held-out data sets. However, as Edsger W. Dijkstra said [27], "testing shows the presence, not the absence of bugs". If deep learning models are to be deployed to applications such as autonomous driving cars, we need to be able to verify safety-critical behaviours.

To this end, some researchers have tried to use formal methods. To the best of our knowledge, Zakrzewski [183] was the first to propose a method to verify simple, one hidden layer neural networks. However, only recently were researchers able to work with non-trivial models by taking advantage of the structure of ReLU-based networks [37, 94]. Even then, these works are not scalable to the large networks encountered in most real world problems.

This chapter advances the field of NN verification by making the following key contributions:

1. We show that state of the art verification methods Reluplex [94] and Planet [58] can be thought of as special cases of branch and bound optimization, which provides us with a **unified framework** to compare them.

2. We gather a data set of test cases based on the existing literature and extend it with new benchmarks. We **provide the first experimental comparison** of verification methods.

3. Based on this framework, we identify algorithmic improvements in the verification process, specifically in the way bounds are computed, the type of branching that are considered, as well as the strategies guiding the branching. Compared to the previous state of the art, these improvements lead to a **speed-up of almost two orders of magnitudes**.

Section 3.3 and 3.4 give the specification of the problem and formalise the verification process. Section 3.5 presents our unified framework, showing that previous methods are special cases and highlighting potential improvements. Section 3.6 presents our experimental setup and Section 3.7 analyses the results.

3.3 Problem specification

Throughout this chapter, lowercase letters such as $\hat{x}_n$ represent scalars, lowercase bold letters such as $\mathbf{x_i}$ will represent vectors, while uppercase letters such as W_i represent matrices. We use brackets to indicate when we refer to a specific coordinate of a vector, so $u_{i[k]}$ would correspond to the k-th coordinate of the vector $\mathbf{u_i}$.

We now specify the problem of formal verification of neural networks. Given a network that implements a function $f : \mathbb{R}^{h_0} \to \mathbb{R}^{h_n}$ such that $\hat{\mathbf{x}}_\mathbf{n} = f(\mathbf{x_0})$, a bounded input domain $\mathcal{C} \subset \mathbb{R}^{h_0}$ and a property $P : \mathbb{R}^{h_n} \to \mathbb{B}$, we want to prove

$$\mathbf{x_0} \in \mathcal{C}, \quad \hat{\mathbf{x}}_\mathbf{n} = f(\mathbf{x_0}) \implies P(\hat{\mathbf{x}}_\mathbf{n}). \tag{3.3.1}$$

$\mathbf{x_0}$ corresponds to the inputs to the neural network, and $\forall i \in \{1 \dots n\}$, $\hat{\mathbf{x}}_\mathbf{i}$ correspond to the activation of the network after the i-th linear layer. $h_i \in \mathbb{N}$ represents the size of the layers of the neural network.

As an example, let us consider a network trained to assign a label $y \in \{0 \dots K\}$ to each input. The property of robustness to adversarial examples in $\mathcal{L}_\infty$ norm around a training sample $\mathbf{a} \in \mathbb{R}^{h_0}$ with label y_a would be encoded by defining the restriction of the input domain $\mathcal{C} \triangleq \left\{\mathbf{x_0} \in \mathbb{R}^{h_0} \mid \|\mathbf{x_0} - \mathbf{a}\|_\infty \leq \epsilon\right\}$ and the boolean property $P(\hat{\mathbf{x}}_\mathbf{n}) = \bigwedge_{y \in \{0 \dots K\} \setminus \{y_a\}} \left[\hat{x}_{n[y_a]} > \hat{x}_{n[y]}\right]$.

In this chapter, we are going to focus on Piecewise-Linear Neural Networks (PL-NN), that is, networks for which we can decompose $\mathcal{C}$ into a finite set of polyhedra $\mathcal{C}_i$ such that $\mathcal{C} = \cup_i \mathcal{C}_i$, and the restriction of f to $\mathcal{C}_i$ is a linear function for each i. While this prevents us from including networks that use activation functions such as sigmoid or tanh, PL-NNs allow the use of linear transformations such as fully-connected or convolutional layers, pooling units such as MaxPooling and activation functions such as ReLUs. PL-NNs represent the majority of networks used in practice. Operations such as Batch-Normalization or Dropout also preserve piecewise linearity at test-time and thus fall under the family of networks considered.

The properties that we are going to consider are Boolean formulas over linear inequalities. In our robustness to adversarial example above, the property is a conjunction of linear inequalities, each of which constrains the output of the original label to be greater than the output of another label.

The scope of this chapter does not include approaches relying on additional assumptions such as the functions implemented by the network being twice differentiable [80, 183], limitation of the activation to binary values [37, 125] or restriction to a single linear domain [17]. Since they do not provide formal guarantees, we also don't include approximate approaches relying on a limited set of perturbation [85] or that are incomplete [173].

3.4 Verification Formalism

3.4.1 Verification as a Satisfiability problem

The methods we involve in our comparison all leverage the piecewise-linear structure of PL-NN to make the problem more tractable. They all follow the same general principle: given a property to prove, they attempt to discover a counterexample that would make the property false. This is accomplished by defining a set of variables corresponding to the inputs, hidden units and output of the network, and the set of constraints that a counterexample would satisfy. The verification problem amounts to solving the satisfiability problem of whether there exist an assignment satisfying these constraints.

To help design a unified framework, we reduce all instances of verification problems to a canonical representation. Specifically, the satisfiability problem will be transformed into a global optimization problem where the decision will be obtained by checking the sign of the minimum.

If the property to verify is a simple inequality $P(\hat{\mathbf{x}}_\mathbf{n}) \triangleq \mathbf{c}^T\hat{\mathbf{x}}_\mathbf{n} > b$, it is sufficient to add to the network a final fully connected layer with one output, with weight of $\mathbf{c}$ and a bias of $-b$. If the global minimum of this network is positive, it indicates that for all $\hat{\mathbf{x}}_\mathbf{n}$ the original network can output, we have $\mathbf{c}^T\hat{\mathbf{x}}_\mathbf{n} - b > 0 \implies \mathbf{c}^T\hat{\mathbf{x}}_\mathbf{n} > b$, and as a consequence the property is True. On the other hand, if the global minimum is negative, then the minimizer provides a counter-example. Negation can easily be achieved by changing the signs of the weights and biases of the inequality.

Clauses specified using `OR` (denoted by $\bigvee$) can be encoded by using a MaxPooling unit. If the property is $P(\hat{\mathbf{x}}_\mathbf{n}) \triangleq \bigvee_i \left[\mathbf{c}_i^T\hat{\mathbf{x}}_\mathbf{n} > b_i\right]$, it is equivalent to $\max_i \left(\mathbf{c}_i^T\hat{\mathbf{x}}_\mathbf{n} - b_i\right) > 0$. Clauses specified using `AND` (denoted by $\bigwedge$) can be encoded similarly: the property $P(\hat{x}_n) = \bigwedge_i \left[\mathbf{c}_i^T\hat{\mathbf{x}}_\mathbf{n} > b_i\right]$ is equivalent to:

$$\min_i \left(\mathbf{c}_i^T\hat{\mathbf{x}}_\mathbf{n} - b_i\right) > 0 \iff -\left(\max_i \left(-\mathbf{c}_i^T\hat{\mathbf{x}}_\mathbf{n} + b_i\right)\right) > 0$$

We can formulate any Boolean formula over linear inequalities on the output of the network as a sequence of additional linear and max-pooling layers. The verification problem will be reduced to the problem of finding whether the scalar output of the potentially modified network can reach a negative value. Assuming the network only contains ReLU activations between each layer, the satisfiability problem to find a counterexample can be expressed as:

$$\mathbf{l_0} \leq \mathbf{x_0} \leq \mathbf{u_0} \tag{3.4.1a}$$

$$\hat{\mathbf{x}}_\mathbf{i+1} = W_{i+1}\mathbf{x_i} + \mathbf{b_{i+1}} \qquad \forall i \in \{0,\ n-1\} \tag{3.4.1b}$$

$$\mathbf{x_i} = \max\left(\hat{\mathbf{x}}_\mathbf{i}, 0\right) \qquad \forall i \in \{1,\ n-1\} \tag{3.4.1c}$$

$$\hat{x}_n \leq 0. \tag{3.4.1d}$$

Eq. (3.4.1a) represents the constraints on the input, where the sample ($\mathbf{x_0} \in \mathbb{R}^{h_0}$) is contained between lower ($\mathbf{l_0} \in \mathbb{R}^{h_0}$) and upper ($\mathbf{u_0} \in \mathbb{R}^{h_0}$) bounds. Eq. (3.4.1d) is the constraint on the neural network output $\hat{x}_n \in \mathbb{R}$, that we can assume to be scalar thanks to the formalism above. Eq. (3.4.1b) encodes the linear layers of the network with weight matrix $W_i \in \mathbb{R}^{h_i \times h_{i-1}}$ and biases $\mathbf{b_i} \in \mathbb{R}^{h_i}$ and Eq. (3.4.1c) encodes the ReLU activation functions. If an assignment to all the values can be found, this represents a counterexample. If this problem is unsatisfiable, no counterexample can exist, implying that the property is True. We emphasise that we are required to prove that no counterexamples can exist, and not simply that none could be found.

While for clarity of explanation, we have limited ourselves to the specific case where only ReLU activation functions are used, this is not restrictive. Subsection 3.9.3 of the additional notes details how each method specifically handles MaxPooling units, as well as how to convert any MaxPooling operation into a combination of linear layers and ReLU activation functions.

The problem described in (3.4.1) is still a hard problem. The addition of the ReLU non-linearities (3.4.1c) transforms a problem that would have been solvable by simple Linear Programming into an NP-hard problem [94]. Converting a verification problem into this canonical representation does not make its resolution simpler but it does provide a formalism advantage. Specifically, it allows us to prove complex properties, containing several `OR` clauses, with a single procedure rather than having to decompose the desired property into separate queries, as was done in previous work [94].

Operationally, a valid strategy is to impose the constraints (3.4.1a) to (3.4.1c) and minimise the value of $\hat{x}_n$. Finding the exact global minimum is not necessary for verification. However, it provides a measure of satisfiability or unsatisfiability. If the value of the global minimum is positive, it will correspond to the margin by which the property is satisfied.

3.4.2 Mixed Integer Programming formulation

A possible way to eliminate the non-linearities is to encode them with the help of binary variables, transforming the PL-NN verification problem (3.4.1) into a Mixed Integer Linear Program (MIP).

By taking advantage of the feed-forward structure of the neural network, lower $\mathbf{l_i} \in \mathbb{R}^{h_i}$ and upper bounds $\mathbf{u_i}\mathbb{R}^{h_i}$ can be obtained by applying interval arithmetic [83]

to propagate the bounds on the inputs, one layer at a time.

$$l_{i+1[j]} = \sum_k \left(W^+_{i+1[j,k]} l^+_{i[k]} + W^-_{i+1[j,k]} u^+_{i[k]} \right) \tag{3.4.2a}$$

$$u_{i+1[j]} = \sum_k \left(W^+_{i+1[j,k]} u^+_{i[k]} + W^-_{i+1[j,k]} l^+_{i[k]} \right) \tag{3.4.2b}$$

with the notation $a^+ = \max(a, 0)$ and $a^- = \min(a, 0)$.

Using these bounds, it is possible to replace the non-linear constraints by linear ones by using the "big-M" formulation [74]. The following encoding was described in several papers Cheng et al. [36], Lomuscio and Maganti [110], and Tjeng et al. [157]:

$$\mathbf{x}_i = \max\left(\hat{\mathbf{x}}_\mathbf{i}, 0\right) \quad \Rightarrow \quad \boldsymbol{\delta_i} \in \{0, 1\}^{h_i}, \tag{3.4.3a}$$

$$\mathbf{x_i} \geq 0, \qquad \mathbf{x_i} \leq \mathbf{u_i} \cdot \boldsymbol{\delta_i} \tag{3.4.3b}$$

$$\mathbf{x_i} \geq \hat{\mathbf{x}}_\mathbf{i}, \qquad \mathbf{x_i} \leq \hat{\mathbf{x}}_\mathbf{i} - \mathbf{l_i} \cdot (1 - \boldsymbol{\delta_i}). \tag{3.4.3c}$$

It is easy to verify that $\delta_{i[j]} = 0$ is equivalent to $x_{i[j]} = 0$ (by replacing $\delta_{i[j]}$ in Eq. (3.4.3b)) and that $\delta_{i[j]} = 1$ is equivalent to $x_{i[j]} = \hat{x}_{i[j]}$ (by replacing $\delta_{i[j]}$ in Eq. (3.4.3c)).

Thanks to the specific feed-forward structure of the problem, the generic, non-linear, non-convex problem has been rewritten into an MIP. Optimization of MIP is well studied and highly efficient off-the-shelf solvers exist. As solving them is NP-hard, performance is going to be dependent on the quality of both the solver used and the encoding. We now ask the following question: how much efficiency can be gained by using a bespoke solver rather than a generic one? In order to answer this, we present specialised solvers for the PL-NN verification task.

3.5 Branch and Bound for Verification

As described in Section 3.4.1, the verification problem can be rephrased as a global optimization problem. Algorithms such as Stochastic Gradient Descent are not appropriate as they have no way of guaranteeing whether or not a minimum is global. In this section, we present an approach to estimate the global minimum, based on the branch and bound paradigm and show that several published methods, introduced as examples of Satisfiability Modulo Theories, fit this framework.

Algorithm 1 Branch and Bound

```
1:  function BAB(net, domain, ϵ)
2:      global_ub ← inf
3:      global_lb ← −inf
4:      doms ← [(global_lb, domain)]
5:      while global_ub − global_lb > ϵ do
6:          (_ , dom) ← pick_out(doms)
7:          [subdom_1, . . . , subdom_s] ← split(dom)
8:          for i = 1 . . . s do
9:              dom_ub ← compute_UB(net, subdom_i)
10:             dom_lb ← compute_LB(net, subdom_i)
11:             if dom_ub < global_ub then
12:                 global_ub ← dom_ub
13:                 prune_domains(doms, global_ub)
14:             end if
15:             if dom_lb < global_ub then
16:                 domains.append((dom_lb, subdom_i))
17:             end if
18:         end for
19:         global_lb ← min{lb | (lb, dom) ∈ doms}
20:     end while
21:     return global_ub
22: end function
```

Algorithm 1 describes its generic form. The input domain is repeatedly split into sub-domains (line 7), over which lower and upper bounds of the minimum are computed (lines 9-10). The best upper-bound found so far serves as a candidate for the global minimum. Any domain whose lower bound is greater than the current global upper bound can be pruned away as it cannot contain the global minimum (line 13, lines 15-17). By iteratively splitting the domains, it is possible to compute tighter lower bounds. We keep track of the global lower bound on the minimum by taking the minimum over the lower bounds of all sub-domains (line 19). When the global upper bound and the global lower bound differ by less than a small scalar ϵ (line 5), we consider that we have converged. Algorithm 1 shows how to optimise and obtain the global minimum. If all that we are interested in is the satisfiability problem, the procedure can be simplified by initialising the global upper bound with 0 (in line 2). Any subdomain with a lower bound greater than 0 (and therefore not eligible to contain a counterexample) will be pruned out (by line 15). The computation of the lower bound can therefore be replaced by the feasibility problem (or its relaxation) imposing the constraint that the output is below zero without changing the algorithm. If it is feasible, there might still be a counterexample and further branching is necessary. If it is infeasible, the subdomain can be pruned out. In addition,

if any upper bound improving on 0 is found on a subdomain (line 11), it is possible to stop the algorithm as this already indicates the presence of a counterexample.

The description of the verification problem as optimization and the pseudo-code of Algorithm 1 are generic and would apply to verification problems beyond the specific case of PL-NN. To obtain a practical algorithm, it is necessary to specify several elements.
A search strategy, defined by the `pick_out` function, which chooses the next domain to branch on. Several heuristics are possible, for example those based on the results of previous bound computations. For satisfiable problems or optimization problems, this allows to discover good upper bounds, enabling early pruning.
A branching rule, defined by the `split` function, which takes a domain `dom` and return a partition in subdomain. For a valid partition, the union of subdomain should correspond to the original domain ($\bigcup_i \texttt{subdom_i} = \texttt{dom}$) and all the subdomains should be disjoint ($\texttt{subdom_i} \cap \texttt{subdom_j}) = \emptyset,\ \forall i \neq j$. This will define the "shape" of the domains, which impacts the hardness of computing bounds. In addition, choosing the right partition can greatly impact the quality of the resulting bounds.
Bounding methods, defined by the `compute_`$\{$`UB, LB`$\}$ functions. These procedures estimate respectively upper bounds and lower bounds over the minimum output that the network `net` can reach over a given input domain. We want the lower bound to be as high as possible, so that this whole domain can be pruned easily. This is usually done by introducing convex relaxations of the problem and minimising them. On the other hand, the computed upper bound should be as small as possible, so as to allow pruning out other regions of the space or discovering counterexamples. As any feasible point corresponds to an upper bound on the minimum, heuristic methods are sufficient.

We now demonstrate how some published work in the literature can be understood as special case of the branch-and-bound framework for verification.

3.5.1 Reluplex

Katz et al. [94] present a procedure named Reluplex to verify properties of Neural Network containing linear functions and ReLU activation unit, functioning as an SMT solver using the splitting-on-demand framework [16]. The principle of Reluplex is to always maintain an assignment to all of the variables, even if some of the constraints are violated.

Starting from an initial assignment, it attempts to fix some violated constraints at each step. It prioritises fixing linear constraints ((3.4.1a), (3.4.1b), (3.4.1d) and some relaxation of (3.4.1c)) using a simplex algorithm, even if it leads to violated ReLU constraints. If no solution to this relaxed problem containing only linear constraints exists, the counterexample search is unsatisfiable. Otherwise, either all ReLU are respected, which generates a counterexample, or Reluplex attempts to fix one of the violated ReLU;

potentially leading to newly violated linear constraints. This process is not guaranteed to converge, so to guarantee progress, non-linearities constraints that are violated too many times during the search are split into two cases. Two new problems are generated, each corresponding to one of the phases of the ReLU. In the worst setting, the problem will be split completely over all possible combinations of activation patterns, at which point the sub-problems will all be simple LPs.

This algorithm can be mapped to the special case of branch-and-bound for satisfiability. The **search strategy** is handled by the SMT core and to the best of our knowledge does not prioritise any domain. The **branching rule** is implemented by the ReLU-splitting procedure: when neither the upper bound search, nor the detection of infeasibility are successful, one non-linear constraint over the j-th neuron of the i-th layer $x_{i[j]} = \max\left(\hat{x}_{i[j]}, 0\right)$ is split out into two subdomains: $\{x_{i[j]} = 0, \hat{x}_{i[j]} \leq 0\}$ and $\{x_{i[j]} = \hat{x}_{i[j]}, \hat{x}_{i[j]} \geq 0\}$. This defines the type of subdomains produced. The prioritisation of ReLUs that have been frequently fixed is a heuristic to decide between possible partitions.

As Reluplex only deal with satisfiability, the analogue of the lower bound computation is an over-approximation of the satisfiability problem. The **bounding method** used is a convex relaxation, obtained by dropping some of the constraints. The following relaxation is applied to ReLU units for which the sign of the input is unknown ($l_{i[j]} \leq 0$ and $u_{i[j]} \geq 0$).

$$\begin{aligned} \mathbf{x_i} = \max\left(\hat{\mathbf{x}}_\mathbf{i}, 0\right) \quad \Rightarrow \quad & \mathbf{x_i} \geq \hat{\mathbf{x}}_\mathbf{i} && \text{(3.5.1a)} \\ & \mathbf{x_i} \geq \mathbf{0} && \text{(3.5.1b)} \\ & \mathbf{x_i} \leq \mathbf{u_i}. && \text{(3.5.1c)} \end{aligned}$$

If this relaxation is unsatisfiable, this indicates that the subdomain cannot contain any counterexample and can be pruned out. The search for an assignment satisfying all the ReLU constraints by iteratively attempting to correct the violated ReLUs is a heuristic that is equivalent to the search for an upper bound lower than 0: success implies the end of the procedure but no guarantees can be given.

3.5.2 Planet

Ehlers [58] also proposed an approach based on SMT. Unlike Reluplex, the proposed tool, named Planet, operates by explicitly attempting to find an assignment to the state of the non-linearities, who can be either passing, when their input is positive or blocking, when their input is negative. Reusing the notation of Section 3.4.2, it assigns a value of 0

or 1 to each $\delta_{i[j]}$ variable, verifying at each step the feasibility of the partial assignment so as to prune infeasible partial assignments early.

As in Reluplex, the **search strategy** is not explicitly encoded and simply enumerates all the domains that have not yet been pruned. The **branching rule** is the same as for Reluplex, as fixing the decision variable $\delta_{i[j]} = 0$ is equivalent to choosing $\{x_{i[j]} = 0, \hat{x}_{i[j]} \leq 0\}$ and fixing $\delta_{i[j]} = 1$ is equivalent to $\{x_{i[j]} = \hat{x}_{i[j]}, \hat{x}_{i[j]} \geq 0\}$. Note however that Planet does not include any heuristic to prioritise which decision variables should be split.

Planet does not include a mechanism for early termination based on a heuristic search of a feasible point. For satisfiable problems, only when a full complete assignment is identified is a solution returned. In order to detect incoherent assignments, Ehlers [58] introduces a global linear approximation to a neural network, which is used as a **bounding method** to over-approximate the set of values that each hidden unit can take. In addition to the existing linear constraints ((3.4.1a), (3.4.1b) and (3.4.1d)), the non-linear constraints are approximated by sets of linear constraints representing the convex hull of the non-linearities. Specifically, ReLUs with input of unknown sign are replaced by the set of equations:

$$\mathbf{x_i} = \max(\hat{\mathbf{x}}_\mathbf{i}, 0) \Rightarrow \quad \mathbf{x_i} \geq \hat{\mathbf{x}}_\mathbf{i} \tag{3.5.2a}$$

$$\mathbf{x_i} \geq 0 \tag{3.5.2b}$$

$$x_{i[j]} \leq u_{i[j]} \frac{\hat{x}_{i[j]} - l_{i[j]}}{u_{i[j]} - l_{i[j]}} \qquad \forall j \in \mathbb{R}^{h_i} \tag{3.5.2c}$$

where $x_{i[j]}$ corresponds to the value of the j-th coordinate of $\mathbf{x_i}$.

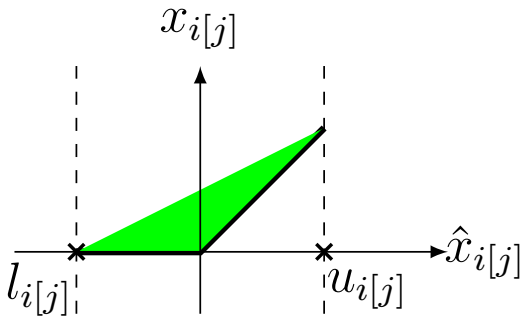


Figure 3.1: Feasible domain of the relaxation of a ReLU activation function

This feasible domain is illustrated in Figure 3.1.

Compared with the relaxation of Reluplex (3.5.1), the Planet relaxation is tighter. Specifically, Eq. (3.5.1a) and (3.5.1b) are identical to Eq. (3.5.2a) and (3.5.2b) but Eq. (3.5.2c) implies Eq. (3.5.1c). Indeed, given that $\hat{x}_{i[j]}$ is smaller than $u_{i[j]}$, the

fraction multiplying $u_{i[j]}$ is necessarily smaller than 1, implying that this provides a tighter bound on $x_{i[j]}$.

To use this approximation to compute better bounds than the ones given by simple interval arithmetic, it is possible to leverage the feed-forward structure of the neural network and obtain bounds one layer at a time. Having included all the constraints up until the i-th layer, it is possible to optimize over the resulting linear program and obtain bounds for all the units of the i-th layer, which in turn will allow us to create the constraints (3.5.2) for the next layer.

In addition to the pruning obtained by convex relaxation, both Planet and Reluplex make use of conflict analysis [118] to discover combinations of splits that cannot lead to satisfiable assignments, allowing them to perform further pruning of the domains.

3.5.3 Potential improvements

As can be seen, previous approaches to neural network verification have relied on methodologies developed in three communities: optimization, for the creation of upper and lower bounds; verification, especially SMT; and machine learning, especially the feed-forward nature of neural networks for the creation of relaxations. A natural question that arises is "Can other existing literature from these domains be exploited to further improve neural network verification?" Our unified branch-and-bound formulation makes it easy to answer this question. To illustrate its power, we now provide a non-exhaustive list of suggestions to speed-up verification algorithms.

Better bounding — While the relaxation proposed by Ehlers [58] is tighter than the one used by Reluplex, it can be improved further still. Specifically, after a splitting operation, on a smaller domain, we can refine all the $\mathbf{l_i}, \mathbf{u_i}$ bounds, to obtain a tither relaxation. We show the importance of this in the experiments section with the **BaB-relusplit** method that performs splitting on the activation like Planet but updates its approximation completely at each step.

One other possible area of improvement lies in the tightness of the bounds used. Equation (3.5.2) is very closely related to the Mixed Integer Formulation of Equation (3.4.3). Indeed, it corresponds to level 0 of the Sherali-Adams hierarchy of relaxations [147]. The proof for this statement can be found in subsection 3.9.2 of the additional notes. Stronger relaxations could be obtained by exploring higher levels of the hierarchy. This would jointly constrain groups of ReLUs, rather than linearising them independently.

Better branching — The decision to perform the branching on the activation of the ReLU non-linearities made by Planet and Reluplex is intuitive as it provides a clear set of decision variables to fix. However, it ignores another natural branching strategy, namely, splitting the input domain. Indeed, it could be argued that since the function encoded by the neural networks are piecewise linear in their input, this could result in the computation of highly useful upper and lower bounds. To demonstrate this, we propose the novel **BaB-input** algorithm: a branch-and-bound method that branches over the input features of the network. Based on a domain with input constrained by Eq. (3.4.1a), the `split` function would return two subdomains where constraints on the input would be identical in all dimension except for the dimension with the largest length, denoted $i^\star \in \mathbb{R}^{h_0}$. The bounds on the input for each subdomain for dimension $i^\star$ are given by $l_{0[i^\star]} \leq x_{0[i^\star]} \leq \frac{l_{0[i^\star]}+u_{0[i^\star]}}{2}$ and $\frac{l_{0[i^\star]}+u_{0[i^\star]}}{2} \leq x_{0[i^\star]} \leq u_{0[i^\star]}$. Based on these tighter input bounds, tighter bounds at all layers can be re-evaluated.

One of the main advantage of branching over the variables is that all subdomains generated by the **BaB-input** algorithm when splitting over the input variables end up only having simple bound constraints over the value that input variables can take. In order to exploit this property to the fullest, we use the highly efficient lower bound computation approach of Wong and Kolter [171]. This approach was initially proposed in the context of robust optimization. However, our unified framework opens the door for its use in verification. Specifically, Wong and Kolter [171] identified an efficient way of computing bounds for the type of problems we encounter, by generating a feasible solution to the dual of the LP generated by the Planet relaxation. While these bounds are quite loose compared to the one obtained through actual optimization, they are very fast to evaluate. We propose a smart branching method **BaBSB** to replace the longest edge heuristic of **BaB-input**. For all possible splits, we compute fast bounds for each of the resulting subdomain, and execute the split resulting in the highest lower bound. The intuition is that despite their looseness, the fast bounds will still be useful in identifying the promising splits.

Another advantage of the branch-and-bound approach is that it's not dependent on the networks being piecewise linear. While methods such as **Reluplex**, **Planet** or the MIP encoding depends on the piecewise linearity, any type of networks for which an appropriate bounding function can be found will be verifiable using branch-and-bound. Recent advances on incomplete verification such as the work of Dvijotham et al. [53] can offer such bounds for activations such as sigmoid or hyperbolic tangent.

3.6 Experimental setup

The problem of PL-NN verification has been shown to be NP-complete [94]. Meaningful comparison between approaches therefore needs to be experimental.

3.6.1 Methods

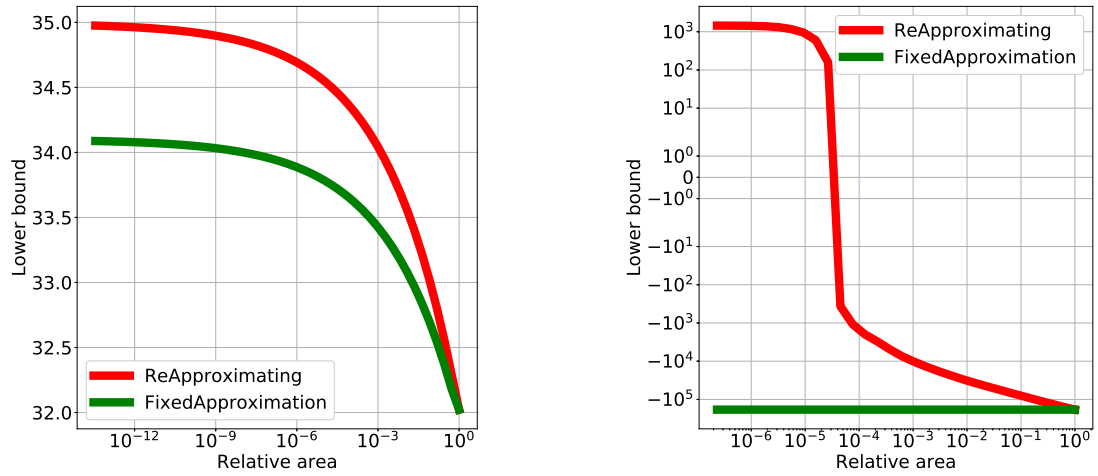
The simplest baseline we refer to is **BlackBox**, a direct encoding of Eq. (3.4.1) into the Gurobi solver, which will perform its own relaxation, without taking advantage of the problem's structure.

For the SMT based methods, **Reluplex** and **Planet**, we use the publicly available versions [59, 93]. Both tools are implemented in C++ and rely on the GLPK library to solve their relaxation. We wrote some software to convert in both directions between the input format of both solvers.

We also evaluate the potential of using MIP solvers, based on the formulation of Eq. (3.4.3). Due to the lack of availability of open-sourced methods at the time of our experiments, we reimplemented the approach in Python, using the Gurobi MIP solver. We report results for a variant called **MIPplanet**, which uses bounds derived from Planet's convex relaxation rather than simple interval arithmetic. Both the MIP and **BlackBox** are not treated as simple feasibility problem but are encoded to minimize the output $\hat{x}_n$ of Equation (3.4.1d), with a callback interrupting the optimization as soon as a negative value is found. Additional discussions on encodings of the MIP problem can be found in subsection 3.9.4 of the additional notes.

In our benchmark, we include the methods derived from our branch and bound analysis. Our implementation follows faithfully Algorithm 1, is implemented in Python and uses Gurobi to solve LPs. The `pick_out` strategy consists in prioritising the domain that currently has the smallest lower bound. Upper bounds are generated by randomly sampling points on the considered domain, and we use the convex approximation of Ehlers [58] to obtain lower bounds. As opposed to the approach taken by Ehlers [58] of building a single approximation of the network, we rebuild the approximation and recompute all bounds for each sub-domain. This is motivated by the observation shown in Figure 3.2 which demonstrate the significant improvements it brings, especially for deeper networks. For `split`, **BaB-input** performs branching by splitting the input domain in half along its longest edge and **BaBSB** does it by splitting the input domain along the dimension improving the global lower bound the most according to the fast bounds of Wong and Kolter [171]. We also include results for the **BaB-relusplit** variant, where the `split` method is based on the phase of the non-linearities, similarly to **Planet**.

All methods were benchmarked using the same machine, restricted to operate in single threaded mode to not penalize the implementation not supporting parallelism. The difference in performance between C++ and python might affect the timings on a small scale, leading to an advantage for **Reluplex** and **Planet** but the difference created should be negligible due to the mixed integer programming and branch and bound solver spending most of their time inside Gurobi who is also implemented in C++.



(a) CollisionDetection network **(b)** ACAS network

Figure 3.2: *Quality of the linear approximation, depending on the size of the input domain. We plot the value of the lower bound as a function of the area on which it is computed (higher is better). The domains are centered around the global minimum and repeatedly shrunk. Rebuilding completely the linear approximation at each step allows to create tighter lower-bounds thanks to better* $\mathbf{l_i}$ *and* $\mathbf{u_i}$*, as opposed to using the same constraints and only changing the bounds on input variables. This effect is even more significant on deeper networks. Note that the first graph has a linear scale while the secod one has a logarithmic scale.*

3.6.2 Evaluation Criteria

For each of the data sets, we compare the different methods using the same protocol. We attempt to verify each property with a timeout of two hours, and a maximum allowed memory usage of 20GB, on a single core of a machine with an i7-5930K CPU. We measure the time taken by the solvers to either prove or disprove the property. If the property is false and the search problem is therefore satisfiable, we expect from the solver to exhibit a counterexample. If the returned input is not a valid counterexample, we don't count the property as successfully proven, even if the property is indeed satisfiable. All code and data necessary to replicate our analysis are open sourced.

3.7 Analysis

We attempt to perform verification on three data sets of properties and report the comparison results. The dimensions of all the problems can be found in Table 3.1.

The **CollisionDetection** data set [58] attempts to predict whether two vehicles with parameterized trajectories are going to collide. 500 properties are extracted from problems arising from a binary search to identify the size of the region around training examples in which the prediction of the network does not change. The network used is relatively

Data set	Count	Model Architecture
Collision Detection	500	6 inputs 40 hidden unit layer, MaxPool 19 hidden unit layer 2 outputs
ACAS	188	5 inputs 6 layers of 50 hidden units 5 outputs
PCAMNIST	27	10 or {5, 10, 25, 100, 500, 784} inputs 4 or {2, 3, 4, 5, 6, 7} layers of 25 or {10, 15, 25, 50, 100} hidden units, 1 output, with a margin of +1000 or {-1e4, -1000, -100, -50, -10, -1 ,1, 10, 50, 100, 1000, 1e4}

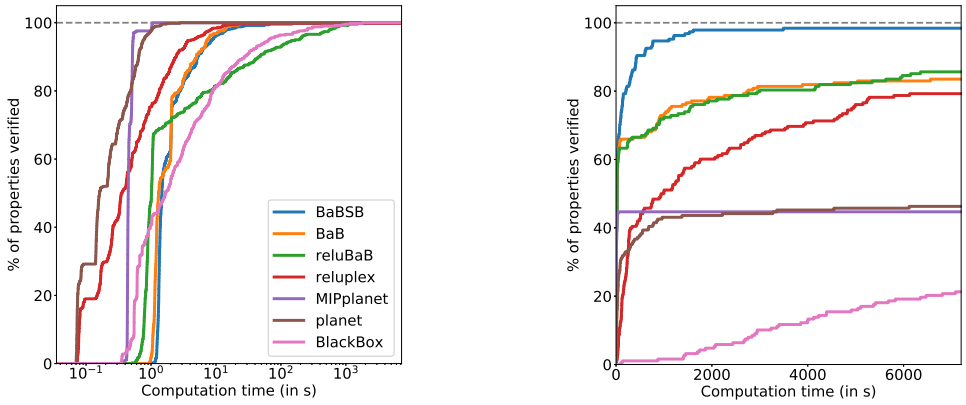
Table 3.1: *Dimensions of all the data sets. For PCAMNIST, we use a base network with 10 inputs, 4 layers of 25 hidden units and a margin of 1000. We generate new problems by changing one parameter at a time, using the values inside the brackets.*

shallow but due to the process used to generate the properties, some lie extremely close between the decision boundary between SAT and UNSAT. Results presented in Figure 3.3 therefore highlight the accuracy of methods.

The **Airborne Collision Avoidance System (ACAS)** data set, as released by Katz et al. [94] is a neural network based advisory system recommending horizontal manoeuvres for an aircraft in order to avoid collisions, based on sensor measurements. Each of the five possible manoeuvres is assigned a score by the neural network and the action with the minimum score is chosen. The 188 properties to verify are based on some specification describing various scenarios. Due to the deeper network involved, this data set is useful in highlighting the scalability of the various algorithms.

Existing data sets do not allow us to explore the impact on the performance of different methods of various problem/model parameters such as depth, number of hidden units, and input dimensionality. Our new data set, **PCAMNIST**, removes this deficiency, and can prove helpful in analysing future verification approaches as well. It is generated in a way to give control over different architecture parameters. Details of the dataset construction can be found in Section 3.9.6. We present plots in Figure 3.5 showing the evolution of runtimes depending on each of the architectural parameters of the networks.

Comparative evaluation of verification approaches — In Figure 3.3a, on the shallow networks of **CollisionDetection**, most solvers succeed against all properties in about 10s. In particular, the SMT inspired solvers **Planet**, **Reluplex** and the MIP solver are extremely fast.



(a) *CollisionDetection Dataset* **(b)** *ACAS Dataset*

Figure 3.3: *Proportion of properties verifiable for varying time budgets depending on the methods employed. A higher curve means that for the same time budget, more properties will be solvable. All methods solve CollisionDetection quite quickly except **reluBaB**, which is much slower and **BlackBox** who produces several incorrect counterexamples.*

On the deeper networks of **ACAS**, in Figure 3.3b, no errors are observed but most methods timeout on the most challenging testcases. The best baseline is **Reluplex**, which reaches 79.26% success rate at the two hour timeout, while our best method, **BaBSB**, already achieves 98.40% with a budget of one hour. To reach Reluplex's success rate, the required runtime is two orders of magnitude smaller.

Impact of each improvement — To identify which changes allow our method to have good performance, we do an ablation study and study the impact of removing some components of our methods. The only difference between **BaBSB** and **BaB** is the smart branching, which represents a significant part of the performance gap.

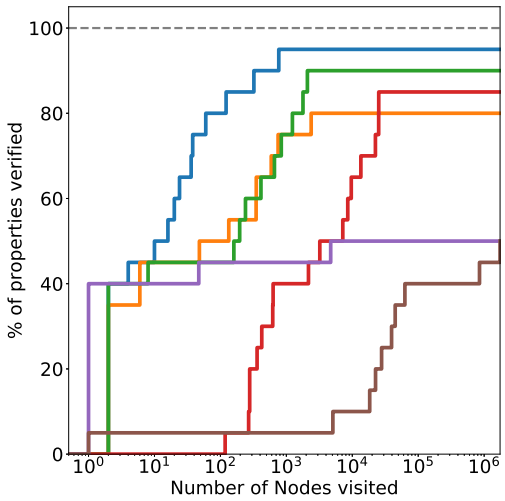
Branching over the inputs rather than over the activations does not contribute much, as shown by the small difference between **BaB** and **reluBaB**. Note however that we are able to use the fast methods of Wong and Kolter [171] for the smart branching because branching over the inputs makes the bounding problem similar to the one solved in robust optimization. Even if it doesn't improve performance by itself, the new type of split enables the use of smart branching.

The rest of the performance gap can be attributed to using a better bounds: **reluBaB** significantly outperforms **Planet** while using the same branching strategy and the same convex relaxations. The improvement comes from the benefits of rebuilding the approximation at each step shown in Figure 3.2.

Figure 3.4 presents some additional analysis on a 20-property subset of the ACAS dataset, showing how the methods used impact the need for branching. Smart branching and the use of better lower bounds reduce heavily the number of subdomains to explore.

Method	Average time per Node
BaBSB	1.81s
BaB	2.11s
reluBaB	1.69s
reluplex	0.30s
MIPplanet	0.017s
planet	1.5e-3s

Table 3.2: *Average time to explore a node for each method.*



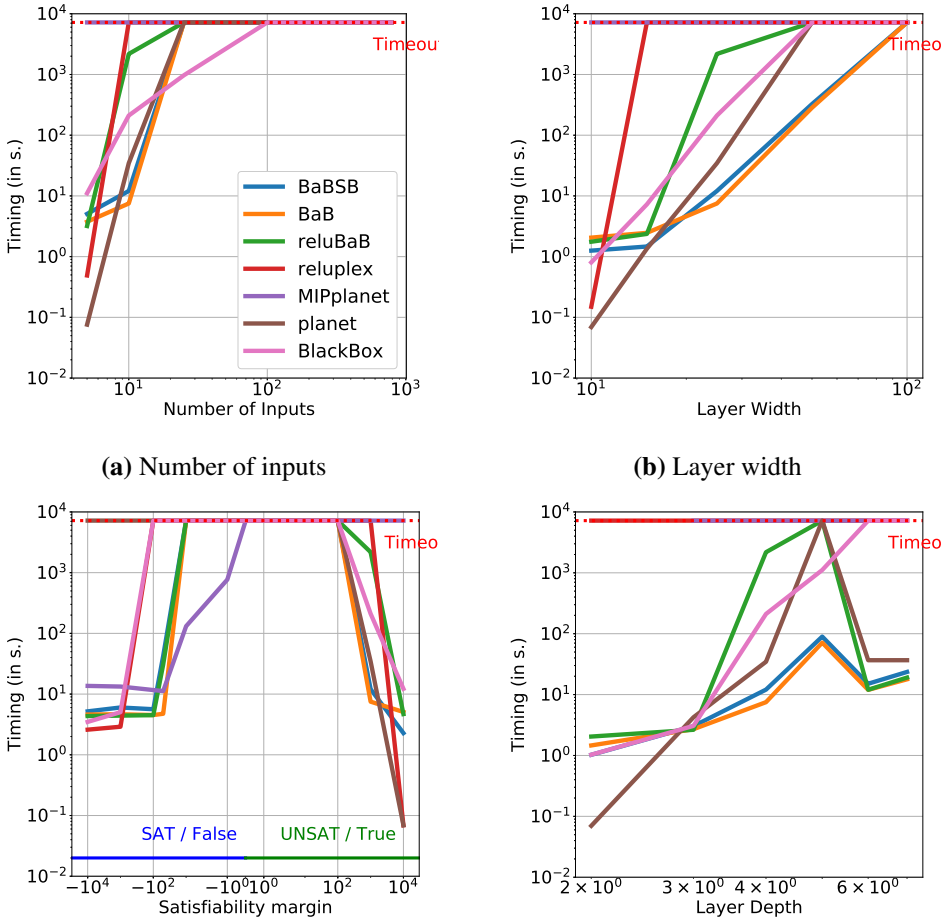
(a) *Properties solved for a given number of nodes to explore (log scale).*

Figure 3.4: *The trade-off taken by the various methods are different. Figure 3.4a shows how many subdomains needs to be explored before verifying properties while Table 3.2 shows the average time cost of exploring each subdomain. Our methods have a higher cost per node but they require significantly less branching, thanks to better bounding. Note also that between **BaBSB** and **BaB**, the smart branching reduces by an order of magnitude the number of nodes to visit.*

In the graphs of Figure 3.5, the trend for all the methods are similar, which seems to indicate that hard properties are intrinsically hard and not just hard for a specific solver. Figure 3.5a shows an expected trend: the larger the number of inputs, the harder the problem is. Similarly, Figure 3.5b shows unsurprisingly that wider networks require more time to solve, which can be explained by the fact that they have more non-linearities. The impact of the margin, as shown in Figure 3.5c is also clear. Properties that are True or False with large satisfiability margin are easy to prove, while properties that have small satisfiability margins are significantly harder.

3.8 Conclusion

The improvement of formal verification of Neural Networks represents an important challenge to be tackled before learned models can be used in safety-critical applications. By providing both a unified framework to reason about methods and a set of empirical benchmarks to measure performance with, we hope to contribute to progress in this direction. Our analysis of published algorithms through the lens of branch and bound optimization has already resulted in significant improvements in runtime on our benchmarks. Its continued analysis should reveal even more efficient algorithms in the future.



(a) Number of inputs

(b) Layer width

(c) Margin

(d) Network depth

Figure 3.5: *Impact of the various parameters over the runtimes of the different solvers. The base network has 10 inputs and 4 layers of 25 hidden units, and the property to prove is True with a margin of 1000. Each of the plot corresponds to a variation of one of this parameters.*

3.9 Additional Notes

3.9.1 Toy Problem example

We have specified the problem of formal verification of neural networks as follows: given a network that implements a function $\hat{\mathbf{x}}_\mathbf{n} = f(\mathbf{x_0})$, a bounded input domain $\mathcal{C}$ and a property P, we want to prove that

$$\mathbf{x_0} \in \mathcal{C}, \quad \hat{\mathbf{x}}_\mathbf{n} = f(\mathbf{x_0}) \implies P(\hat{\mathbf{x}}_\mathbf{n}). \tag{3.9.1}$$

A toy-example of the Neural Network verification problem is given in Figure 3.6. On the domain $\mathcal{C} = [-2; 2] \times [-2; 2]$, we want to prove that the output y of the one hidden-layer network always satisfy the property $P(y) \triangleq [y > -5]$. We will use this as a running example to explain the methods used for comparison in our experiments.

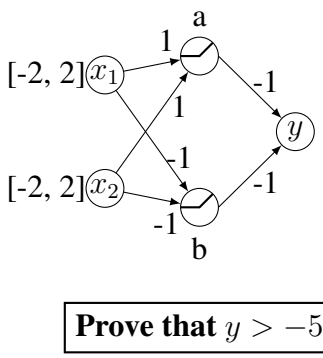


Figure 3.6: Example Neural Network. We attempt to prove the property that the network output is always greater than -5

Problem formulation

For the network of Figure 3.6, the variables would be $\{x_1, x_2, a_{\text{in}}, a_{\text{out}}, b_{\text{in}}, b_{\text{out}}, y\}$ and the set of constraints would be:

$$-2 \leq x_1 \leq 2 \qquad -2 \leq x_2 \leq 2 \tag{3.9.2a}$$

$$\hat{a} = x_1 + x_2 \qquad \hat{b} = -x_1 - x_2$$

$$y = -a - b \tag{3.9.2b}$$

$$a = \max(\hat{a}, 0) \qquad b = \max(\hat{b}, 0) \tag{3.9.2c}$$

$$y \leq -5 \tag{3.9.2d}$$

Here, $\hat{a}$ is the input value to hidden unit, while a is the value after the ReLU. Any point satisfying all the above constraints would be a counterexample to the property, as it would imply that it is possible to drive the output to -5 or less.

MIP formulation

In our example, the non-linearities of equation (3.9.2c) are replaced by

$$\begin{aligned} a \geq 0 \qquad & a \geq \hat{a} \\ a \leq \hat{a} - l_a(1 - \delta_a) \qquad & a \leq u_a \delta_a \\ \delta_a \in \{0, 1\} & \end{aligned} \tag{3.9.3}$$

where l_a is a lower bound of the value that $\hat{a}$ can take (such as -4) and u_a is an upper bound (such as 4). The binary variable δ_a indicates which phase the ReLU is in: if $\delta_a = 0$, the ReLU is blocked and $a_{\text{out}} = 0$, else the ReLU is passing and $a_{\text{out}} = a_{\text{in}}$. The problem remains difficult due to the integrality constraint on δ_a.

Step	x_1	x_2	$\hat{a}$	a	$\hat{b}$	b	y
1	0	0	0	0	0	0	0
	Fix linear constraints						
	0	0	0	1	0	4	-5
2	0	0	0	1	0	4	-5
	Fix a ReLU						
	0	0	0	1	4	4	-5
3	0	0	0	1	4	4	-5
	Fix linear constraints						
	-2	-2	-4	1	4	4	-5
	...						

Figure 3.7: *Evolution of the Reluplex algorithm. Red cells corresponds to value violating Linear constraints, and orange cells corresponds to value violating ReLU constraints. Resolution of violation of linear constraints are prioritized.*

Running Reluplex

Table 3.7 shows the initial steps of a run of the Reluplex algorithm on the example of Figure 3.6. Starting from an initial assignment, it attempts to fix some violated constraints at each step. It prioritizes fixing linear constraints using a simplex algorithm ((3.9.2a), (3.9.2b) and (3.9.2d) in our illustrative example) , even if it leads to violated ReLU constraints (3.9.2c). This can be seen in step 1 and 3 of the process.

If no solution to the problem containing only linear constraints exists, this shows that the counterexample search is unsatisfiable. Otherwise, all linear constraints are fixed and Reluplex attempts to fix one violated ReLU at a time, such as in step 2 of Table 3.7 (fixing the ReLU b), potentially leading to newly violated linear constraints. In the case where no violated ReLU exists, this means that a satisfiable assignment has been found and that the search can be interrupted.

This process is not guaranteed to converge, so to guarantee progress, non-linearities getting fixed too often are split into two cases. This generates two new sub-problems, each involving an additional linear constraint instead of the linear one. The first one solves the problem where $\hat{a} \leq 0$ and $a = 0$, the second one where $\hat{a} \geq 0$ and $a = \hat{a}$. In the worst setting, the problem will be split completely over all possible combinations of activation patterns, at which point the sub-problems are simple LPs.

3.9.2 Link between the MIP formulation and the Planet relaxation

The feasible set of the Mixed Integer Programming formulation is given by the following set of equations. We assume that all $\mathbf{l_i}$ are negative and $\mathbf{u_i}$ are positive. If this is

not true, it is possible to update the bounds without modifying the tightness of the relaxation by using different bounds:

$$\begin{aligned} \mathbf{l_i} &\leftarrow \min(\mathbf{l_i}, 0) \\ \mathbf{u_i} &\leftarrow \max(\mathbf{u_i}, 0) \end{aligned} \tag{3.9.4}$$

$$\mathbf{l_0} \leq \mathbf{x_0} \leq \mathbf{u_0} \tag{3.9.5a}$$
$$\hat{\mathbf{x}}_{\mathbf{i+1}} = W_{i+i}\mathbf{x_i} + \mathbf{b_{i+i}} \qquad \forall i \in \{0,\ n-1\} \tag{3.9.5b}$$
$$\mathbf{x_i} \geq 0 \qquad \forall i \in \{1,\ n-1\} \tag{3.9.5c}$$
$$\mathbf{x_i} \geq \hat{\mathbf{x}}_\mathbf{i} \qquad \forall i \in \{1,\ n-1\} \tag{3.9.5d}$$
$$\mathbf{x_i} \leq \hat{\mathbf{x}}_\mathbf{i} - \mathbf{l_i} \cdot (1 - \boldsymbol{\delta}_i) \qquad \forall i \in \{1,\ n-1\} \tag{3.9.5e}$$
$$\mathbf{x_i} \leq \mathbf{u_i} \cdot \boldsymbol{\delta}_i \qquad \forall i \in \{1,\ n-1\} \tag{3.9.5f}$$
$$\boldsymbol{\delta}_i \in \{0,1\}^{h_i} \qquad \forall i \in \{1,\ n-1\} \tag{3.9.5g}$$
$$\hat{x}_n \leq 0 \tag{3.9.5h}$$

The level 0 of the Sherali-Adams hierarchy of relaxation Sherali and Adams [147] doesn't include any additional constraints. Indeed, polynomials of degree 0 are simply constants and their multiplication with existing constraints followed by linearization therefore doesn't add any new constraints. As a result, the feasible domain given by the level 0 of the relaxation corresponds simply to the removal of the integrality constraints:

$$\mathbf{l_0} \leq \mathbf{x_0} \leq \mathbf{u_0} \tag{3.9.6a}$$
$$\hat{\mathbf{x}}_{\mathbf{i+1}} = W_{i+i}\mathbf{x_i} + \mathbf{b_{i+i}} \qquad \forall i \in \{0,\ n-1\} \tag{3.9.6b}$$
$$\mathbf{x_i} \geq 0 \qquad \forall i \in \{1,\ n-1\} \tag{3.9.6c}$$
$$\mathbf{x_i} \geq \hat{\mathbf{x}}_\mathbf{i} \qquad \forall i \in \{1,\ n-1\} \tag{3.9.6d}$$
$$\mathbf{x_i} \leq \hat{\mathbf{x}}_\mathbf{i} - \mathbf{l_i} \cdot (1 - \boldsymbol{d}_i) \qquad \forall i \in \{1,\ n-1\} \tag{3.9.6e}$$
$$\mathbf{x_i} \leq \mathbf{u_i} \cdot \boldsymbol{d}_i \qquad \forall i \in \{1,\ n-1\} \tag{3.9.6f}$$
$$\underline{0 \leq \boldsymbol{d}_i \leq 1} \qquad \forall i \in \{1,\ n-1\} \tag{3.9.6g}$$
$$\hat{x}_n \leq 0 \tag{3.9.6h}$$

Combining the equations (3.9.6e) and (3.9.6f), looking at a single unit j in layer i, we obtain:

$$x_{i[j]} \leq \min\left(\hat{x}_{i[j]} - l_i(1 - d_{i[j]}), u_{i[j]}d_{i[j]}\right) \tag{3.9.7}$$

The function mapping $d_{i[j]}$ to an upper bound of $x_{i[j]}$ is a minimum of linear functions, which means that it is a concave function. As one of them is increasing and the other is decreasing, the maximum will be reached when they are both equals.

$$\begin{aligned} \hat{x}_{i[j]} - l_{i[j]}(1 - d^\star_{i[j]}) &= u_{i[j]} d^\star_{i[j]} \\ \Leftrightarrow \qquad d^\star_{i[j]} &= \frac{\hat{x}_{i[j]} - l_{i[j]}}{u_{i[j]} - l_{i[j]}} \end{aligned} \tag{3.9.8}$$

Plugging this equation for $d^\star$ into Equation(3.9.7) gives that:

$$x_{i[j]} \leq u_{i[j]} \frac{\hat{x}_{i[j]} - l_{i[j]}}{u_{i[j]} - l_{i[j]}} \tag{3.9.9}$$

which corresponds to the upper bound of $x_{i[j]}$ introduced for Planet [58].

3.9.3 MaxPooling

We now present how to handle MaxPooling activations, either by converting them to the already handled case of ReLU activations or by introducing an explicit encoding of them when appropriate.

Mixed Integer Programming

Similarly to the encoding of ReLU constraints using binary variables and bounds on the inputs, it is possible to similarly encode MaxPooling constraints. The constraint

$$y = \max(x_1, \ldots, x_k) \tag{3.9.10}$$

can be replaced by

$$y \geq x_i \qquad \forall i \in \{1 \ldots k\} \tag{3.9.11a}$$

$$y \leq x_i + (u_{x_{1:k}} - l_{x_i})(1 - \delta_i) \qquad \forall i \in \{1 \ldots k\} \tag{3.9.11b}$$

$$\sum_{i \in \{1 \ldots k\}} \delta_i = 1 \tag{3.9.11c}$$

$$\delta_i \in \{0, 1\} \qquad \forall i \in \{1 \ldots k\} \tag{3.9.11d}$$

where $u_{x_{1:k}}$ is an upper-bound on all x_i for $i \in \{1 \ldots k\}$ and l_{x_i} is a lower bound on x_i.

Reluplex

In the version introduced by [94], there is no support for MaxPooling units. As the canonical representation we evaluate needs them, we provide a way of encoding a MaxPooling unit as a combination of Linear function and ReLUs.

To do so, we decompose the element-wise maximum into a series of pairwise maximum

$$\begin{aligned}\max\left(x_j, x_2, x_3, x_4\right) = \max(\ &\max\left(x_1, x_2\right), \\ &\max\left(x_3, x_4\right))\end{aligned} \tag{3.9.12}$$

and decompose the pairwise maximums as sum of ReLUs:

$$\max\left(x_1, x_2\right) = \max\left(x_1 - x_2,\ 0\right) + \max\left(x_2 - l_{x_2}, 0\right) + l_{x_2}, \tag{3.9.13}$$

where l_{x_2} is a pre-computed lower bound of the value that x_2 can take.

As a result, we have seen that an elementwise maximum such as a MaxPooling unit can be decomposed as a series of pairwise maximum, which can themselves be decomposed into a sum of ReLUs units. The only requirement is to be able to compute a lower bound on the input to the ReLU, for which the methods discussed in Section 3.5.2 can help.

Planet

As opposed to Reluplex, Planet Ehlers [58] directly supports MaxPooling units. The SMT solver driving the search can split on ReLUs, by considering separately the case of the ReLU being passing or blocking. It also has the possibility on splitting on MaxPooling units, by treating separately each possible choice of units being the largest one.

For the computation of lower bounds, the constraint

$$y = \max\left(x_1, x_2, x_3, x_4\right) \tag{3.9.14}$$

is replaced by the set of constraints:

$$y \geq \mathrm{x}_i \qquad \forall i \in \{1 \ldots 4\} \tag{3.9.15a}$$

$$y \leq \sum_i \left(x_i - l_{x_i}\right) + \max_i l_{x_i}, \tag{3.9.15b}$$

where x_i are the inputs to the MaxPooling unit and l_{x_i} their lower bounds.

3.9.4 Mixed Integer Variants

While mixed integer programming solvers are very flexible tools, it is not straightforward to predict their performance. Different formulation of the same problem will result in the same solution as the solvers are complete, but the runtime might be completely different. In this subsection, we describe several design choices that needed to be made and provide experimental evidence justifying the strategy employed by our mixed integer programming baseline in Section 3.6.

Encoding

Several variants of encoding are available to use Mixed Integer Programming as a solver for Neural Network Verification. As a reminder, the formulation used in the main chapter was the encoding of Tjeng et al. [157]:

$$x_i = \max(\hat{\mathbf{x}}_\mathbf{i}, 0) \quad \Rightarrow \quad \boldsymbol{\delta_i} \in \{0,1\}^{h_i}, \tag{3.9.16a}$$

$$\mathbf{x_i} \geq 0, \qquad \mathbf{x_i} \leq \mathbf{u_i} \cdot \boldsymbol{\delta_i} \tag{3.9.16b}$$

$$\mathbf{x_i} \geq \hat{\mathbf{x}}_\mathbf{i}, \qquad \mathbf{x_i} \leq \hat{\mathbf{x}}_\mathbf{i} - \mathbf{l_i} \cdot (1 - \boldsymbol{\delta_i}) \tag{3.9.16c}$$

An alternative formulation is the one of Lomuscio and Maganti [110] and Cheng et al. [36]:

$$x_i = \max(\hat{\mathbf{x}}_\mathbf{i}, 0) \quad \Rightarrow \quad \boldsymbol{\delta_i} \in \{0,1\}^{h_i}, \tag{3.9.17a}$$

$$\mathbf{x_i} \geq 0, \qquad \mathbf{x_i} \leq \mathbf{M_i} \cdot \boldsymbol{\delta_i} \tag{3.9.17b}$$

$$\mathbf{x_i} \geq \hat{\mathbf{x}}_\mathbf{i}, \qquad \mathbf{x_i} \leq \hat{\mathbf{x}}_\mathbf{i} + \mathbf{M_i} \cdot (1 - \boldsymbol{\delta_i}) \tag{3.9.17c}$$

where $\mathbf{M_i} = \max(-\mathbf{l_i}, \mathbf{u_i})$. This is fundamentally the same encoding but with a slightly worse bounds that is used, as one of the side of the bounds isn't as tight as it could be. The bounds used for the lower and upper bound are the same, so we will refer to this encoding as *symmetric* and to the other one as *assymetric*. Intuitively, models encoded with the assymetric encoding (3.9.16) should perform better that ones encoded with the symmetric encoding (3.9.17) due to using tighter bounds.

Obtaining bounds

Both formulation described in Equations (3.9.16) and (3.9.17) are dependent on obtaining lower and upper bounds for the values of the activations of the network.

One way to obtain them is the use of interval arithmetic [83]. If we have bounds $\mathbf{l_i}, \mathbf{u_i}$ for a vector $\mathbf{x_i}$, we can derive the bounds $\hat{\mathbf{l}}_\mathbf{i+1}, \hat{\mathbf{u}}_\mathbf{i+1}$ for a vector $\hat{\mathbf{x}}_\mathbf{i+1} = W_{i+1}\mathbf{x_i} + b_{i+1}$

$$\hat{l}_{i+1[j]} = \sum_k \left(W^+_{i+1[j,k]} l^+_{i[k]} + W^-_{i+1[j,k]} u^+_{i[k]} \right) + b_{i+1[j]} \tag{3.9.18a}$$

$$\hat{u}_{i+1[j]} = \sum_k \left(W^+_{i+1[j,k]} u^+_{i[k]} + W^-_{i+1[j,k]} l^+_{i[k]} \right) + b_{i+1[j]} \tag{3.9.18b}$$

with the notation $a^+ = \max(a, 0)$ and $a^- = \min(a, 0)$. Propagating the bounds through a ReLU activation is simply equivalent to applying the ReLU to the bounds.

An alternative way to obtain bounds is to use the relaxation of Planet. This is the method used for the experiments of section 3.6: we build incrementally the network

approximation, layer by layer. To obtain the bounds over an activation, we optimize its value subject to the constraints of the relaxation. Given that this is a convex problem, we will achieve the optimum. Given that it is a relaxation, the optimum will be a valid bound for the activation (given that the feasible domain of the relaxation includes the feasible domains subject to the original constraints). Once this value is obtained, we can use it to build the relaxation for the following layers. We can build the linear approximation for the whole network and extract the bounds for each activation to use in the encoding of the MIP.

In this situation, it is not straightforward to predict which alternative is going to perform better. Bounds derived from interval arithmetic are looser, making only poor relaxation accessible to the MIP solver, but they are much faster to compute. On the other hand, bounds based on the linear programming relaxation of Ehlers [58] are tighter but significantly more expensive to compute. The first approach will make the definition of the MIP fast but its solving slow, while the second will make the MIP definition a slow process, but in exchange of potentially faster solving.

Satisfiability versus Optimization

In section 3.4.1, we have formalised the verification problem as a satisfiability problem, but also shown that the same problem could be rephrased as an optimization problem. Mixed integer programming solvers are capable of functioning with both types of specification: either as a satisfiability problem for the existence of a counterexample, or as an equivalent optimization problem. Both approaches are theoretically equivalent, but in practice may perform significantly differently.

Comparison

We ran experiments with different strategies to encode the verification problem into the mixed integer programming solver, corresponding to different choices for the design decisions highlighted above. Experiments were ran on the CollisionDetection and ACAS dataset.

The different variants that we tested were:

1. **Planet-feasible** uses the assymetric encoding of Equation (3.9.16). The intermediate bounds used to formulate the non-linear constraints were based on the Planet relaxation, and the problem was solved as a satisfiability problem.

2. **Planet-symfeasible** uses the symetric encoding of Equation (3.9.17). The intermediate bounds used to formulate the non-linear constraints were based on the Planet relaxation and the problem was solved as a satisfiability problem.

3. **Interval** uses the assymetric encoding of Equation (3.9.17). The intermediate bounds used to formulate the non-linear constraints were based on the interval analysis method (3.9.18) and the problem was solved as a satisfiability problem.

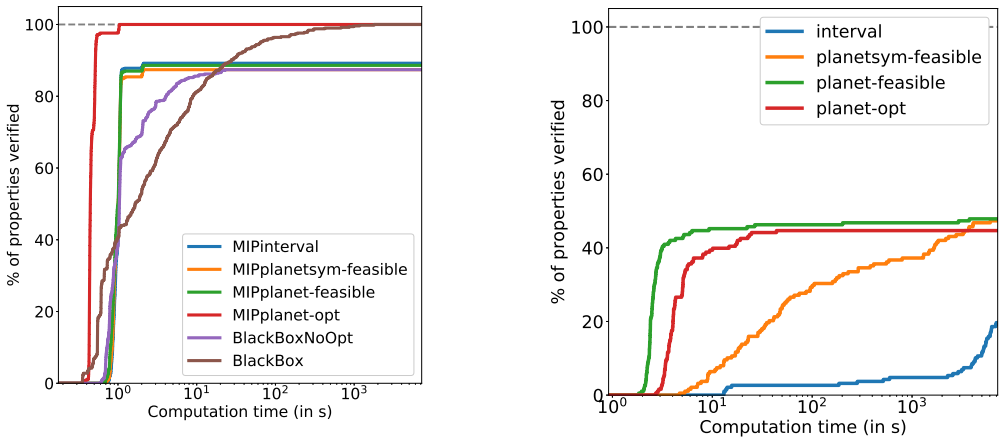
4. **Planet-opt** uses the assymetric encoding of Equation (3.9.16). The intermediate bounds used to formulate the non-linear constraints were based on the planet relaxation, and the problem was solved as an optimization problem. We configure the mixed integer programming solver to find the global minimum of the output of the network modified as explained in Section 3.4.1. This procedure makes it similar to our proposed branch and bound method, except that rather than using our implementation of branch and bound for optimization, we use the Gurobi solver. By using the callback features of Gurobi, we implement early stopping of the method whenever a counter-example is exhibited, even if the global minimum has not been achieved. This corresponds to the version whose results are reported in section 3.7 under the name of **MIPPlanet**.

The comparison between **Planet-feasible** and **Planet-symfeasible** should reveal which of the encoding (3.9.16) or (3.9.17) leads to better performance. The comparison between **Planet-symfeasible** and **Interval** should reveal what is the most judicious type of bounds to use, at least for the problems considered here. Comparing the results of **Planet-opt** and **Planet-feasible** will show the benefits of formulating the mixed integer programs as optimization problem rather than as satisfiability problems.

The comparison also include two variants of **BlackBox**: **BlackBox** and **BlackBoxNoOpt**. Similarly to **Planet-opt**, **BlackBox** attempts to do global optimization and interrupt the search when a feasible solution with negative value is found. **BlackBoxNoOpt** works like the other MIP encoding by simply encoding the problem as satisfiability.

The first observation that can be made is that when we look at the CollisionDetection dataset in Figure 3.8a, only **Planet-opt** and **BlackBox** solves the dataset to 100% accuracy. The reason why the other methods don't reach it is not because of timeout but because they return spurious counterexamples. As they encode only satisfiability problem, they terminate as soon as they identify a solution with a value of zero. Due to the large constants involved in the big-M and floating point errors, those solutions are sometimes not actually valid counterexamples. This is a significant advantage to encoding the problem as optimization problems as opposed to a simple satisfiability problems.

The other results that we can observe is the impact of the quality of the bounds when the networks get deeper, and the problem becomes therefore more complex, such as in the ACAS dataset. **Interval** has the worst bounds and is much slower than the other methods. **Planetsym-feasible**, with its slightly worse bounds, performs worse than **Planet-feasible** and **Planet-opt**.

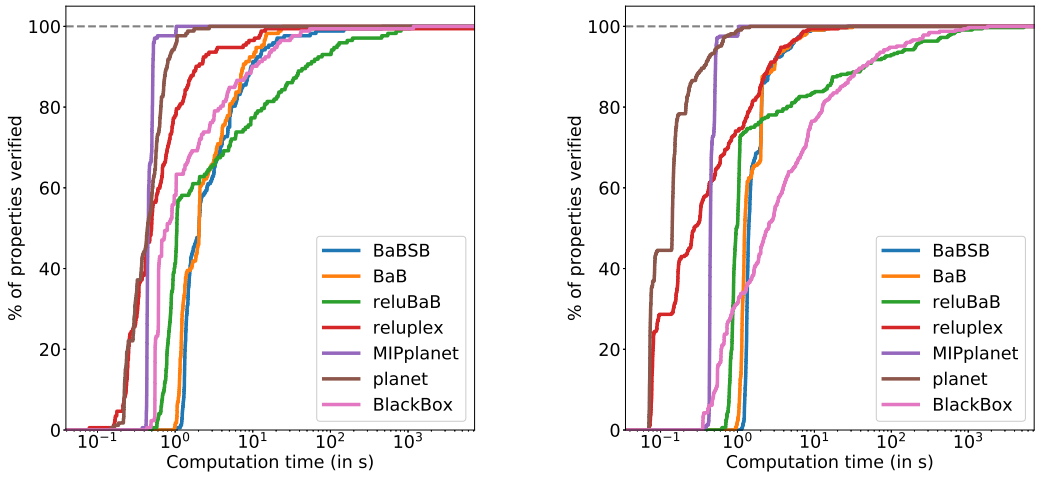


(a) CollisionDetection Dataset **(b)** ACAS Dataset

Figure 3.8: *Comparison between the different variants of MIP formulation for Neural Network verification.*

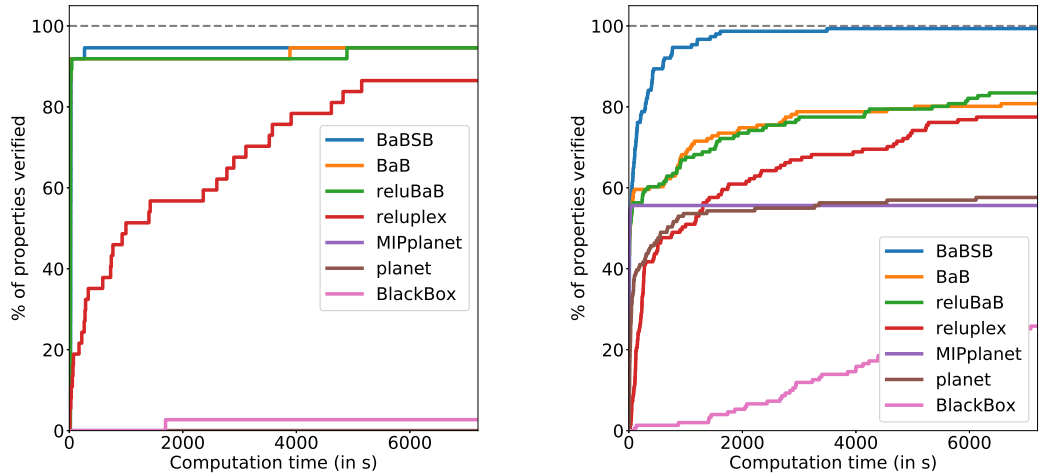
3.9.5 Additional performance details

Given that there is a significant difference in the way verification works for SAT problems vs. UNSAT problems, we report also comparison results on the subset of data sorted by decision type.



(a) *On SAT properties* **(b)** *On UNSAT properties*

Figure 3.9: *Proportion of properties verifiable for varying time budgets depending on the methods employed on the CollisionDetection dataset. We can identify that all the errors that* ***BlackBox*** *makes are on SAT properties, as it returns incorrect counterexamples.*



(a) *On SAT properties* **(b)** *On UNSAT properties*

Figure 3.10: *Proportion of properties verifiable for varying time budgets depending on the methods employed on the ACAS dataset. We observe that* ***planet*** *doesn't succeed in solving any of the SAT properties, while our proposed methods are extremely efficient at it, even if there remains some properties that they can't solve.*

3.9.6 PCAMNIST details

PCAMNIST is a novel data set that we introduce to get a better understanding of what factors influence the performance of various methods. It is generated in a way to give control over different architecture parameters. The networks takes k features as input, corresponding to the first k eigenvectors of a Principal Component Analysis decomposition of the digits from the MNIST data set. We also vary the depth (number of layers), width (number of hidden unit in each layer) of the networks. We train a different network for each combination of parameters on the task of predicting the parity of the presented digit. This results in the accuracies reported in Table 3.3.

The properties that we attempt to verify are whether there exists an input for which the score assigned to the odd class is greater than the score of the even class plus a large confidence. By tweaking the value of the confidence in the properties, we can make the property either True or False, and we can choose by how much is it true. This gives us the possibility of tweaking the "margin", which represent a good measure of difficulty of a network.

In addition to the impact of each factors separately, we can also look at it as a generic dataset and plot the cactus plots like for the other datasets. This can be found in Figure 3.11.

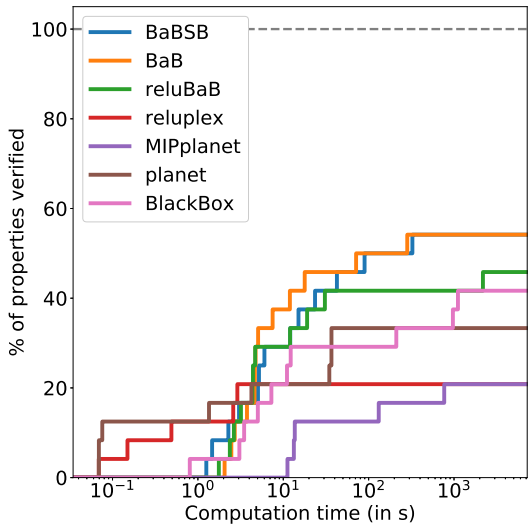


Figure 3.11: *Proportion of properties verifiable for varying time budgets depending on the methods employed. The PCAMNIST dataset is challenging as none of the methods reaches more than 50% success rate.*

Network Parameter			Accuracy	
Nb inputs	**Width**	**Depth**	**Train**	**Test**
5	25	4	88.18%	87.3%
10	25	4	97.42%	96.09%
25	25	4	99.87%	98.69%
100	25	4	100%	98.77%
500	25	4	100%	98.84%
784	25	4	100%	98.64%
10	10	4	96.34%	95.75%
10	15	4	96.31%	95.81%
10	25	4	97.42%	96.09%
10	50	4	97.35%	96.0%
10	100	4	97.72%	95.75%
10	25	2	96.45%	95.71%
10	25	3	96.98%	96.05%
10	25	4	97.42%	96.09%
10	25	5	96.78%	95.9%
10	25	6	95.48%	95.2%
10	25	7	96.81%	96.07%

Table 3.3: Accuracies of the network trained for the PCAMNIST dataset.

4

Knowing When to Stop: Evaluation and Verification of Conformity to Output-size Specifications

4.1 Preamble

This chapter contains a paper published at CVPR 2019 [163]. This was joint work with Chenglong Wang, who was in charge of developping the adversarial attacks, while I had the responsability of developping the verification methods.

Models such as *Sequence-to-Sequence* and *Image-to-Sequence* are widely used in real-world applications. While the ability of these neural architectures to produce variable-length outputs makes them extremely effective for problems like Machine Translation and Image Captioning, it also leaves them vulnerable to failures of the form where the model produces outputs of undesirable length. This behaviour can have severe consequences such as usage of increased computation and induce faults in downstream modules that expect outputs of a certain length. Motivated by the need to have a better understanding of the failures of these models, this chapter proposes and studies a novel output-size modulation problem and makes two key technical contributions. First, to evaluate model robustness, we develop an easy-to-compute differentiable proxy objective that can be used with gradient-based algorithms to find inputs that leads to abnormally long outputs. Second and more importantly, we develop a verification approach to formally prove that the network cannot produce outputs greater than a certain length. Our experimental results on Machine Translation and Image Captioning show that our output-lengthening approach can produce outputs that are 50 times longer than what would be expected,

while our verification approach can, given a model and input domain, prove that the output length is below a certain size.

4.2 Introduction

Neural networks with variable output lengths have become ubiquitous in several applications. In particular, recurrent neural networks (RNNs) such as LSTMs [84], used to form "sequence" models [154], have been successfully and extensively applied in image captioning [7, 35, 61, 92, 112, 138, 145, 162, 177], video captioning [161, 164, 177, 180, 181], machine translation (MT) [40, 154], summarization [41], and in other sequence-based transduction tasks.

The ability of these sequence neural models to generate variable-length outputs is key to their performance on complex prediction tasks. However, this ability lead to a vulnerability to adversaries that can try to force the model to produce outputs of specific lengths that, for instance, lead to increased computation or affect the correct operation of down-stream modules. To address this issue, we introduce the *output-length modulation* problem where given a *specification* of the form that the model should produce outputs with less than a certain maximum length, we want to find adversarial examples, *i.e.* search for inputs that lead the model to produce outputs with a larger length and thus show that the model under consideration violates the specification. Different from existing work on targeted or untargeted attacks where the goal is to perturb the input such that the output is another class or sequence in the development dataset (thus within the dataset distribution), the output-modulation problem requires solving a more challenging task of finding inputs such that the output sequences are outside of the training distribution, which was previously claimed difficult [32].

The naive approach to the solution of the output-length modulation problem involves a computationally intractable search over a large discrete search space. To overcome this, we develop an easy-to-compute differentiable proxy objective that can be used with gradient-based algorithms to find output-lengthening inputs. Our experimental results on Machine Translation show that our adversarial output-lengthening approach can produce outputs that are 50 times longer than the input. However, when evaluated on the image-to-text image captioning model, the method is less successful.

As the procedure of searching for adversarial examples based on a proxy and using a gradient based method is not guaranteed to find any if they exist, there are two potential reasons for this result: the image-to-text architecture might be truly robust, or the adversarial approach is not powerful enough to find adversarial examples for this model. To resolve this question, we develop a verification method for checking and formally

proving whether a network is consistent with the output-size specification for the given range of inputs. To the best of our knowledge, our verification algorithm is the first formal verification approach to check properties of recurrent models with variable output lengths.

Our Contributions To summarize, the key contributions of this chapter are as follows:

- We propose and formulate the novel output-size modulation problem to study the behaviour of neural architectures capable of producing variable length outputs, and we study its evaluation and verification problems.
- For evaluation, we design an efficiently computable differentiable proxy for the expected length of the output sequence. Experiments show that this proxy can be optimized using gradient descent to efficiently find inputs causing the model to produce long outputs.
- Empirical demonstration that popular machine translation models can be forced to produce long outputs that are 50 times longer than the input sequence. The long output sequences help expose modes that the model can get stuck in, such as undesirable loops where they continue to emit a specific token for several steps.
- We demonstrate the feasibility of formal verification of recurrent models by proposing the use of mixed-integer programming to formally verify that a certain neural image-captioning model will be consistent with the specification for the given range of inputs.

Motivations and Implications Our focus on studying the output-length modulation problem is motivated by the following key considerations:

- *Achieving Computational Robustness:* Many ML models are now offered as a service to customers via the cloud. In this context, ML services employing variable-output models could be vulnerable to denial-of-service attacks that cause the ML model to perform wasteful computations by feeding it inputs that induce long outputs. This is particularly relevant for *variable compute models*, like Seq2Seq [40, 154]. Given a trained instance of the model, no method is known to check for the consistency of the model with a specification on the number of computation steps. Understanding the vulnerabilities of ML models to such output-lengthening and computation-increasing attacks is important for the safe deployment of ML services.

- *Understanding and Debugging Models:* By designing inputs that cause models to produce long outputs, it is possible to reason about the internal representations learned by the model and isolate where the model exhibits undesirable behavior. For example, we find that an English to German sequence-to-sequence model can produce outputs that end with a long string of question marks ('?'). This indicates that when the output decoder state is conditioned on a sequence of '?'s, it can end up stuck in the same state.
- *Uncovering security vulnerabilities through adversarial stress-testing:* The adversarial approach to output-length modulation tries to find parts of the space of inputs where the model exhibits improper behavior. Such inputs do not only reveal abnormal output size, but could also uncover other abnormalities like privacy violations of the kind that were recently revealed by [30] where an LSTM was forced to output memorized data.
- *Canonical specification for testing generalization of variable-output models:* Norm-bounded perturbations of images [155] have become the standard specification to test attacks and defenses on image classifiers. While the practical relevance of this particular specification can be questioned [66], it is still served as a useful canonical model encapsulating the essential difficulty in developing robust image classifiers. We believe stability of output-lengths can serve a similar purpose: as a canonical specification for variable output-length models. The main difficulties in studying variable output length models in an adversarial sense (the non-differentiability of the objective with respect to inputs) are exposed in output-lengthening attack, making it a fertile testing ground for both evaluating attack methods and defenses. We hope that advances made here will facilitate the study of robustness on *variable compute models* and other specifications for variable-output models, such as monotonicity.

4.3 Related Work

There are several recent studies on generating adversarial perturbations on variable-output models. [88, 139] show that question answering and machine comprehension models are sensitive to attacks based on semantics preserving modification or the introduction of unrelated information. [57, 179] find that character-level classifiers are highly sensitive to small character manipulations. [146] shows that models predicting the correctness of image captions struggle against perturbations consisting of a single word change. [32] and [39] further study adversarial attacks for sequential-output models (machine-translation, image captioning) with specific target captions or keywords.

We focus on sequence output models, and analyze the output-length modulation problem, where the models should produce outputs with at least a certain number of output tokens. We study whether a model can be adversarially perturbed to change the size of the output, which is a more challenging task compared to targeted attacks (see details in Section 4.4). On the one hand, existing targeted attack tasks aim to perturb the input such that the output is another sequence in the validation dataset (thus within the training distribution), but attacking output size requires the model to generate out-of-distribution long sequences. On the other hand, since the desired output sequence is only loosely constrained by the length rather than directly provided by the user, the attack algorithm is required to explore the output size to make the attack possible.

For models that cannot be adversarially perturbed, we develop a verification approach to show that it isn't simply a lack of power by the adversary but the sign of true robustness from the model. Our method encodes the verification problem in the form of a mixed integer program and use an off-the-shelf solver to resolve it. Similar approaches have been investigated for feedforward networks [25, 36, 110, 157] or recurrent network with a fixed sequence length [95] but our work is the first to handle variable output length models and the corresponding decoding mechanisms.

Most closely related to our approach is the work on neural agent-environment systems. Initial work by Maganti [117] attempted to tackle formal verification of properties expressed in the Linear Temporal Logic language [133] over a multiple step process encoding the evolution of a system constituted by both a model and an environment. This did not describe a complete algorithm, instead proposing a method based on over-approximating the set of states reachable by the process. This was improved upon by Akintunde et al. [3] who showed that, assuming that the environment could also be modeled as a mixed integer program, it was possible to perform complete verification of the joint system using a mixed integer programming formulation. Follow-up work [4] extended the procedure to additionally support the use of recurrent neural networks. The method presented in this chapter can be seen as a special case of verification of a neural agent-environment system where the greedy decoding process plays the role of the environment, modifying the state on which the network is conditioned on based on the action taken.

4.4 Modulating Output-Size

We study neural network models capable of producing outputs of variable length. We start with a canonical abstraction of such models, and later specialize to concrete models used in machine translation and image captioning.

We denote by x the input to the network and by $\mathcal{X}$ the space of all inputs to the network. Depending on the application domain, x might be an image represented by a vector of pixel intensities ($\mathcal{X} \subset \mathbb{R}^{h \times w}$) or a sequence of tokens ($\mathcal{X} \subset V^n$ with $V = \{1 \dots D_{\text{in}}\}$ being the input vocabulary and n the length of the sequence). We consider a set of inputs of interest $\mathcal{S}$, which can denote, for example, the set of "small"[1] perturbations of a nominal input. We study models that produce variable-length outputs sequentially. Let $y_t \in \mathcal{Y}$ denote the t-th decoded output of the model, where $\mathcal{Y} = \{1 \dots D_{\text{out}}\}$ is the output vocabulary of the model. At each time step t, the output of the network is a vector $\mathbf{o}_t \in \mathbb{R}^{D_{\text{out}}}$. $\mathbf{o}_t$ defines a probability over the next element $P(y_t|x, y_{0:t-1})$ in the decoded sequence. There exists a special end-of-sequence element in the output vocabulary $\mathbf{eos} \in \mathcal{Y}$ that signals termination of the output sequence.

In practice, different models adopt different decoding strategies for generating y_t from the probability $P(y_t|x, y_{0:t-1})$ [65, 87, 98]. In this chapter, we focus on the commonly used deterministic *greedy decoding* strategy [65], where at each time step, the generated token is given by taking the mode of the probability distribution over tokens produced by the network.

$$y_0 = \operatorname{argmax} \{P(\cdot|x)\} \tag{4.4.1a}$$

$$y_{t+1} = \operatorname{argmax} \{P(\cdot|x, y_{0:t})\} \text{ if } y_t \neq \mathbf{eos}. \tag{4.4.1b}$$

Since greedy decoding is deterministic, for a given sample x with a finite length output, we can define the length of the greedily decoded sequence as:

$$L(x) = t \text{ s.t } \begin{cases} y_t = \mathbf{eos} \\ y_i \neq \mathbf{eos} \quad \forall i < t \\ y_{i+1} = \operatorname{argmax} \{P(.|x, y_{0:i})\} \quad \forall i < t. \end{cases} \tag{4.4.2}$$

Note that there is a unique t that satisfies the above conditions, which is precisely the first t at which $y_t = \mathbf{eos}$ when using greedy decoding.

Output length modulation specification A network is said to satisfy the *output length modulation specification* parameterized by $\mathcal{S}, \hat{K}$, if for all inputs x in $\mathcal{S}$, the model terminates within $\hat{K}$ steps under greedy decoding for all $x \in \mathcal{S}$, formally:

$$\forall x \in \mathcal{S} \qquad L(x) \leq \hat{K}. \tag{4.4.3}$$

In Section 4.4.1, we study the problem of finding adversarial examples, i.e., searching for inputs that lead the model to produce outputs with a larger length and thus show that the model violates the specification. In Section 4.5, we use a formal verification method to prove that a model will be consistent with the specification for the given range of inputs, if such attacks are indeed impossible.

[1] The precise definition of small is specific to the application studied.

4.4.1 The Output-Size Modulation Problem

In order to check whether the specification, Eq. (4.4.3), is valid, one can consider a falsification approach that tries to find counterexamples proving that Eq. (4.4.3) is invalid. If an exhaustive search over $\mathcal{S}$ for such counterexamples fails, the specification is indeed true. However, exhaustive search is computationally intractable - hence in this section we develop gradient-based algorithms that can efficiently find counterexamples (although they may miss them even if they exist). To develop the falsification approach, we study the solution to the following optimization objective:

$$\max_{x \in \mathcal{S}} L(x), \tag{4.4.4}$$

where $\mathcal{S}$ is the valid perturbation space. If the optimal solution x in the space $\mathcal{S}$ has $L(x) > \hat{K}$, then (4.4.3) is invalid.

The attack spaces $\mathcal{S}$ we consider in this chapter include both continuous inputs (for image-to-text models) and discrete inputs (for Seq2Seq models).

Continuous inputs: For continuous inputs, such as image captioning tasks, the input is an $n \times m$ image with pixel values normalized to be in the range $[-1, 1]$. x is an $n \times m$ matrix of real numbers and $\mathcal{X} = [-1, 1]^{n \times m}$. We define the perturbation space $\mathcal{S}(x, \delta)$ as follows:

$$\mathcal{S}(x, \delta) = \{x' \in \mathcal{X} \mid \|x' - x\|_\infty \leq \delta\}, \tag{4.4.5}$$

i.e., the space of δ perturbations of the input x in the L_∞ ball.

Discrete inputs: For discrete inputs, e.g. machine translation tasks, inputs are discrete tokens in a language vocabulary. Formally, given the vocabulary V of the input language, the input space $\mathcal{X}$ is defined as all finite sentences composed of tokens in V, i.e., $\mathcal{X} = \{(x_1, \ldots, x_n) \mid x_i \in V, n > 0\}$. Given an input sequence $x = (x_1, \ldots, x_n)$, we define the δ-perturbation space of a sequence as all sequences of length n with at most $\lceil \delta \cdot n \rceil$ tokens different from x (i.e., $\delta \in [0, 1]$ denotes the percentage of tokens that an attacker is allowed to modify). Formally, the perturbation space $\mathcal{S}(x, \delta)$ is defined as follows:

$$\mathcal{S}(x, \delta) = \{(x'_1, \ldots, x'_n) \in V^n \mid \sum \|x_i - x'_i\|_0 \leq \lceil \delta \cdot n \rceil\} \tag{4.4.6}$$

4.4.2 Extending Projected Gradient Descent Attacks

In the projected gradient descent (PGD) attacks [116],[2] given an objective function $J(x)$, the attacker calculates the adversarial example by searching for inputs in the attack space to maximize $J(x)$. In the basic attack algorithm, we perform the following updates at each iteration:

$$x' = \Pi_{\mathcal{S}(x,\delta)} \left(x + \alpha \nabla_x J(x) \right) \tag{4.4.7}$$

where $\alpha > 0$ is the step size and $\Pi_{\mathcal{S}(x,\delta)}$ denotes the projection of the attack to the valid space $\mathcal{S}(x, \delta)$. Observe that the adversarial objective in Eq. (4.4.4) cannot be directly used as $J(x)$ to update x as the length of the sequence is not a differentiable objective function. This hinders the direct application of PGD to output-lengthening attacks. Furthermore, when the input space $\mathcal{S}$ is discrete, gradient descent cannot be directly be used because it is only applicable to continuous input spaces.

In the following, we show our extensions of the PGD attack algorithm to handle these challenges.

Greedy approach for sequence lengthening We introduce a differentiable proxy of $L(x)$. Given an input x whose decoder output logits are $(o_1, \ldots, o_k)$ (i.e., the decoded sequence is $y = (\mathrm{argmax}(o_1), \ldots, \mathrm{argmax}(o_k))$), instead of directly maximizing the output sequence length, we use a greedy algorithm to find an output sequence whose length is longer than k by minimizing the probability of the model to terminate within k steps. In other words, we minimize the log probability of the model to produce **eos** at any of the timesteps between 1 to k. Formally, the proxy objective $\tilde{J}$ is defined as follows:

$$\tilde{J}(x) = \sum_{t=1}^{k} \max \left\{ o_t[\mathbf{eos}] - \max_{z \neq \mathbf{eos}} o_t[z],\ -\epsilon \right\} \tag{4.4.8}$$

where $\epsilon > 0$ is a hyperparameter to clip the loss. This is piecewise differentiable wrt the inputs x (in the same sense that the ReLU function is differentiable) and can be efficiently optimized using PGD.

4.4.3 Continuous relaxation for discrete inputs

While we can apply the PGD attack with the proxy objective on the model with continuous inputs by setting the projection function $\Pi_{\mathcal{S}(x,\delta)}$ as the Euclidean projection, we cannot directly update discrete inputs. To enable a PGD-type attack in the discrete input space,

[2]Here the adversarial objective is stated as maximization, so the algorithm is Projected Gradient *Ascent*, but we stick with the PGD terminology since it is standard in the literature

we use the Gumbel trick [86] to reparameterize the input space to perform continuous relaxation of the inputs.

Given an input sequence $x = (x_1, \ldots, x_n)$, for each x_i, we construct a distribution $\pi_i \in \mathbb{R}^{|V|}$ initialized with $\pi_i[x_i] = 1$ and $\pi_i[z] = -1$ for all $z \in V \setminus \{x_i\}$. By passing this unnormalized distribution through the softmax function

$$p_i[z] = \text{softmax}(\pi_i)[z] = \frac{\exp \pi_i[z]}{\sum_{z' \in V} \exp \pi_i[z']}, \tag{4.4.9}$$

we obtain a probability distribution over input tokens at position i with a mode at x_i.

With this reparameterization, instead of feeding $x = (x_1, \ldots, x_n)$ into the model, we feed the Gumbel softmax sampling from the distribution $(u_1, \ldots, u_n)$. The sample $\tilde{x} = (\tilde{x}_1, \ldots, \tilde{x}_n)$ is calculated as follows:

$$u_i \sim \text{Uniform}(0, 1)^{|V|} \tag{4.4.10a}$$

$$g_i = -\log(-\log(u_i)) \tag{4.4.10b}$$

$$\tilde{x}_i = \text{softmax}(\frac{g_i + \log p_i}{\tau}) \tag{4.4.10c}$$

where τ is the Gumbel-softmax sampling temperature that controls the discreteness of $\tilde{x}$. If the temperature is close to zero, $\tilde{x}_i$ will become close to a one-hot encoding, but a high temperature will represent interpolation between several options, allowing differentiability. With this relaxation, we perform PGD attack on the distributions π_i at each iteration. Since π_i is unconstrained, the projection step in (4.4.7) is unnecessary.

When the final $\pi' = (\pi'_1, \ldots, \pi'_n)$ is obtained from the PGD attack, we draw samples $x'_i \sim \text{Categorical}(\pi_i)$ to get the final adversarial example for the attack.

Note that this method provides no guarantee of finding adversarial examples even if they exist. As a result, running it and claiming that no adversarial examples exist when it does not succeed would result in a unsound decision procedure. It is nevertheless useful because in certain cases, it will succeed in proving that the model studied are not robust by exhibiting counter examples.

4.5 Verified Bound on Output Length

While heuristics approaches can be useful in finding attacks, they can fail due to the difficulty of optimizing nondifferentiable nonconvex functions. These challenges show up particularly when the perturbation space is small or when the target model is trained with a strong bias in the training data towards short output sequences (e.g., the Show-and-Tell model, as we will show in Section 4.7). Thus, we design a formal verification approach for complete reasoning of the output-size modulation problem, i.e., finding

provable guarantees that no input within a certain set of interest can result in an output sequence of length above a certain threshold.

Our approach relies on counterexample search using intelligent brute-force search methods, taking advantage of powerful modern integer programming solvers [68]. We encode all the constraints that an adversarial example should satisfy as linear constraints, possibly introducing additional binary variables. Once in the right formalism, these can be fed in to an off-the-shelf Mixed Integer Programming (MIP) solver, which provably solves the problem, albeit with a potentially large computational cost. The constraints consist of four parts: (1) the initial restrictions on the model inputs (encoding $\mathcal{S}(x, \delta)$), (2) the relations between the different activations of the network (implementing each layer), (3) the decoding strategy (connection between the output logits and the inputs at the next step), and (4) the condition for it being a counterexample (*ie*. a sequence of length larger than the threshold). In the following, we show how each part of the constraints is encoded into MIP formulas.

Our formulation is inspired by pior work on encoding feed-forward neural networks as MIPs [25, 36, 157]. The image captioning model we use consists of an image embedding model, a feedforward convolutional neural network that computes an embedding of the image, followed by a recurrent network that generates tokens sequentially starting with the initial hidden state set to the image embedding.

The image embedding model is simply a sequence of linear or convolutional layers and ReLU activation functions. Linear and convolutional layers are trivially encoded as linear equality constraints between their inputs and outputs, while ReLUs are represented by introducing a binary variable and employing the big-M method [74]:

$$x_i = \max(\hat{x}_i, 0) \Rightarrow \delta_i \in \{0, 1\}, \quad x_i \geq 0 \tag{4.5.1a}$$

$$x_i \leq u_i \cdot \delta_i, \quad x_i \geq \hat{x}_i \tag{4.5.1b}$$

$$x_i \leq \hat{x}_i - l_i \cdot (1 - \delta_i), \tag{4.5.1c}$$

with l_i and u_i being lower and upper bounds of $\hat{x_i}$ which can be obtained using interval arithmetic (details in [25]).

Our novel contribution is to introduce a method to extend the techniques to handle greedy decoding used in recurrent networks. For a model with greedy decoding, the token with the most likely prediction is fed back as input to the next time step. To implement this mechanism as a mixed integer program, we employ a big-M method [170]:

$$o_{\text{max}} = \max_{y \in \mathcal{Y}}(o_y) \Rightarrow \quad o_{\text{max}} \geq o_y \qquad \forall y \in \mathcal{Y} \tag{4.5.2a}$$

$$o_{\text{max}} \leq o_y + (\mathbf{u} - l_y)(1 - \delta_y) \qquad \forall y \in \mathcal{Y} \tag{4.5.2b}$$

$$\delta_y \in \{0, 1\} \qquad \forall y \in \mathcal{Y} \tag{4.5.2c}$$

$$\sum_{y \in \mathcal{Y}} \delta_y = 1, \tag{4.5.2d}$$

with l_y, u_y being a lower/upper bound on the value of o_y and $\mathbf{u} = \max_{y \in \mathcal{Y}} u_y$ (these can again be computed using interval arithmetic). Implementing the maximum in this way gives us both a variable representing the value of the maximum ($o_{\max}$), as well as a one hot encoding of the argmax (δ_y). If the embedding for each token is given by $\{\text{emb}_i \mid i \in \mathcal{Y}\}$, we can simply encode the input to the following RNN timestep as $\sum_{y \in \mathcal{Y}} \delta_y \cdot \text{emb}_y$, which is a linear function of the variables that we previously constructed.

With this mechanism to encode the greedy decoding, we can now unroll the recurrent model for the desired number of timesteps. To search for an input x with output length $L(x) \geq \hat{K}$, we unroll the recurrent network for $\hat{K}$ steps and attempt to prove that at each timestep, **eos** is not the maximum logit, as in (4.4.2). We set the problem as:

$$\max \min_{t=1..\hat{K}} \left[\max_{z \neq \textbf{eos}} o_t[z] - o_t[\textbf{eos}] \right], \tag{4.5.3}$$

where $o(k)$ represents the logits in the k-th decoding step. To encode the $\min$ operation in the objective function (4.5.3), we use a strategy analogous to the one of Equation (4.5.2). If the global optimal value of Eq. (4.5.3) is positive, this is a valid counterexample: at all timesteps $t \in [1..\hat{K}]$, there is at least one token greater than the **eos** token, which means that the decoding should continue. On the other hand, if the optimal value is negative, that means that those conditions can't be satisfied and that it's not possible to generate a sequence of length greater than $\hat{K}$. The **eos** token would necessarily be predicted before. This would imply that our robustness property is True.

4.6 Target Model Mechanism

We use image captioning and machine translation models as specific target examples to study the output length modulation problem. We now introduce their mechanism.

4.6.1 Image captioning models

The image captioning model we consider is an encoder-decoder model composed of two modules: a convolutional neural network (CNN) as an encoder for image feature extraction and a recurrent neural network (RNN) as a decoder for caption generation [162].

Formally, the input to the model x is an $m \times n$ sized image from the space $\mathcal{X} = [-1, 1]^{m \times n}$, the CNN-RNN model computes the output sequence as follows, where emb denotes

the embedding function:

$$i_0 = \text{CNN}(x); \tag{4.6.1a}$$

$$h_0 = \mathbf{0} \tag{4.6.1b}$$

$$o_t, h_{t+1} = \text{RNNCell}(i_t, h_t) \tag{4.6.1c}$$

$$y_t = \arg\max(o_t), \tag{4.6.1d}$$

$$i_{t+1} = \text{emb}(y_t) \tag{4.6.1e}$$

The captioning model first runs the input image x through an CNN to obtain the image embedding and feeds it to the RNN as the initial input i_0 along with the initial state h_0. At each decode step, the RNN uses the input i_t and state h_t to compute the new state h_{t+1} as well as the logits o_t representing the log-probability of the output token distribution in the vocabulary. The output y_t is the token in the vocabulary with highest probability based on o_t, and it is embedded into the continuous space using an embedding matrix W_{emb} as $W_{emb}[y_t]$. The embedding is fed to the next RNN cell as the input for the next decoding step.

4.6.2 Machine translation models

The machine translation model is an encoder-decoder model [40, 154] with both the encoder and the decoder being RNNs. Given the vocabulary V_{in} of the input language, the valid input space $\mathcal{X}$ is defined as all sentences composed of tokens in V_{in}, i.e., $\mathcal{X} = \{(x_1, \ldots, x_n) \mid x_i \in V, n > 0\}$. Given an input sequence $x = (x_1, \ldots, x_n)$, the model first calculates its embedding $f(x)$ RNN as follows (h^e_t, i^e_t denotes the encoder hidden states and the inputs at the t-th time step, and emb^e denotes the embedding function for each token in the vocabulary).

$$h^e_0 = \mathbf{0} \tag{4.6.2a}$$

$$i^e_t = \text{emb}^e(x_t) \tag{4.6.2b}$$

$$h^e_t = \text{RNNCell}^e(i^e_t, h^e_{t-1}) \tag{4.6.2c}$$

$$f(x) = h^e_n \tag{4.6.2d}$$

The model then use $f(x)$ as the initial state h_0 for the decoder RNN to generate the output sequence, in the same way as the image captioning model.

4.7 Experiments

We consider the following three models, namely, Multi-MNIST captioning, Show-and-Tell [162], and Neural Machine Translation (NMT) [13, 40, 154] models.

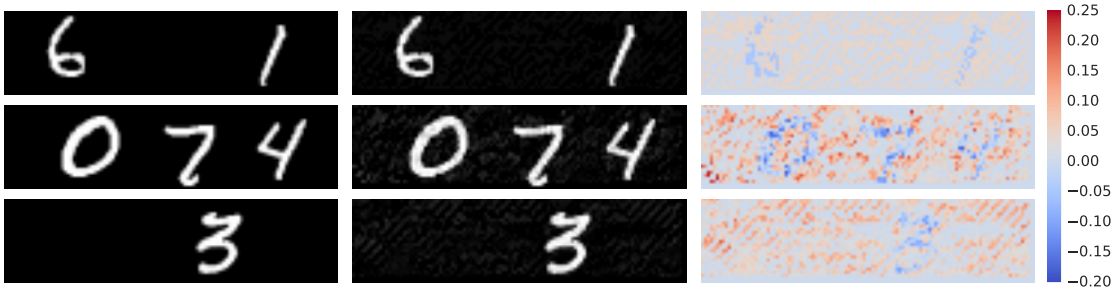


Figure 4.1: Multi-MNIST examples (left), adversarial examples found by PGD attack (mid), and their differences. For the first group, the model correctly predicts label $l_1 = [6, 1]$ on the original image but predicts $l_1' = [6, 1, 1]$ for its corresponding adversarial input. Predictions on the original/adversarial inputs made by model for the second group are $l_2 = [0, 7, 4], l_2' = [0, 1, 4, 3]$, and $l_3 = [3], l_3' = [3, 3, 5, 3]$ for the third group. The adversarial inputs in the first/second/third groups are found within the perturbation radius $\delta_1 = 0.1, \delta_2 = 0.25, \delta_3 = 0.25$.

4.7.1 Details of models and datasets

Multi-MNIST. The first model we evaluate is a minimal image captioning model for Multi-MNIST dataset. The Multi-MNIST dataset is composed from the MNIST dataset (Figure 4.1 left). Each image in the dataset is composed from 1-3 MNIST images: each MNIST image (28 * 28) is placed on the canvas of size (28 * 112) with random bias on the x-axis. The composition process guarantees that every MNIST image is fully contained in the canvas without overlaps with other images. The label of each image is the list of MNIST digits appearing in the canvas, ordered by their x-axis values. The dataset contains 50,000 training images and 10,000 test images, where the training set is constructed from MNIST training set and the test set is constructed from MNIST test set. The images are normalized to $[-1, 1]$ before feeding to the captioning model. For this dataset, we train a CNN-RNN model for label prediction. The model encoder is a 4-layers CNN (2 convolution layers and 2 fully connected layers with Relu activation functions applied in between). The decoder is a RNN with ReLU activation. Both the embedding size and the hidden size are set to 32. We train the model for 300 steps with Adam optimizer based on the cross-entropy loss. The model achieves 91.2% test accuracy, and we observe that for inputs from the training set, all generated outputs have a length smaller than 3.

Show-and-Tell. Show and Tell model [162] is an image captioning model with CNN-RNN encoder-decoder architecture similar to the Multi-MNIST model trained on the MSCOCO 2014 dataset [107]. Show-and-Tell model uses Inception-v3 as the CNN encoder and an LSTM for caption generation. We use a public version of the pretrained model[3] for evaluation. All images are normalized to $[-1, 1]$ and all the ground truth captions in the dataset have a length smaller than 20.

[3] https://github.com/tensorflow/models/

NMT. The machine translation model we study is a Seq2Seq model [40, 154] with the attention mechanism [13] trained on the WMT15 German-English dataset. The model uses byte pair segmentation (BPE) subword units [145] as vocabulary. The input vocabulary size is $36,548$. The model consists of 4-layer LSTMs of 1024 units with bidirectional encoder, with embedding dimension set to 1024. We use a publicly available checkpoint[4] with 27.6 BLEU score on the WMT15 test datasets in our evaluation. At training time, the model restricts the length of the generated sequence to 50.

4.7.2 Adversarial Attacks

Our first experiment studies whether adversarial inputs exist for the above models and how they affect model decoding. For each model, we randomly select 100 inputs from the development dataset as attack targets, and compare the output length distributions from random perturbation and PGD attacks.

Multi-MNIST

We evaluate the distribution of output lengths of images with a L^∞ perturbation radius of $\delta \in \{0.001, 0.005, 0.01, 0.05, 0.1, 0.5\}$ using both random search and PGD attack. In random search, we generate 10,000 random images within the given perturbation radius for each image in the target dataset as new inputs to the model. In PGD attack, the adversarial inputs are obtained by running 10,000 gradient descent steps with a learning rate of 0.0005 using the Adam Optimizer.

Neither of the attack methods can find any adversarial inputs for $\delta \in \{0.001, 0.005, 0.01\}$ perturbation radius (i.e., no perturbation is found for any images in the target dataset within the above δ to generate an output sequence longer than the original one). Figure 4.2 shows the distribution of the output lengths for images with different perturbation radius. Results show that the PGD attack is successful at finding attacks that push the distribution of output lengths higher, particularly at larger values of δ. Examples of adversarial inputs found by the model are shown in Figure 4.1.

Show-and-Tell

For the Show-and-Tell model, we generate attacks within an L^∞ perturbation radius of $\delta = 0.5$ with both random search and PGD attack on 500 images randomly selected from the development dataset. However, except one adversarial input found by PGD attack that would cause the model to produce an output with size 25, no other adversarial inputs are found that can cause the model to produce outputs longer than 20 words,

[4] https://github.com/tensorflow/nmt

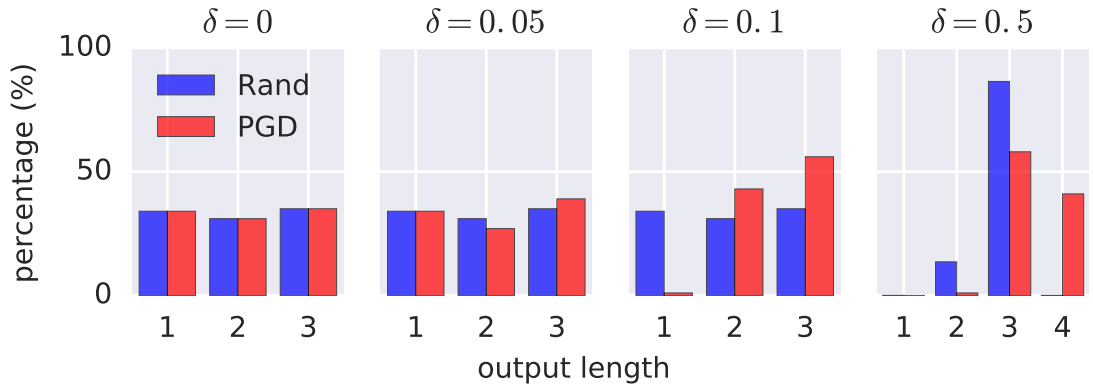


Figure 4.2: The distribution of output length for random search (denoted as **Rand**) and PGD attack with different perturbation radius δ. The x-axis denotes the output length and the y-axis denotes the number of outputs with the corresponding length. $\delta = 0$ (no perturbation allowed) refers to the original output distribution of the target dataset.

which is the training length cap. Our analysis shows that the difficulty of attacking the model is resulted from its strong bias on the output sequence distribution and the saturation of sigmoid gates in the LSTM decoder. This result is also consistent with the result found by [32] that Show-and-Tell model is "only able to generate relevant captions learned from the training distribution".

Neural Machine Translation

We evaluate the NMT model by comparing the output length distribution from adversarial examples generated from random search and PGD attack algorithms. We randomly draw 100 input sentences from the development dataset. The maximum input length is 78 and their corresponding translations made by the model are all within 75 tokens. We consider the perturbation $\delta \in \{0.3, 0.5, 1.0\}$.

1. *Random Search.* In each run of the random attack, given an input sequence with length n, we first randomly select $\lceil \delta \cdot n \rceil$ locations to modify, then randomly select substitutions of the tokens at these locations from the input vocabulary, and finally run the NMT model on the modified sequence. We run 10,000 random search steps for the 100 selected inputs, and show the distributions of all outputs obtained from the translation (in the total 1M output sequences).
2. *PGD Attack.* In PGD attack, we also start by randomly selecting $\lceil \delta \cdot n \rceil$ locations to modify for each input sequence with length n. We then run 800 iterations of PGD attack with Adam optimizer using an initial learning rate of 0.005 to find substitutions of the tokens at these selected locations. We plot the output length obtained from running these adversarial inputs through the translation model.

Figure 4.3 shows the distribution of output sequence lengths obtained from random search methods with different δ. We aggregate all sequences with length longer than 100 into the group '>100' in the plot. Results show that even random search approach could often craft inputs such that the corresponding output lengths are more than 75 and occasionally generates sentences with output length over 100. The random search algorithm finds 79, 11, 3 for δ =0.3, 0.5, 1, respectively, among the 1M translations that are longer than 100 tokens (at small δ, the search space is more restricted, and random search has a higher success rate of finding long outputs). Notably, the longest sequence found by the random search is a sequence with output length 312 tokens, where the original sequence is only 6.

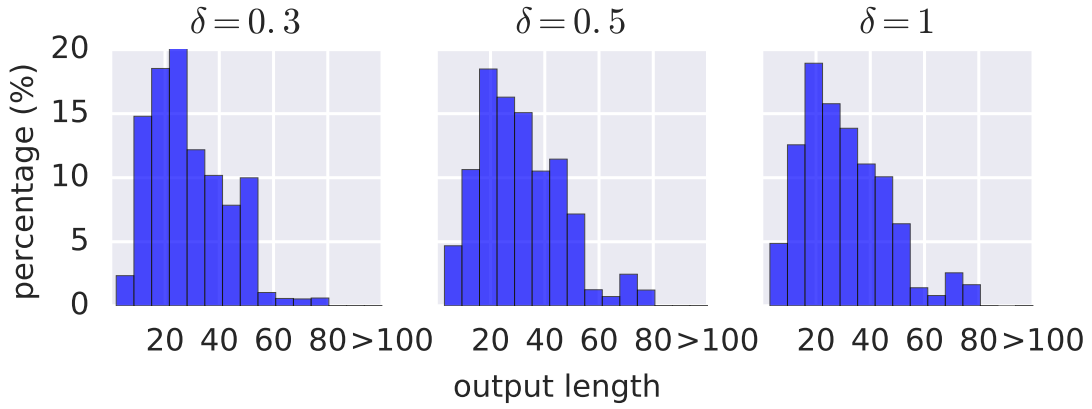


Figure 4.3: The histogram representing the output length distribution of the NMT model using random search with different perturbations ($\delta \in \{0.3, 0.5, 1\}$). The x-axis shows the output length. y-axis values are divided by 10,000, the number of random perturbation rounds per image.

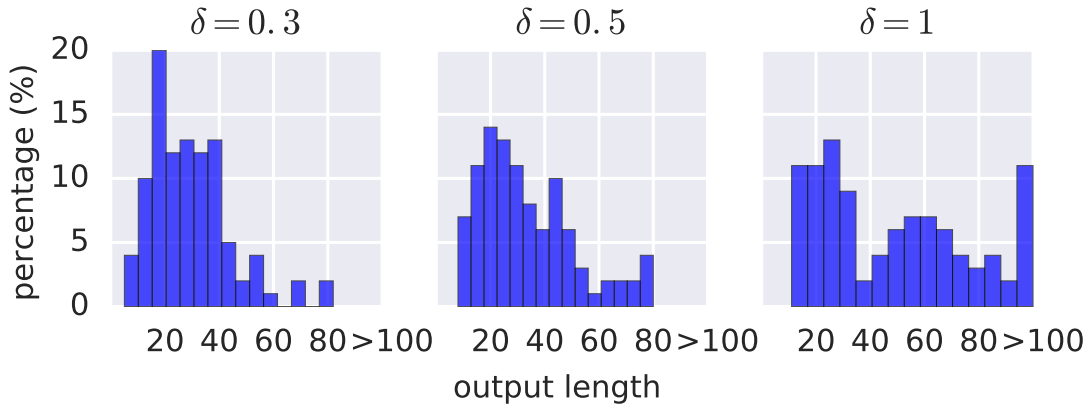


Figure 4.4: The histogram representing the output length distribution of the NMT model under PGD attack for different δ. x-axis shows the output length and y-axis shows the number of instances with the corresponding length.

(I) Die Waffe wird ausgestellt und durch den Zaun übergeben.
(O) The weapon is issued and handed over by the fence . ***eos***
(I′) Die namen name descri und ames utt origin i.e. meet grammatisch .
*(O′) names name names name names grammatically name names names names

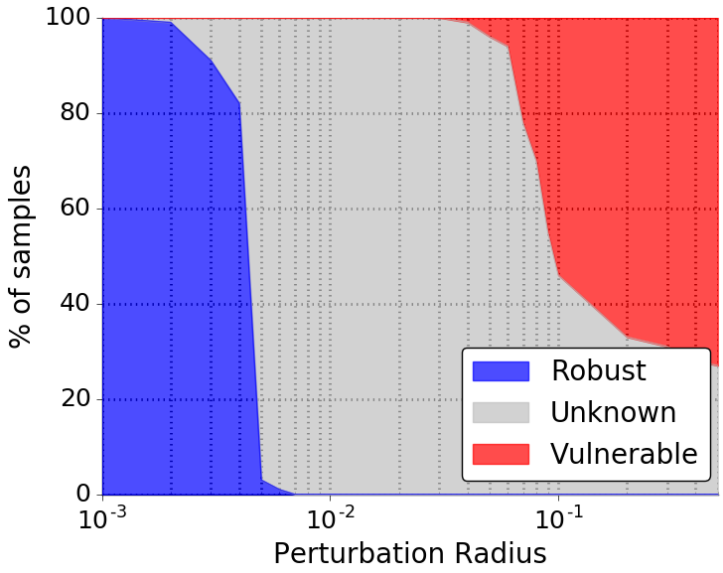


Figure 4.6: Proportion of samples being provably robust (in blue), vulnerable (in red) or of unknown robustness status (in gray) to an attack attempting to make the model generate an output sequence longer than the ground truth, as a function of the perturbation radius allowed to the attacker. For small radiuses, the MIP can prove that no attacks can be successful. For large radiuses, we are able to find successful attacks.

increases, our verification algorithm times out and is not able to explore the full space of valid perturbations and thus cannot decide whether attacks exists in the given space. For this reason, the number of robust samples we report is only a lower bound on the actual number of robust samples. Conversely, the vulnerable samples that we exhibit give us an upper bound on the number of those robust samples. As shown by the large proportion of samples of unknown status, there is currently still a gap between the capabilities of formal verification method and attacks.

4.8 Conclusion

In this chapter, we introduced the existence and the construction of the output-length modulation problem. We proposed a differentiable proxy that can be used with PGD to efficiently find output-lengthening inputs. We also developped a verification approach to formally prove certain models cannot produce outputs greater than a certain length. We showed that the proposed algorithm can produce adversarial examples that are 50 times longer than the input for machine translation models, and the image-captioning model can conform the output size is less than a given maximum length using the verification approach. In future work, we plan to study adversarial training of sequential output models using the generated attacks, to models that are robust against output lengthening attacks, and further, verify this formally.

In the previous chapter, we showed that it was possible to achieve better performance for complete verification by using specifically tailored branch-and-bound algorithms rather than relying on mixed integer programming solvers. It might therefore seem surprising that we still used an off-the-shelf solver for verification in this chapter.

Despite having the potential to be more efficient, branch-and-bound solvers tailored to the specific problem require more tuning and considerably more labour to use. Generic mixed integer programming solvers will have acceptable performance by default in most cases. When attempting to determine the feasibility of verifying a new type of property, we argue that it is reasonable to experiment with the more flexible approach first in order to iterate more quickly. More performant specialized approaches can be developped at a later stage.

4.9 Additional Notes

4.9.1 NMT Examples

We show a few more examples of the NMT model. We use I_k, O_k to refer to original inputs and their translations produced by the model, and I'_k, O'_k for adversarial inputs / outputs. The following adversarial examples are found by with $\delta = 1$ (unconstrained perturbation in the discrete token space).

(I_1) Eink-aufen vom So-fa aus

(O_1) Shopping from the sofa ***eos***

(I'_1) Schlusselwortern Anreise nähe Hotelsafe kes hotel

(O'_1) keyword 4 Com-city Hotel near Safety Deposit Box kes hotel kes garage kes kes kes kes kes kes kes kes kes kes kes kes kes hotel kes kes next street kes kes garage kes kes hotel kes kes kes kes kes kes kes kes Hotel kes kes kes kes kes kes kes Hotel kes kes kes kes kes kes kes kes kes kes kes kes kes kes kes kes kes kes kes kes kes kes kes kes kes kes kes kes ***eos***

(I_2) Un-zufrie-dene

(O_2) Related dis-satisfied ***eos***

(I'_2) Anmeldung aspects ICEcat

*(O'_2) Regi- ment aspects ICEcat ICEcat ICEcat

*ICEcat ICEcat ICEcat

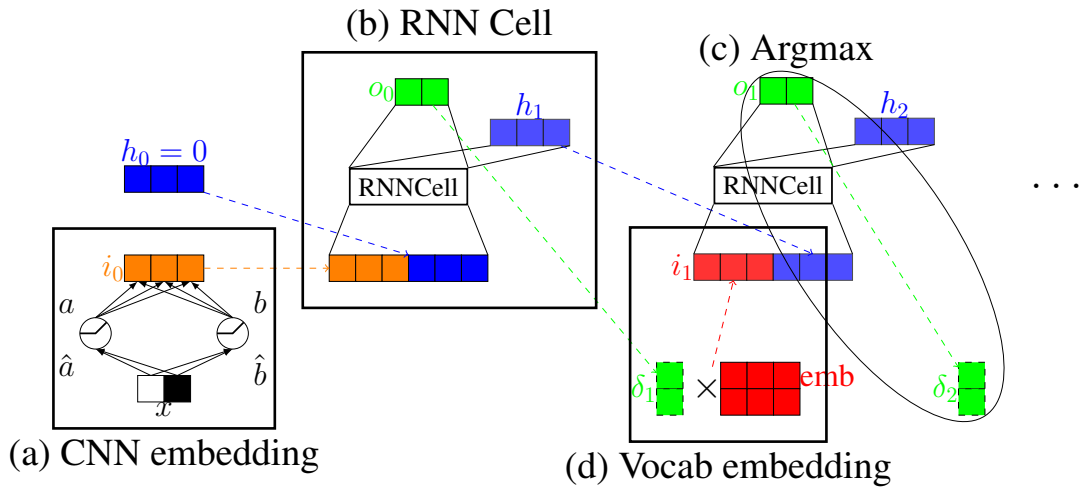


Figure 4.7: Illustration of a toy model like the one we used for the Multi-MNIST application. (a) The input image is composed of two pixels, the image embedding network has one hidden layer with two neurons a and b, from which the inital input i_0 to the network is generated. (b) At each time step, the input to the network i_k is combined (here simply concatenated) with the state h_k, based on which this step's output o_k and the next stat h_{k+1} are computed. (c) The maximum output of o_k is identified and fed back into the network. (d) The input of the next step of the network is the vocabulary embedding corresponding to the maximum output.

We would then need to create some variables corresponding to the input of the hidden layer, which would be constrained by a linear equality depending on the network input variable, and the network weights W_1

$$\underline{\hat{a}} = W_{1,a}\underline{x} + b_{1,a} \qquad \underline{\hat{b}} = W_{1,b}\underline{x} + b_{1,b} \tag{4.9.2}$$

The next step consists in creating variables representing the output of the hidden layer, after the ReLU activation function. In order to do so, we need to first obtain bounds on $\underline{\hat{a}}$ and $\underline{\hat{b}}$. Interval analysis is one of the way to achieve this:

$$l_a = W^+_{1,a} l_0 + W^-_{1,a} u_0 + b_{1,a}$$
$$u_a = W^+_{1,a} u_0 + W^-_{1,a} l_0 + b_{1,a},$$

which bounds on $\underline{\hat{b}}$ being obtained similarly.

Using those bounds, we can now generate variables representing the ReLU outputs, and constrain them to only take the appropriate values. Using the formulation described in Equation (4.5.1), the constraints required for the ReLU A are:

$$\begin{aligned} &\underline{\delta_a} \in \{0, 1\} \\ &\underline{a} \geq 0 \\ &\underline{a} \geq \underline{\hat{a}} \\ &\underline{a} \leq u_a \underline{\delta_a} \\ &\underline{a} \leq \underline{\hat{a}} - l_a \left(1 - \underline{\delta_a}\right) \end{aligned} \tag{4.9.3}$$

The outputs of the embedding network can be constructed in a manner similar to the way we constructed the input to the hidden layer, by enforcing a single linear equality constraint:

$$\underline{i_0} = W_2 \begin{bmatrix} \underline{a} \\ \underline{b} \end{bmatrix} + b_2 \tag{4.9.4}$$

and we can easily derive bounds (l_{i_0}, u_{i_0}) by using the same interval analysis method from before.

We have now seen how to represent the allowed images, as well as the corresponding activations throughout the embedding network, as a set of variables and the linear constraints that they should satisfy. We will now move on to explaining how the recurrent part of the network is encoded.

4.9.4 RNN Cell Encoding

The encoding of the recurrent cell is exactly similar to the CNN embedding described in the previous section.

Based on an initial inputs $\left(\underline{i_k}, \underline{h_k}\right)$, for which we have bounds and constraints, we will create the new variables representing the RNN cell activations corresponding to those inputs. Our example doesn't specifically show how the RNN cell is implemented, but provided that it is composed of only linear operations (which will give constraints in the style of Equations (4.9.2) and (4.9.4)) or ReLU operations (with constraints in the style of Equations (4.9.3)), the previously described techniques will be able to generate constrained variables $\left(\underline{o_k}, \underline{h_{k+1}}\right)$, as well as upper and lower bound for those variables.

4.9.5 Argmax Encoding

We will now show how we can create a variable that represents the argmax of the RNN cell output. In our example, the size of the vocabulary is 2, but the technique, described in Equation 4.5.2, is valid beyond that.

Assuming we have already created the variables $\underline{o_k}$, and we have access to their bounds (l_{o_k}, u_{o_k}), we will create the new variables $\underline{o_{k,\max}}$ and $\underline{\delta_k}$, subject to the constraints:

$$\begin{aligned}
&\underline{\delta_k[0]} \in \{0,1\} && \underline{\delta_k[1]} \in \{0,1\} \\
&\underline{o_{k,\max}} \geq \underline{o_k[0]} && \underline{o_{k,\max}} \geq \underline{o_k[1]} \\
&\underline{o_{k,\max}} \leq \underline{o_k[0]} + (\max[u_{o_k}] - l_{o_k}[0])\left(1 - \underline{\delta_k[0]}\right) \\
&\underline{o_{k,\max}} \leq \underline{o_k[1]} + (\max[u_{o_k}] - l_{o_k}[1])\left(1 - \underline{\delta_k[1]}\right) \\
&\underline{\delta_k[0]} + \underline{\delta_k[1]} = 1
\end{aligned} \tag{4.9.5}$$

The value $\underline{o_{k,\max}}$ can be used when we need to encode the values of the maximum (such as in the inner term of Equation (4.5.3)), and we will be able to use the argmax variable $\underline{\delta_k}$ in order to implement the greedy decoding mechanism.

4.9.6 Vocabulary Embedding

We need to generate the variable $\underline{i_{k+1}}$ that corresponds to the embedding of the token that was last predicted by the network. We have access to the $\underline{\delta_k}$ variable, which is by construction a one-hot binary variable indicating which was the generated token. We can simply encode:

$$\underline{i_{k+1}} = \sum_{y=1..2} \underline{\delta_k[y]} \times \text{emb}[y] \tag{4.9.6}$$

Given that only one of the $\underline{\delta_k[y]}$ is going to be equal to 1, all the other being 0, this will exactly correspond to choosing the embedding corresponding to the selected token.

Note that in order to be able to continue to build the Mixed Integer Programming problem, we need to have access to lower and upper bounds on the values taken by $\underline{i_{k+1}}$. These can be simply obtained by taking a maximum / minimum along the token dimension of the token embeddings:

$$l_{i_{k+1}} = \min_{y=1..2} \text{emb}[y]$$
$$u_{i_{k+1}} = \max_{y=1..2} \text{emb}[y]$$

Using these bounds, as well as the derived bounds on $\underline{h_{k+1}}$ that came from propagating bounds through the RNN cell, we now have variables and bounds for the next step of RNN enrolling. This enrolling can be repeated until the target length to obtain a MIP encoding of the sequence generated by the network.

Inputting all these constraints into a Mixed Integer Programming solver will then generate the desired formal verification result.

5 A Proximal Solver for Neural Network Bounds

5.1 Preamble

Several methods have been proposed for computing bounds on the values that the outputs of a neural network can take. To obtain tight bounds, it is necessary to solve a large number of linear programs, one per unit of the neural Network. Previous methods have either used off the shelf solvers, discarding the problem structure, or relaxed the problem even further, making the bounds unnecessarily conservative. We propose a novel algorithm based on the Augmented Lagrangian method, which operates in the dual. It offers three advantages: (i) it is based on operations analogous to forward/backward pass of neural networks layers and is therefore easily parallelizable, amenable to GPU implementation and able to take advantage of the convolutional structure of problems; (ii) it allows for anytime stopping while still providing valid bounds; and (iii) it replaces the difficult choice of step-size schedules with the simpler task of setting weights of proximal terms. We empirically show that we can obtain bounds comparable with off the shelf solvers at only a fraction of the running time.

5.2 Introduction

As machine learning powered systems become more and more common, the lack of robustness of neural networks and their reputation for being "Black Boxes" is increasingly worrisome. As they start to be deployed in critical scenarios where safety and robustness would be a prerequisite, it becomes necessary to develop techniques to properly test them.

In order to tackle this problem, an emerging field of research aims to develop methods that prove formal guarantees about their behaviour. One particular example of this trend is the community's focus on robustness to adversarial examples [70, 155]: perturbations maliciously crafted with the intent of fooling even extremely well performing models. After several published approaches claiming to suppress this phenomenon were shown to still be susceptible [11, 29, 160], some progress has been made in being able to formally verify whether or not there existed any adversarial examples in the neighbourhood of a data point [157, 171].

A critical component of the systems capable of providing guarantees developed so far is the computation of lower and upper bounds on the output of neural networks when their inputs are constrained to lie in a bounded set. One application is to identify safe regions of the input space, by deriving bounds on the changes of the prediction vector under restricted perturbations. These results allow the rigorous comparison of adversarial defenses and prevent making overconfident statements about their efficacy [171]. Bounds might also be used as a training signal, so as to guide the network towards greater robustness and more verifiability [71, 121, 171]. They can also be used as essential subroutines of branch and bound complete verifiers [25].

Previous algorithms for computing bounds are either computationally expensive [58] or sacrifice tightness for scalability [71, 121, 171]. In this chapter, we design a novel customised solver for obtaining bounds on neural networks. Our approach offers the following advantages:

- The basic step of the algorithm are linear operations similar to the ones used during forward/backward pass of the network. Our method is therefore **easily parallelizable**. To derive bounds on the output of a neural network, it is necessary to obtain intermediate bounds over all activations. To compute these, we need to solve two problems for each hidden unit of the network (one for the upper bound and one for the lower bound). Our solver takes advantage of the similarities between these problems. Parallelism allows us to achieve speedups by solving batches of problems on GPUs instead of serially solving them on CPUs. In addition, our method allows us to **leverage the convolutional structure** when necessary, while standard solvers are often restricted to treating it as a general linear operation.
- Standard linear programming based relaxations [58] will only return a valid bound if the problem is solved to optimality. The method described in this chapter is **anytime** and interrupting it even before complete convergence still provides a valid bound. This is useful in the context of a subroutine for complete verifiers, as this enables the user to choose an appropriate speed versus accuracy trade-off. It can also be used as a standalone incomplete verification method.

- Our algorithm decomposes the problem into a series of strongly convex subproblems. For each of these, we use an iterative method for which we derive **optimal step sizes**. Consequently, as opposed to a subgradient-based baseline [53], we don't need to choose a step size schedule. While the algorithm still requires to choose proximal terms, its performance is less sensitive to this choice in practice. This allows us to achieve tighter bounds for the same computational budget.

5.3 Related Work

Bound computation is mainly used for formal verification methods. Some methods are complete [36, 58, 94, 157, 174], always returning a verdict for each problem instances. Others are incomplete , based on relaxations of the verification problem. They trade speed for completeness: while they can't verify properties for all problem instances, they scale significantly better. Two main styles can be observed: on the one hand, some approaches rely on off-the-shelf solvers ([58, 143]) to solve accurate relaxation such as Planet [58], which is the best known linear-sized relaxation of the problem. On the other hand, as Planet and other more complex relaxations don't have closed form solutions, some researchers have also proposed easier to solve, looser formulations. For example, Wong and Kolter [171] considered the Lagrangian dual of the problem and used a specific choice of dual variables to derive a bound. Weng et al. [169] achieved an equivalent relaxation by not taking the convex hull of the ReLU, but a larger overapproximation, leading to a problem with a closed form solution. Others relaxed the problem even further in order to obtain faster solutions, either by propagating intervals [71], or through abstract interpretation [121]. Explicitly or implicitly, these are all equivalent to propagating a convex domain through the network to overapproximate the set of reachable values. Our approach consists in tackling a relaxation equivalent to the Planet one (although generalised beyond ReLU), by designing a custom solver that achieves faster performance, without sacrificing tightness.

Some potentially tighter convex relaxations exist but involve a quadratic number of variables, such as the semi-definite programming method of Raghunathan et al. [137] . Better relaxation obtained from strong Mixed Integer Programming formulations [8] either have a quadratic number of variables or a potentially exponential number of constraints. We don't address them in this chapter.

A related approach to ours is the work of Dvijotham et al. [53]. Both their method and ours are anytime and operate on similar duals. Even though the duals are derived differently (dual of the non-relaxed problem versus dual of the convex relaxation [58]) the two duals share a lot of similarities. We study their relation in detail in the additional

notes 5.8.4, showing that our dual problem provides better bounds when evaluated on the same dual variables. The main difference between their approach and ours lies in the optimisation strategy followed: they optimise the bounds using subgradient descent over the dual variables, while we use a proximal method for which we can derive optimal step sizes. We re-implement their method on our improved dual as a strong baseline and show that our solver still delivers a significant improvement in performance.

5.4 Preliminaries

Throughout this chapter, we will use bold lower case letters (like $\mathbf{z}$) to represent vectors and upper case letters (like W) to represent matrices. Brackets ($\mathbf{z}[i]$) are used to indicate the i-th coordinate of a vector. We will study the computation of the lower bound problem based on a feedforward neural network, with element-wise activation function $\sigma(\cdot)$. The network inputs are restricted to a convex domain $\mathcal{C}$, over which we assume that we can easily optimise linear functions. This is the same assumption that was made by Dvijotham et al. [53]. The computation for an upper bound is analogous. Formally, we wish to compute the following:

$$\min_{\mathbf{z},\hat{\mathbf{z}}} \quad \hat{\mathbf{z}}_n \qquad \text{s.t.} \quad \mathbf{z}_0 \in \mathcal{C} \tag{5.4.1a}$$

$$\hat{\mathbf{z}}_{k+1} = W_{k+1}\mathbf{z}_k + \mathbf{b}_{k+1} \qquad \text{for } k \in [0..n-1] \tag{5.4.1b}$$

$$\mathbf{z}_k = \sigma(\hat{\mathbf{z}}_k) \qquad \text{for } k \in [1..n-1], \tag{5.4.1c}$$

The activation of the network, before being passed through the activation function of layer k are denoted by $\hat{\mathbf{z}}_k$, while $\mathbf{z}_k$ represents their value after. For ease of notation, we use $\mathbf{z}$ and $\hat{\mathbf{z}}$ to refer to the ensemble of all these variables ($\mathbf{z} = \{\mathbf{z}_i\}_{i\in\{0\dots n-1\}}$ and $\hat{\mathbf{z}} = \{\hat{\mathbf{z}}_i\}_{i\in\{1\dots n\}}$). Constraints (5.4.1b) implements the linear layers (fully connected or convolutional, defined by the weight matrix $W_k \in \mathbb{R}^{h_k \times h_{k-1}}$ and the bias vector $\mathbf{b}_k \in \mathbb{R}^{h_k}$) while constraints (5.4.1c) implement the non-linear activation function. Constraints (5.4.1a) define the region bounds are being computed on. While our method can be extended to more complex networks (such as ResNets), we focus on problem (5.4.1) for the sake of clarity.

The difficulty of problem (5.4.1) comes from the non-linearity (5.4.1c). While several bounds have been proposed in the literature, we deal with a relaxation that is equivalent to the Planet relaxation, as introduced by Ehlers [58] in the context of ReLU. We will use it as an example throughout the chapter but this can be extended to other non-linearities. For example, subsection 5.8.2 describes the case of sigmoid activation function. The relaxation independently considers each activation function constraint in the network, and

replaces it with constraints corresponding to its convex hull. This relaxation is the tightest linearly sized relaxation published thus far. By linearly sized, we mean that the numbers of variables and constraints necessary to describe the relaxation only grows linearly with the number of units in the network. Formally, we solve the following convex program:

$$\min_{\mathbf{z},\hat{\mathbf{z}}} \quad \hat{\mathbf{z}}_n \qquad \text{s.t.} \quad \mathbf{z}_0 \in \mathcal{C} \tag{5.4.2a}$$

$$\hat{\mathbf{z}}_{k+1} = W_{k+1}\mathbf{z}_k + \mathbf{b}_{k+1} \qquad \text{for } k \in [0..n-1] \tag{5.4.2b}$$

$$\texttt{cvx_hull}_\sigma\left(\hat{\mathbf{z}}_k, \mathbf{z}_k, \mathbf{l}_k, \mathbf{u}_k\right) \qquad \text{for } k \in [1..n-1]\,. \tag{5.4.2c}$$

A notable point is that the relaxation of the activation (5.4.2c) depends on knowing upper ($\mathbf{u}_k$) and lower bounds ($\mathbf{l}_k$) on the value that $\hat{\mathbf{z}}_k$ can take, for $k \in [0..n-1]$. For ReLUs, this convex hull takes the following form:

$$\texttt{cvx_hull}_\sigma\left(\hat{\mathbf{z}}_k, \mathbf{z}_k, \mathbf{l}_k, \mathbf{u}_k\right) \equiv \begin{cases} \mathbf{z}_k[i] \geq 0 \quad \mathbf{z}_k[i] \geq \hat{\mathbf{z}}_k[i] & \text{if } \begin{matrix}\mathbf{l}_k[i] \leq 0 \\ \mathbf{u}_k[i] \geq 0\end{matrix} \\ \mathbf{z}_k[i] \leq \dfrac{\mathbf{u_k}[i](\hat{\mathbf{z}}_k[i] - \mathbf{l}_k[i])}{\mathbf{u_k}[i] - \mathbf{l_k}[i]} & \text{(5.4.3a)} \\ \mathbf{z}_k[i] = 0 \qquad \text{if } \mathbf{u}_k[i] \leq 0 & \text{(5.4.3b)} \\ \mathbf{z}_k[i] = \hat{\mathbf{z}}_k[i] \qquad \text{if } \mathbf{l}_k[i] \geq 0. & \text{(5.4.3c)} \end{cases}$$

Constraints (5.4.3a), (5.4.3b), and (5.4.3c) correspond to the relaxation of the ReLU (5.4.1c) in different cases (respectively: ambiguous state, always blocking or always passing).

The values $\mathbf{l}_k$ and $\mathbf{u}_k$ are obtained by solving (5.4.2) over subsets of the network as follows. Instead of defining the objective function of (5.4.2) on the activation of layer n, we start by defining it over $\hat{\mathbf{z}}_2$ (due to our assumptions over $\mathcal{C}$, the bounds for $\hat{\mathbf{z}}_1$ are easy to compute). This allows us to compute all the elements of $\mathbf{l}_2$ and $\mathbf{u}_2$. Repeating this procedure, we similarly compute the *intermediate bounds* $\mathbf{l}_k$ and $\mathbf{u}_k$ on activation $\hat{\mathbf{z}}_k$ by solving all the problems dealing with only the first k layers, once we have obtained all bounds for the first $k-1$ layers. Computing those intermediate bounds is the main computational cost of this approach as we need to solve two LPs for each of the neurons in the network. Using standard off-the-shelf LP solvers forces us to solve all of these problems sequentially.

5.5 Algorithm

We will now describe a novel algorithm to solve problem (5.4.2). A summary can be found in subsection 5.8.3. We will focus on computing bounds for an output of the last layer of the neural network. However, intermediate bounds can be analogously obtained.

5.5.1 Lagrangean Decomposition of the Problem

Our approach is based on Lagrangean decomposition [75]. Due to the hierarchical structure of neural networks, most constraints involve only a very limited number of variables. As a result, we will be able to split the problem into meaningful, easy to solve subproblems. We will then impose constraints that the solutions of the different subproblems should be the same.

To obtain a decomposition, we divide the constraints into subsets. Each subset will correspond to a pair composed of an activation layer and the following linear layer. The only exception is the first linear layer, which is combined with the restriction on the input domain to $\mathcal{C}$. This choice is motivated by the fact that for piecewise-linear activation functions, we will be able to easily perform linear optimisation over the resulting subdomains. For a different activation function, it might be required to have a different decomposition. Formally, we introduce the following notation for subsets of constraints:

$$\mathcal{P}_0(\mathbf{z}_0, \hat{\mathbf{z}}_1) \equiv \begin{cases} \mathbf{z}_0 \in \mathcal{C} \\ \hat{\mathbf{z}}_1 = W_1 \mathbf{z}_0 + \mathbf{b}_1, \end{cases} \tag{5.5.1}$$

$$\mathcal{P}_k(\hat{\mathbf{z}}_k, \hat{\mathbf{z}}_{k+1}) \equiv \begin{cases} \mathbf{l}_k \leq \hat{\mathbf{z}}_k \leq \mathbf{u}_k \\ \texttt{cvx_hull}_\sigma\left(\hat{\mathbf{z}}_k, \mathbf{z}_k, \mathbf{l}_k, \mathbf{u}_k\right) \\ \hat{\mathbf{z}}_{k+1} = W_{k+1}\mathbf{z}_k + \mathbf{b}_{k+1}. \end{cases} \tag{5.5.2}$$

The set $\mathcal{P}_k$ is defined without an explicit dependency on $\mathbf{z}_k$ because $\mathbf{z}_k$ only appears internally to the subset of constraints and is not involved in the objective function. Using this grouping of the constraints, we can concisely write the problem (5.4.2) as:

$$\begin{aligned} \min_{\mathbf{z}, \hat{\mathbf{z}}} \quad & \hat{\mathbf{z}}_n \\ \text{s.t.} \quad & \mathcal{P}_0(\mathbf{z}_0, \hat{\mathbf{z}}_1); \\ & \mathcal{P}_k(\hat{\mathbf{z}}_k, \hat{\mathbf{z}}_{k+1}), \qquad \text{for } k \in [1..n-1]\,. \end{aligned} \tag{5.5.3}$$

To obtain a Lagrangean decomposition, we duplicate the variables so that each subset of constraints has its own copy of the variables it is involved in. Formally, we rewrite problem (5.5.3) as follows:

$$\begin{aligned} \min_{\mathbf{z}, \hat{\mathbf{z}}} \quad & \hat{\mathbf{z}}_{A,n} \\ \text{s.t.} \quad & \mathcal{P}_0(\mathbf{z}_0, \hat{\mathbf{z}}_{A,1}); && && \text{(5.5.4a)} \\ & \mathcal{P}_k(\hat{\mathbf{z}}_{B,k}, \hat{\mathbf{z}}_{A,k+1}) && \text{for } k \in [1..n-1] && \text{(5.5.4b)} \\ & \hat{\mathbf{z}}_{A,k} = \hat{\mathbf{z}}_{B,k} && \text{for } k \in [1..n-1]\,. && \text{(5.5.4c)} \end{aligned}$$

The additional equality constraints (5.5.4c) impose agreements between the various copies of variables. We introduce the dual variables $\boldsymbol{\rho}$ and derive the Lagrangean dual:

$$\begin{aligned} q\left(\boldsymbol{\rho}\right) = \min_{\mathbf{z},\hat{\mathbf{z}}} \; & \hat{\mathbf{z}}_{A,n} + \Sigma_{k=1..n-1} \boldsymbol{\rho}_k^T \left(\hat{\mathbf{z}}_{B,k} - \hat{\mathbf{z}}_{A,k}\right) \\ \text{s.t.} \quad & \mathcal{P}_0(\mathbf{z}_0, \hat{\mathbf{z}}_{A,1}); \\ & \mathcal{P}_k(\hat{\mathbf{z}}_{B,k}, \hat{\mathbf{z}}_{A,k+1}) \qquad \text{for } k \in [1..n-1] \end{aligned} \tag{5.5.5}$$

Any value of $\boldsymbol{\rho}$ provides a valid lower bound on the problem, by virtue of weak duality. While we will maximise over the choice of dual variables $\boldsymbol{\rho}$ in order to obtain as tight a bound as possible, we will be able to interrupt the optimisation process at any point and obtain a valid bound by evaluating q.

The rest of the chapter aims at maximising (5.5.5). The formulation differs from the dual of Dvijotham et al. [53]: our decomposition considers larger subset of constraints and therefore results in a smaller number of variables to optimise. We show in subsection 5.8.4 that our formulation dominates theirs, producing tighter bounds based on the same dual variables. However, the optimal solution of both formulations are equivalent as they are both reformulations of the same convex problem. Our dual is also related to the one that Wong and Kolter [171] operates in. We show in subsection 5.8.5 how our problem can be initialised to a set of dual variables so as to generate a bound matching the one provided by their algorithm. We use this approach for our initialisation.

5.5.2 Proximal Method for Optimisation of the Bound

Applying proximal maximisation to the dual function q results in the Augmented Lagrangian Method, also known as the method of multipliers. The derivation of the update steps is given in detail by Bertsekas and Tsitsiklis [18]. For our problem, this will correspond to alternating between updates to the dual variables $\boldsymbol{\rho}$ and updates to the primal variables $\hat{\mathbf{z}}$, which are given by the following equations (with the superscript t indicating the value at the t-th iteration):

$$\boldsymbol{\rho}_k^{t+1} = \boldsymbol{\rho}_k^t + \frac{\hat{\mathbf{z}}_{B,k}^t - \hat{\mathbf{z}}_{A,k}^t}{\eta_k} \tag{5.5.6}$$

$$\begin{aligned} \left[\mathbf{z}^t, \hat{\mathbf{z}}^t\right] = & \operatorname*{argmin}_{\mathbf{z},\hat{\mathbf{z}}} \mathcal{L}\left(\hat{\mathbf{z}}, \boldsymbol{\rho}^t\right) \\ & \text{s.t.} \quad \mathcal{P}_0(\mathbf{z}_0, \hat{\mathbf{z}}_{A,1}); \\ & \qquad \mathcal{P}_k(\hat{\mathbf{z}}_{B,k}, \hat{\mathbf{z}}_{A,k+1}) \text{ for } k \in [1..n-1] \end{aligned}$$

$$\text{where } \mathcal{L}\left(\hat{\mathbf{z}}, \boldsymbol{\rho}\right) = \hat{\mathbf{z}}_{A,n} + \sum_{k=1..n-1} \boldsymbol{\rho}_k^T \left(\hat{\mathbf{z}}_{B,k} - \hat{\mathbf{z}}_{A,k}\right) + \sum_{k=1..n-1} \frac{1}{2\eta_k} \|\hat{\mathbf{z}}_{B,k} - \hat{\mathbf{z}}_{A,k}\|^2. \tag{5.5.7}$$

The term $\mathcal{L}(\hat{\mathbf{z}}, \boldsymbol{\rho})$ is the Augmented Lagrangian of problem (5.5.4). In (5.5.7), the additional quadratic term compared to the objective of $q(\boldsymbol{\rho})$ appears due to the choice of using proximal methods. It has the advantage of making the problem strongly convex, and hence easier to optimize. We will also show later that this allows us to derive optimal step-sizes in closed form. The weight η_k is a hyperparameter of the algorithm. A high value will make the problem more strongly convex and therefore quicker to solve, but it will also limit the ability of the algorithm to perform large updates.

While obtaining the new values of $\boldsymbol{\rho}$ is trivial using Equation (5.5.6), problem (5.5.7) does not have a closed form solution. We show in the next section how it can be efficiently solved nonetheless.

5.5.3 Frank-Wolfe Algorithm

Problem (5.5.7) can be optimised using the conditional gradient method, also known as the Frank-Wolfe algorithm [63]. The advantage it provides is that there is no need to perform a projection step to remain in the feasible domain. Indeed, the different iterates remain in the feasible domain by construction as convex combination of points in the feasible domain. At each time step, we replace the objective by its linear approximation and optimise this linear function over the feasible domain to get an update direction. We then take a step in this direction, named the conditional gradient.

Obtaining the conditional gradient requires solving the linear problem defined by the gradient of $\mathcal{L}(\hat{\mathbf{z}}, \boldsymbol{\rho})$ over the feasible domain. By design, each of the variables is only involved in one subset of constraints. As a result, computation of the conditional gradient completely decomposes over the subproblems, each corresponding to one of the subset of constraints. We therefore simply need to optimise linear functions over one subset of constraints at a time.

$\mathcal{P}_0$ Subproblems

To obtain the conditional gradient corresponding to the $\mathcal{P}_0$ constraints, we need to solve:

$$\begin{aligned}[\mathbf{x}_0, \hat{\mathbf{x}}_{A,1}] = \underset{\mathbf{z}_0, \hat{\mathbf{z}}_{A,1}}{\operatorname{argmin}} \; & \mathbf{c}^T \hat{\mathbf{z}}_{A,1} \\ \text{s.t} \quad & \mathbf{z}_0 \in \mathcal{C}, \\ & \hat{\mathbf{z}}_{A,1} = W_1 \mathbf{z}_0 + \mathbf{b}_0,\end{aligned} \tag{5.5.8}$$

with $\mathbf{c}^T = \nabla_{\hat{\mathbf{z}}_{A,1}} \mathcal{L}(\hat{\mathbf{z}}, \boldsymbol{\rho})$. Rewriting the problem as a linear function of $\mathbf{z}_0$ only, problem (5.5.8) is simply equivalent to minimising $\mathbf{c}^T W_1 \mathbf{z}_0$ over $\mathcal{C}$. We initially assumed that optimisation of linear function over $\mathcal{C}$ was efficient. We now give examples of $\mathcal{C}$ where (5.5.8) can be solved efficiently.

Bounded Input Domain If $\mathcal{C}$ is defined by a set of lower bounds $\mathbf{l}_0$ and upper bounds $\mathbf{u}_0$ (as in the case of $\mathcal{L}_\infty$ adversarial examples), optimisation will simply amount to choosing either the lower or upper bound depending on the sign of the linear function. The optimal solution is:

$$\begin{aligned} \mathbf{x}_0 &= \mathbb{1}_{\mathbf{c}^T W_1 > 0} \cdot \mathbf{l}_0 + \mathbb{1}_{\mathbf{c}^T W_1 \leq 0} \cdot \mathbf{u}_0, \\ \hat{\mathbf{x}}_{A,1} &= W_1 \mathbf{x}_0 + \mathbf{b}_1. \end{aligned} \tag{5.5.9}$$

$\mathcal{L}_2$ **Balls** If $\mathcal{C}$ is defined by an $\mathcal{L}_2$ ball of radius ϵ around a point $\bar{\mathbf{x}}$ ($\|\mathbf{x}_0 - \bar{\mathbf{x}}\|_2 \leq \epsilon$), optimisation amounts to choosing the point on the boundary of the ball such that the vector from the center to it is opposed to the cost function. Formally, the optimal solution is given by:

$$\begin{aligned} \hat{\mathbf{x}}_{A,1} &= W_1 \mathbf{x}_0 + \mathbf{b}_1, \\ \mathbf{x}_0 &= \bar{\mathbf{x}} - (\sqrt{\epsilon}/\|\mathbf{c}\|_2)\mathbf{c}. \end{aligned} \tag{5.5.10}$$

$\mathcal{P}_k$ **Subproblems:**

For the component of the conditional gradient corresponding to the $\mathcal{P}_k$ subproblem, we need to solve:

$$\begin{aligned} [\hat{\mathbf{x}}_{B,k}, \hat{\mathbf{x}}_{A,k+1}] = \underset{\hat{\mathbf{z}}_{B,k}, \hat{\mathbf{z}}_{A,k+1}}{\text{argmin}} \quad & \mathbf{c}_k^T \hat{\mathbf{z}}_{B,k} + \mathbf{c}_{k+1}^T \hat{\mathbf{z}}_{A,k+1} \\ \text{s.t} \quad & \mathbf{l}_k \leq \hat{\mathbf{z}}_{B,k} \leq \mathbf{u}_k \\ & \texttt{cvx_hull}_\sigma \left(\hat{\mathbf{z}}_{B,k}, \mathbf{z}_k, \mathbf{l}_k, \mathbf{u}_k\right) \\ & \hat{\mathbf{z}}_{A,k+1} = W_{k+1}\mathbf{z}_k + \mathbf{b}_{k+1}, \end{aligned} \tag{5.5.11}$$

where we will have $\mathbf{c}_k^T = \nabla_{\hat{\mathbf{z}}_{B,k}} \mathcal{L}\left(\hat{\mathbf{z}}, \boldsymbol{\rho}\right)$ and $\mathbf{c}_{k+1}^T = \nabla_{\hat{\mathbf{z}}_{A,k+1}} \mathcal{L}\left(\hat{\mathbf{z}}, \boldsymbol{\rho}\right)$ for $k \in [1 \dots n-2]$.

In the case where the activation function σ is the ReLU and `cvx_hull` is given by Equation (5.4.3), we can find a closed form solution. Using the last equality of the problem, we can rewrite the objective function as $\mathbf{c}_k^T \hat{\mathbf{z}}_{B,k} + \mathbf{c}_{k+1}^T W_{k+1} \mathbf{z}_k$. If the ReLU is ambiguous, the shape of the convex hull is represented in Figure 5.1. If the sign of $\mathbf{c}_{k+1}^T W_{k+1}$ is positive, minimizing (5.5.11) corresponds to having $\mathbf{z}_k$ at its lower bound $\max\left(\hat{\mathbf{z}}_{B,k}, 0\right)$. If on the contrary the sign is negative, $\mathbf{z}_k$ must be equal to its upper bound $\frac{\mathbf{u}_k}{\mathbf{u}_k - \mathbf{l}_k}\left(\hat{\mathbf{z}}_{B,k} - \mathbf{l}_k\right)$. We can therefore rewrite the subproblem (5.5.11) in the case of ambiguous ReLUs as:

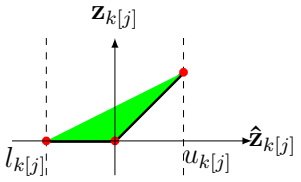


Figure 5.1: Feasible domain of the convex hull for an ambiguous ReLU. Red circles indicate the vertices of the feasible region.

$$\underset{\hat{\mathbf{z}}_{B,k} \in [\mathbf{l}_k, \mathbf{u}_k]}{\text{argmin}} \quad \left(\mathbf{c}_k^T + \left[\mathbf{c}_{k+1}^T W_{k+1}\right]_- \frac{\mathbf{u}_k}{\mathbf{u}_k - \mathbf{l}_k}\right) \hat{\mathbf{z}}_{B,k} + \left[\mathbf{c}_{k+1}^T W_{k+1}\right]_+ \max\left(\hat{\mathbf{z}}_{B,k}, 0\right), \tag{5.5.12}$$

where $[A]_-$ corresponds to the negative values of A ($[A]_- = \min(A, 0)$) and $[A]_+$ corresponds to the positive value of A ($[A]_+ = \max(A, 0)$). The objective function and the constraints all decompose completely over the coordinates, so all problems can be solved independently. For each dimension, the problem is a convex, piecewise linear function, which means that the optimal point will be a vertex. The possible vertices are $(\mathbf{l}_k[i], 0)$, $(0, 0)$, and $(\mathbf{u}_k[i], \mathbf{u}_k[i])$. We can evaluate the objective function at these three points and keep the one with the smallest value as the conditional gradient.

For the ReLU for which we can infer the phase ($\mathbf{l}_k \geq 0$ or $\mathbf{u}_k \leq 0$), the `cvx_hull` constraint is a simple linear equality constraint. For those coordinates, the problem is analogous to the one solved by Equation (5.5.9), with the linear function being minimised over the $\hat{\mathbf{z}}_{B,k}$ box bounds being $\mathbf{c}_k^T \hat{\mathbf{z}}_{B,k}$ in the case of blocking ReLUs or $\left(\mathbf{c}_k + \mathbf{c}_{k+1}^T W_{k+1}\right) \hat{\mathbf{z}}_{B,k}$ in the case of passing ReLUs.

We have described how to solve problem (5.5.11) by simply evaluating linear functions at given points. The matrix operations involved correspond exactly to standard operations of the neural network's layers. Computing $\hat{\mathbf{z}}_{A,k+1} = W_{k+1}\mathbf{z}_k + \mathbf{b}_{k+1}$ is exactly the forward pass of the network and computing $\hat{\mathbf{c}}_{k+1}^T W_{k+1}$ is analogous to the backpropagation of gradients. We can therefore take advantage of existing deep learning frameworks to gain access to efficient implementations. When dealing with convolutional layers, we can employ specialised implementations rather than building the equivalent W_k matrix ,which would contain a lot of redundancy.

We described here the solving process in the context of ReLU activation functions, but this can be generalised to non piecewise linear activation functions. For example, subsection 5.8.2 describes the solution for sigmoid activation functions.

Optimal Step Size Computation

Having obtained the conditional gradients for our problem, we need to select a step size. Computing the optimal step size amounts to solving a one dimensional quadratic problem:

$$\min_{\gamma \in [0,1]} \mathcal{L}(\gamma \hat{\mathbf{x}} + (1-\gamma)\hat{\mathbf{z}}, \boldsymbol{\rho}). \tag{5.5.13}$$

Computing the gradient with regards to γ and setting it to 0 will return the optimal step size. This gives us the following formula:

$$\gamma_{\text{opt}} = \texttt{clamp}_{[0,1]}\left(-\frac{(\mathbf{x}_{A,n} - \mathbf{z}_{A,n}) + \sum_k \boldsymbol{\rho}_k^T \left(\hat{\mathbf{x}}_{B,k} - \hat{\mathbf{z}}_{B,k} - \hat{\mathbf{x}}_{A,k} + \hat{\mathbf{z}}_{A,k}\right)}{\sum_k \frac{1}{\eta_k} \|\hat{\mathbf{x}}_{B,k} - \hat{\mathbf{z}}_{B,k} - \hat{\mathbf{x}}_{A,k} + \hat{\mathbf{z}}_{A,k}\|^2}\right). \tag{5.5.14}$$

We therefore have no need to choose the step size. In practice, we still have to choose the strength of the proximal term η_k. This parameter is easier to set: given that we get the optimal solution, and that 0 is a valid choice for γ, each iteration of conditional gradient will improve the value of the inner problem. If we had used a subgradient method, we would not have any guarantee of improvement.

5.5.4 Alternative Method

Proximal maximisation is not the only approach to solve problem (5.5.5). In their algorithm to obtain neural networks bounds, Dvijotham et al. [53] advocate for the use of subgradient descent. We will use this method as one of our baselines, so let us briefly describe how it will be implemented.

At a given point $\boldsymbol{\rho}$, obtaining the subgradient requires us to know the values of $\hat{\mathbf{z}}_A$ and $\hat{\mathbf{z}}_B$ for which the inner minimisation is achieved. This computation corresponds exactly to the one we do to obtain the conditional gradient of $\mathcal{L}\left(\hat{\mathbf{z}}, \boldsymbol{\rho}\right)$. The equivalence of conditional gradient and subgradient is not particular to our specific problem. A more general description can be found in the work of Bach [12].

Based on the identified values of $\hat{\mathbf{z}}_A$ and $\hat{\mathbf{z}}_B$, we can compute the subgradient $\nabla_{\boldsymbol{\rho}} q = (\hat{\mathbf{z}}_B - \hat{\mathbf{z}}_A)$. The updates to $\boldsymbol{\rho}$ are then the usual subgradient descent updates:

$$\boldsymbol{\rho}^{t+1} = \boldsymbol{\rho}^t + \alpha^t \nabla_{\boldsymbol{\rho}} q(\boldsymbol{\rho}^t), \tag{5.5.15}$$

where α^t corresponds to a step size schedule that needs to be provided. It is also possible to use any variants of gradient descent, such as Adam [96].

5.6 Experiments

5.6.1 Implementation Details

One of the benefits of our algorithm (and of the subgradient baseline) is that they are easily parallelizable. To compute the upper and lower bounds for all the neurons of a layer, there is no dependency between the different problems, so we are free to solve them all simultaneously. As explained in Section 5.5.3, the crucial part of the algorithm, for both the proximal method and the subgradient descent, works by applying linear algebra operations equivalent to the ones employed during the forward and backward passes of neural networks. Those operations are applied to values corresponding to the current solution. The weights used are the same for all problems, so we can execute those operations in a batched mode. We take the building blocks provided by deep learning frameworks, and apply them to batches of solutions (one per problem), in the same way that normal training applies them to a batch of samples. This makes it possible for us to leverage the engineering efforts done to enable fast training and evaluation of neural networks, and easily take advantage of GPU accelerations. The implementation used in our experiment is based on Pytorch [131] and will be made publicly available.

5.6.2 Experimental setting

In order to evaluate the performance of the proposed methods, we perform a comparison on the quality and speed of bounds obtained by different methods. We report results on various settings, using different networks trained with different methodologies on the CIFAR-10 dataset. The networks that we consider have all the same architecture, which corresponds to the small model used by Wong and Kolter [171] and whose structure is given in subsection 5.8.1.

For each network, we compute an upper bound on the difference between the ground truth logit and all the other logits, under an allowed perturbation ϵ in infinity norm of the inputs. All networks are trained to achieve robustness against perturbation of a size up to $\epsilon = 8/255$. In order to control for the possible effects that a specific training method could have on the usefulness of a method, we evaluate on different networks. **WK** is a network trained with the method of Wong and Kolter [171]; **Adversarial** is a network trained using adversarial training [116], based on an attacker using 50 steps of Projected gradient descent attacks to obtain the samples. The **Baseline** network is trained using standard stochastic gradient descent and cross entropy, with no robustness objective.

As methods for computing bounds, we compare several methods:

- **WK** uses the method of Wong and Kolter [171] to compute bounds, based on their publicly released Pytorch implementation. This is equivalent to a specific choice of $\boldsymbol{\rho}$.
- **Proximal** corresponds to our method, based on proximal maximisation. We limit the total number of inner iterations for each problem and report results for different values. Adjusting this parameter would lead to a different tradeoff between quality of the bounds obtained versus the speed at which the bounds are obtained. Unless otherwise specified, we use $\eta_k = 10^3$ for all layers.
- **Subgradient** corresponds to subgradient descent over problem (5.5.5), as described in Section 5.5.4. We use the Adam [96] updates rules and decrease the step size linearly from $\alpha_k = 10^{-3}$ to $\alpha_k = 10^{-6}$, similarly to the experiments of Dvijotham et al. [53]. We experimented with other step size schedules, like constant step size or $\frac{1}{t}$ schedules, which all performed worse.
- **Gurobi** is our gold standard method, employing the commercial black box solver Gurobi. All the problems are solved to optimality, providing us with the best result that our solver could achieve.

We chose to not include computation based on **Naive bounds** in our analysis. It has been described as the Box domain [121] or as Interval Bound Propagation [71], and corresponds to setting all the dual variables to 0 in our dual formulation. While Gowal et al. [71] showed that it could produce state of the art results when used as a training signal, our experiments showed it to produce bounds significantly worse than other methods for

networks that were not trained with it. All methods with the exception of Gurobi were ran on GPU. While it may seem to lead to an unfair comparison, the flexibility and parallelism of the methods are a big part of their advantages over off-the-shelf solvers.

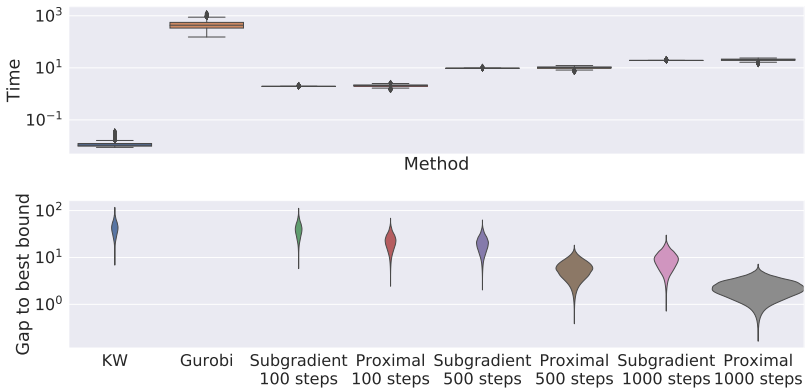
5.6.3 Results

We use samples from the CIFAR-10 test set and compute robustness margin for all non ground truth classes. We measure the time to compute all the bounds (including all the necessary intermediate bounds), and report the optimality gap compared to the best achieved solution.

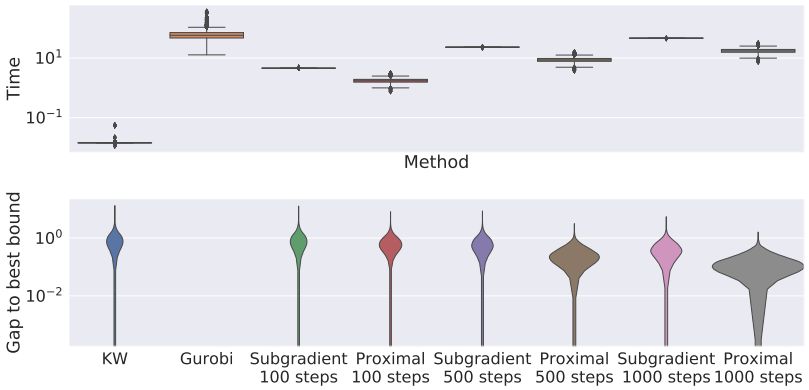
Figure 5.2a shows how the different algorithms behave when the network they are applied to have not been specifically trained to be easy to bound and therefore generates difficult optimisation problems. The **WK** method of Wong and Kolter [171] performs a single pass over the network per optimisation problem, which allows it to be extremely fast, but this comes at the cost of generating looser bounds. At the complete opposite end of the spectrum, **Gurobi** is extremely slow but provides the best achievable bounds. Iterative methods like **Proximal** and **Subgradient** allow the user to choose the tradeoff between tightness and speed. The improved quality of the bounds as compared to the non-iterative method **WK** shows that there is benefits in actually solving the best relaxation rather than simply evaluating looser bounds. By using more computation time, some samples can be proven robust by iterative methods, even when **WK** would not be able to prove robustness.

For a similar number of iterations, **Proximal** outperforms **Subgradient** in the quality of the bounds. Even when giving a computational budget twice as large to the subgradient method, **Proximal** still manages to achieve better results.This difference in speed of convergence is also clearly visible in Figure 5.3. For this optimisation problem, we have used the same intermediate bounds for both methods, in order for them to have the same optimal solution so as to allow proper comparison. In a real system, looser intermediate bounds obtained by the **Subgradient** method in earlier layers would lead to a looser relaxation and a worse optimal solution, making **Proximal** dominate even further. Figure 5.4 shows a bound-by-bound comparison of these results rather than just in the distributional sense and clearly demonstrates the advantages of the proximal methods.

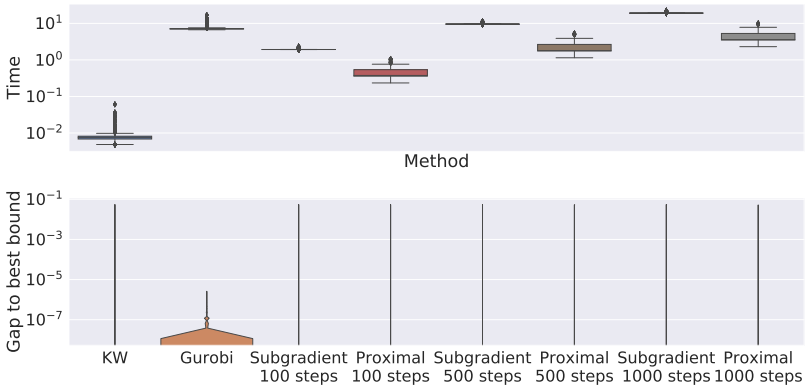
While we can definitely see the benefits of our method on normally and adversarially trained networks, it does not provide benefits on networks trained against a specific bound. The graph of Figure 5.2c shows that increasing computation does not yield improvements in those cases. This is to be expected: the network has specifically been trained to generate good bounds with a single pass. As a result, simple methods get very close to the global optimum and there is no margin for improvement. The problems of proving bounds on those networks are too simple to benefit from better optimisation. In those cases, running the iterative algorithms with a smaller number of iterations would be sufficient.



(a) Results on the **Baseline** network.



(b) Results on the **Adversarial** network. Compared to Figure 5.2a, the difference between methods are not as large.

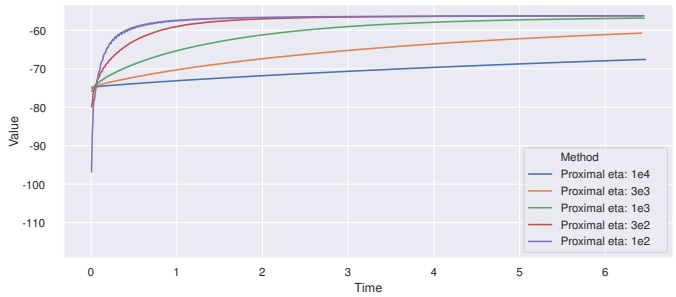


(c) Results on the **WK** network. Almost all methods return the same bounds, with only Gurobi managing to improve slightly.

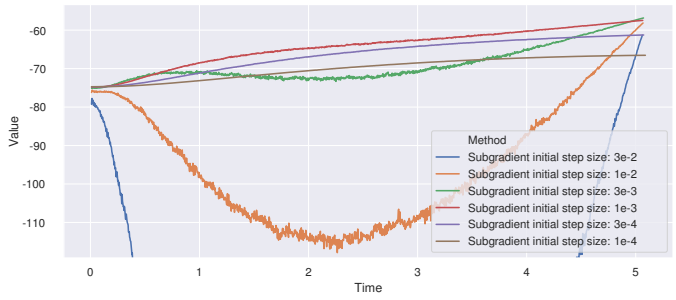
Figure 5.2: *Comparison of the distribution of runtime (top) and gap to optimality (bottom) between different methods. In both cases, lower is better. The width at a given value represents the proportion of problems for which this is the result. Gurobi always return the optimal solution so does not appear on the optimality gap plot, with the exception of floating point errors for the* **WK** *network. It always has the highest runtime.* **WK** *is significantly faster than all other methods but provides relatively loose bounds.* **Proximal** *and* **Subgradient** *methods provide a trade-off between computation time and quality of the bounds, with the proximal methods providing better bounds than the subgradient one.*

5.6.4 Sensitivity to hyperparameters

Figure 5.3 compares the robustness of the dual optimisers to their tuning hyperparameters. After identifying the best performing parameters, we ran the same optimisation problem (corresponding to the margins between incorrect output logits and ground truth logit at the last layer of the **Baseline** network) with several hyperparameters. In addition to obtaining worse results on the best run, the quality of the optimisation degrades significantly faster for hyperparameters beyond the optimal ones for the **Subgradient** method, while the **Proximal** method shows itself to be more robust.

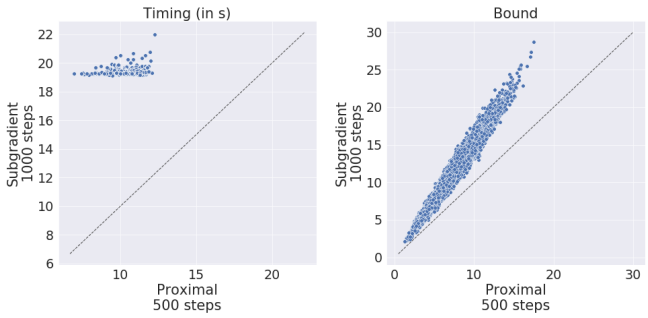


(a) Proximal methods

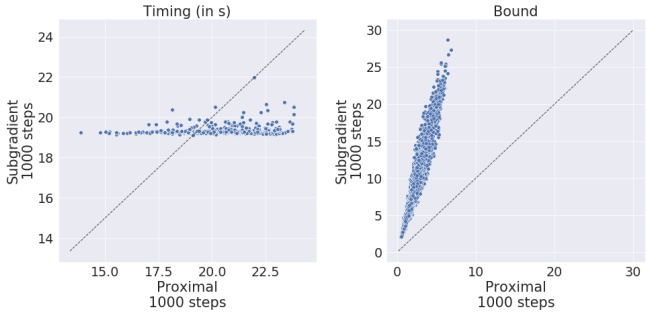


(b) Subgradient methods

Figure 5.3: *Evolution of the computed lower bounds for the iterative methods, for a range of hyperparameters (higher score is better).* ***Proximal*** *methods outperforms the* ***Subgradient*** *method, even though we ran the latter for a larger number of iterations (1500 vs. 1000). For proximal methods, several choices of hyperparameters result in the same high quality solution. In contrast, for subgradient methods, the choice of hyperparameters has a much larger impact on the optimisation. Note that the surprising shape of the subgradient optimization is due to the decrease in step-size (three orders of magnitude linearly in total over the iterations). Letting them run for more iterations at the final step size doesn't improve bounds.*



(a) 500 steps of proximal vs. 1000 steps of subgradient



(b) 1000 steps of proximal vs. 1000 steps of subgradient

Figure 5.4: *Comparison run by run of the optimisation. Each datapoint corresponds to a CIFAR sample, where we report the bound and time achieved by both methods. The dotted line correspond to the equality and in both graphs, lower is better along both axis. The result is achieved on the* ***Baseline*** *network. Note that by a "step", we mean a subgradient / conditional gradient update, therefore making the comparisons valid.*

5.7 Discussion

We have presented a novel algorithm to compute bounds over the activation of neural networks. It provides significant benefits compared to off-the-shelf solvers and allows us to obtain better bounds than looser relaxations. It would be interesting to see if similar decomposition methods could be used to make tighter but more expensive relaxations computationally feasible. Improving the quality of bounds may allow the training of robust networks to avoid over-regularisation.

5.8 Additional Notes

5.8.1 Network Architecture

Conv2D(channels=16, kernel_size=4, stride=2) ReLU
Conv2D(channels=32, kernel_size=4, stride=2) ReLU
Linear(channels=100) ReLU
Linear(channels=10)

Table 5.1: Network architecture

5.8.2 Sigmoid Activation function

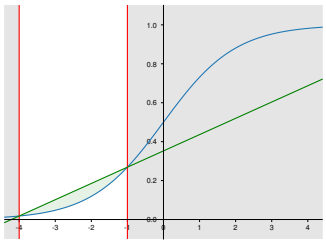
This section describes the computation highlighted in section 5.5.3 in the context where the activation function $\sigma(x)$ is the sigmoid function:

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{5.8.1}$$

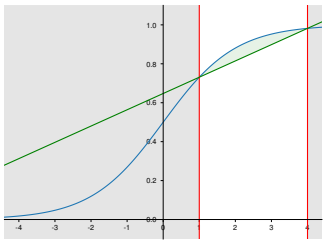
A similar methodology to the ones described in this section could be used to adapt the method to work with other activation function such as hyperbolic tangent, but we won't discuss it.

We will start with a reminders about some properties of the sigmoid activation function. It takes values between 0 and 1, with $\sigma(0) = 0.5$. We can easily compute its derivatives:

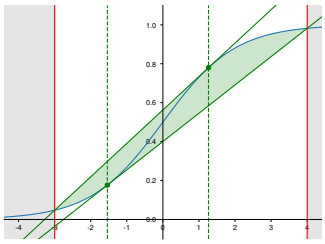
$$\begin{aligned} \sigma'(x) &= \sigma(x) \times (1 - \sigma(x)) \\ \sigma''(x) &= \sigma(x) \times (1 - \sigma(x)) \times (1 - 2\sigma(x)) \end{aligned} \tag{5.8.2}$$



(a) The upper bound is in Case 1, while the lower bound is in the special case of Case 2 with only one piece.



(b) The upper bound is in the special case of Case 2 with only one piece, while the lower bound is in Case 1.



(c) Both upper and lower bounds are in Case 2 with the two pieces visible.

Figure 5.5: Convex hull of the Sigmoid activation function for different input bound configurations.

If we limit the domain of study to the negative inputs ($[-\infty, 0]$), then the function $x \mapsto \sigma(x)$ is a convex function, as can be seen by looking at the sign of the second derivative over that domain. Similarly, if we limit the domain of study to the positive inputs ($[0, \infty]$), the function is concave.

Convex hull computation

In the context of ReLU, the convex hull of the activation function is given by Equation(5.4.3), as introduced by Ehlers [58]. We will now derive it for sigmoid functions. Upper and lower bounds will be dealt in the same way, so our description will only focus on how to obtain the concave upper bound, limiting the activation function convex hull by above. The computation to derive the convex lower bound is equivalent.

Depending on the range of inputs over which the convex hull is taken, the form of the concave upper bound will change. We distinguish two cases.

Case 1: $\sigma'(\mathbf{u}_k) \geq \frac{\sigma(\mathbf{u}_k)-\sigma(\mathbf{l}_k)}{\mathbf{u}_k-\mathbf{l}_k}$**.** We will prove that in this case, the upper bound will be the line passing through the points $(\mathbf{l}_k, \sigma(\mathbf{l}_k))$ and $(\mathbf{u}_k, \sigma(\mathbf{l}_k))$. The equation of it is given by:

$$\phi_{\mathbf{u}_k,\mathbf{l}_k}(\mathbf{x}) = \frac{\sigma(\mathbf{u}_k) - \sigma(\mathbf{l}_k)}{\mathbf{u}_k - \mathbf{l}_k}(\mathbf{x} - \mathbf{l}_k) + \sigma(\mathbf{l}_k) \tag{5.8.3}$$

Consider the function $d(\mathbf{x}) = \phi_{\mathbf{u}_k,\mathbf{l}_k}(\mathbf{x}) - \sigma(\mathbf{x})$. To show that $\phi_{\mathbf{u}_k,\mathbf{l}_k}$ is a valid upper bound, we need to prove that $\forall \mathbf{x} \in [\mathbf{l}_k, \mathbf{u}_k]$, $d(\mathbf{x}) \geq 0$. We know that $d(\mathbf{l}_k) = 0$ and $d(\mathbf{u}_k) = 0$, and that d is a continuous function. Its derivative is given by:

$$d'(\mathbf{x}) = \frac{\sigma(\mathbf{u}_k) - \sigma(\mathbf{l}_k)}{\mathbf{u}_k - \mathbf{l}_k} - \sigma(\mathbf{x})(1 - \sigma(\mathbf{x})). \tag{5.8.4}$$

To find the roots of d', we can solve $d'(\mathbf{x}) = 0$ for the value of $\sigma(\mathbf{x})$ and then use the logit function to recover the value of $\mathbf{x}$. In that case, this is only a second order polynomial, so it can admit at most two roots.

We know that $\lim_{\mathbf{x}\to\infty} d'(\mathbf{x}) \geq 0$, and our hypothesis tells us that $d'(\mathbf{u}_k) \leq 0$. This means that at least one of the root lies necessarily beyond $\mathbf{u}_k$ and therefore, the derivative of d change signs at most once on the $[\mathbf{l}_k, \mathbf{u}_k]$ interval. If it never changes sign, it is monotonous. Given that the values taken at both extreme points are the same, d being monotonous would imply that d is constant, which is impossible. We therefore conclude that this means that the derivative change its sign exactly once on the interval, and is therefore unimodal. As we know that $d'(\mathbf{u}_k) \leq 0$, this indicates that d is first increasing and then decreasing. As both extreme points have a value of zero, this means that $\forall \mathbf{x} \in [\mathbf{l}_k, \mathbf{u}_k],\ d(x) \geq 0$.

From this result, we deduce that $\phi_{\mathbf{u}_k,\mathbf{l}_k}$ is a valid upper bound of σ. As a linear function, it is concave by definition. Given that it constitutes the line between two points on the curve, all of its points necessarily belong to the convex hull. Therefore, it is not possible for a concave upper bound of the activation function to have lower values. This means that $\phi_{\mathbf{u}_k,\mathbf{l}_k}$ defines the upper bounding part of the convex hull of the activation function.

Case 2: $\sigma'(\mathbf{u}_k) \leq \frac{\sigma(\mathbf{u}_k)-\sigma(\mathbf{l}_k)}{\mathbf{u}_k-\mathbf{l}_k}$**.** In this case, we will have to decompose the upper bound into two parts, defined as follows:

$$\phi_{\mathbf{u}_k,\mathbf{l}_k}(\mathbf{x}) \equiv \begin{cases} \dfrac{\sigma(\mathbf{t}_k)-\sigma(\mathbf{l}_k)}{\mathbf{t}_k-\mathbf{l}_k}(\mathbf{x}-\mathbf{l}_k)+\sigma(\mathbf{l}_k) & \text{if } \mathbf{x}\in[\mathbf{l}_k,\mathbf{t}_k] \quad (5.8.5a) \\ \sigma(\mathbf{x}) & \text{if } \mathbf{x}\in[\mathbf{t}_k,\mathbf{u}_k], \quad (5.8.5b) \end{cases}$$

where $\mathbf{t}_k$ is defined as the point such that $\sigma'(\mathbf{t}_k) = \frac{\sigma(\mathbf{t}_k)-\sigma(\mathbf{l}_k)}{\mathbf{t}_k-\mathbf{l}_k}$ and $\mathbf{t}_k > 0$. The value of $\mathbf{t}_k$ can be computed by solving the equation $\sigma(\mathbf{t}_k)(1-\sigma(\mathbf{t}_k)) = \frac{\sigma(\mathbf{t}_k)-\sigma(\mathbf{l}_k)}{\mathbf{t}_k-\mathbf{l}_k}$, which can be done using the Newton-Raphson method or a binary search. Note that this needs to be done only when defining the problem, and not at every iteration of the solver. In addition, the value of $\mathbf{t}_k$ is dependant only on $\mathbf{l}_k$ so it's possible to start by building a table of the results at the desired accuracy and cache them.

Evaluating both pieces of the function of Equation(5.8.5) at $\mathbf{t}_k$ show that $\phi_{\mathbf{u}_k,\mathbf{l}_k}$ is continuous. Both pieces are concave (for $\mathbf{x} \geq \mathbf{t}_k \geq 0$, σ is concave) and they share a supergradient (the linear function of slope $\frac{\sigma(\mathbf{t}_k)-\sigma(\mathbf{l}_k)}{\mathbf{t}_k-\mathbf{l}_k}$) in $\mathbf{t}_k$, so $\phi_{\mathbf{u}_k,\mathbf{l}_k}$ is a concave function. The proof we did for **Case 1** can be duplicated to show that the linear component is the best concave upper bound that can be achieved over the interval $[\mathbf{l}_k, \mathbf{t}_k]$. On the interval $[\mathbf{t}_k, \mathbf{u}_k]$, $\phi_{\mathbf{u}_k,\mathbf{l}_k}$ is equal to the activation function, so it is also an upper bound which can't be improved upon. Therefore, $\phi_{\mathbf{u}_k,\mathbf{l}_k}$ is the upper bounding part of the convex hull of the activation function.

Note that a special case of this happens when $\mathbf{l}_k \geq \mathbf{t}_k$. In which case, $\phi_{\mathbf{u}_k,\mathbf{l}_k}$ consists of only Equation (5.8.5b).

All cases are illustrated in Figure 5.5. Case 1 is shown in 5.5a, where the upper bound contains only the linear upper bound. Case 2 with both segments is visible in Figure5.5c, with the cutoff points tb_k highlighted by a green dot, and the special case with $\mathbf{l}_k \geq \mathbf{t}_k$ is demonstrated in Figure 5.5b.

Solving the $\mathcal{P}_k$ subproblems over sigmoid activation

As a reminder, the problem that needs to be solved is the following:

$$\begin{aligned}[\hat{\mathbf{x}}_{B,k}, \hat{\mathbf{x}}_{A,k+1}] = \underset{\hat{\mathbf{z}}_{B,k},\hat{\mathbf{z}}_{A,k+1}}{\operatorname{argmin}} \quad & \mathbf{c}_k^T \hat{\mathbf{z}}_{B,k} + \mathbf{c}_{k+1}^T \hat{\mathbf{z}}_{A,k+1} \\ \text{s.t} \quad & \mathbf{l}_k \leq \hat{\mathbf{z}}_{B,k} \leq \mathbf{u}_k \\ & \texttt{cvx_hull}_\sigma\left(\hat{\mathbf{z}}_{B,k}, \mathbf{z}_k, \mathbf{l}_k, \mathbf{u}_k\right) \\ & \hat{\mathbf{z}}_{A,k+1} = W_{k+1}\mathbf{z}_k + \mathbf{b}_{k+1},\end{aligned} \tag{5.8.6}$$

where `cvx_hull` is defined either by equations (5.8.3) or (5.8.5). In this case, we will still be able to compute a closed form solution.

We will denote $\phi_{\mathbf{u}_k,\mathbf{l}_k}$ and $\psi_{\mathbf{u}_k,\mathbf{l}_k}$ the upper and lower bound functions defining the convex hull of the sigmoid function, which can be obtained as described in the previous subsection. If we use the last equality constraint of Problem (5.8.6) to replace the $\hat{\mathbf{z}}_{A,k+1}$ term in the objective function, we obtain $\mathbf{c}_k^T \hat{\mathbf{z}}_{B,k} + \mathbf{c}_{k+1}^T W_{k+1}\mathbf{z}_k$. Depending on the sign of $\mathbf{c}_{k+1}^T W_{k+1}$, $\mathbf{z}_k$ will either take the value $\phi_{\mathbf{u}_k,\mathbf{l}_k}$ or $\psi_{\mathbf{u}_k,\mathbf{l}_k}$, resulting in the following problem:

$$\begin{aligned}\min_{\hat{\mathbf{z}}_{B,k}} \quad & \mathbf{c}_k^T \hat{\mathbf{z}}_{B,k} + \left[\mathbf{c}_{k+1}^T W_{k+1}\right]_- \phi_{\mathbf{u}_k,\mathbf{l}_k}\left(\hat{\mathbf{z}}_{B,k}\right) + \left[\mathbf{c}_{k+1}^T W_{k+1}\right]_+ \psi_{\mathbf{u}_k,\mathbf{l}_k}\left(\hat{\mathbf{z}}_{B,k}\right) \\ \text{s.t} \quad & \mathbf{l}_k \leq \hat{\mathbf{z}}_{B,k} \leq \mathbf{u}_k.\end{aligned} \tag{5.8.7}$$

To solve this problem, several observations can be made: First of all, at that point, the optimisation decomposes completely over the components of $\hat{\mathbf{z}}_{B,k}$ so all problems can be solved independently from each other. The second observation is that to solve this problem, we can decompose the minimisation over the whole range $[\mathbf{l}_k, \mathbf{u}_k]$ into a set of minimisation over the separate pieces, and then returning the minimum corresponding to the piece producing the lowest value.

The minimisation over the pieces can be of two forms. Both bounds can be linear (such as between the green dotted lines in Figure 5.5c), in which case the problem is easy to solve by looking at the signs of the coefficient of the objective function. The other option

is that one of the bound will be equal to the activation function (such as in Figures 5.5a or 5.5b, or in the outer sections of Figure 5.5c), leaving us with a problem of the form:

$$\begin{aligned} \min_{\hat{\mathbf{z}}_{B,k}} \quad & \mathbf{c}_{\text{lin}}^T \hat{\mathbf{z}}_{B,k} + \mathbf{c}_\sigma^T \sigma\left(\hat{\mathbf{z}}_{B,k}\right) \\ & \mathbf{l} \leq \hat{\mathbf{z}}_{B,k} \leq \mathbf{u}, \end{aligned} \tag{5.8.8}$$

where $\mathbf{l}$, $\mathbf{u}$, $\mathbf{c}_{\text{lin}}$ and $\mathbf{c}_\sigma$ will depend on what part of the problem we are trying to solve.

This is a convex problem so the value will be reached either at the extremum of the considered domain ($\mathbf{l}$ or $\mathbf{u}$), or it will be reached at the points where the derivative of the objective functions cancels. This corresponds to the roots of $\mathbf{c}_{\text{lin}} + \mathbf{c}_\sigma^T \sigma\left(\hat{\mathbf{z}}_{B,k}\right)\left(1 - \sigma\left(\hat{\mathbf{z}}_{B,k}\right)\right)$. Provided that $1 + \frac{4\mathbf{c}_{\text{lin}}}{\mathbf{c}_\sigma} \geq 0$, the possible roots will be given by $\sigma^{-1}\left(\frac{1 \pm \sqrt{1 + \frac{4\mathbf{c}_{\text{lin}}}{\mathbf{c}_\sigma}}}{2}\right)$, with σ^{-1} being the logit function, the inverse of the sigmoid function $\left(\sigma^{-1}(\mathbf{x}) = \log\left(\frac{\mathbf{x}}{1-\mathbf{x}}\right)\right)$. To solve problem (5.8.8), we evaluate its objective function at the extreme points ($\mathbf{l}$ and $\mathbf{u}$) and at those roots if they are in the feasible domain, and return the point corresponding to the minimum score achieved. With this method, we can solve the $\mathcal{P}_k$ subproblems even when the activation function is a sigmoid.

5.8.3 Algorithm summary

We provide a high level summary of the entire optimization method in Algorithm 2.

We decompose problem (5.4.2) into several subset of constraints and enforce their consistency by introducing dual variables $\boldsymbol{\rho}$, which we are going to optimize, resulting in problem (5.5.5). Our initial solution (line 2) is given by the algorithm of Wong and Kolter [171]. The details can be found in Appendix 5.8.5.

This initial solution is then improved by using the Augmented Lagrangian method. We alternate updates to the dual variables (applying the updates from (5.5.6); line 5) and to the primal variables (update of (5.5.7); from line 6 to 10). The inner minimization problem is solved only approximately, using the Frank-Wolfe algorithm (line 9), where we have closed form solution for both the conditional gradient (line 7) and the optimal step-size (line 8).

Algorithm 2 Bound optimization

function COMPUTE_BOUNDS($\{W_k, b_k, l_k, u_k\}_{k=1..n}$)
  Initialise dual variables $\boldsymbol{\rho}$ using the algo of Wong and Kolter [171]
  Initialise primal variables $\hat{\mathbf{z}}$ at the conditional gradient of $\boldsymbol{\rho}$
  for `nb_outer_iterations` **do**
    Update dual variable using Equation (5.5.6)
    for `nb_inner_iterations` **do**
      Compute conditional gradient $[\hat{\mathbf{x}}]$ using Section 5.5.3
      Compute optimal step size γ_{opt} using (5.5.14)
      Update subproblem variables:
        $\hat{\mathbf{z}} = \gamma_{\text{opt}}\hat{\mathbf{x}} + (1 - \gamma_{\text{opt}})\hat{\mathbf{z}}$
    end for
  end for
  return $q(\boldsymbol{\rho})$
end function

5.8.4 Comparison between the duals

We will show the relation between the decomposition done by Dvijotham et al. [53] and the one proposed in this chapter. We will show that, in the context of ReLU activation functions, from any dual solution of their dual providing a bound, our dual provides a better bound. This is the justification for using subgradient on our dual rather than on theirs as a baseline in the experiment section: it is guaranteed to produce better results.

We start by reproducing the formulation that Dvijotham et al. [53] uses, which we slightly modify in order to have a notation more comparable to ours[1]. With our convention, doing the decomposition to obtain Problem (6) in [53] would give:

$$\begin{aligned}
\min_{\mathbf{z},\hat{\mathbf{z}}} \quad & W_n\mathbf{z}_{n-1} + \mathbf{b}_n + \sum_{k=1}^{n-1} \mu_k^T \left(\hat{\mathbf{z}}_k - W_k\mathbf{z}_{k-1} - \mathbf{b}_k\right) + \sum_{k=1}^{n-1} \lambda_k^T \left(\mathbf{z}_k - \sigma(\hat{\mathbf{z}}_k)\right) \\
\text{such that} \quad & \mathbf{l}_k \leq \hat{\mathbf{z}}_k \leq \mathbf{u}_k \qquad \text{for } k \in [1..n-1] \\
& \sigma(\mathbf{l}_k) \leq \mathbf{z}_k \leq \sigma(\mathbf{u}_k) \qquad \text{for } k \in [1..n-1] \\
& \mathbf{z}_0 \in \mathcal{C}
\end{aligned} \tag{5.8.9}$$

Decomposing it, in the same way that Dvijotham et al. [53] do it in order to obtain

[1] Our activation function σ is denoted h in their paper. The equivalent of $\mathbf{x}$ in their paper is $\mathbf{z}$ in ours, while the equivalent of their $\mathbf{z}$ is $\hat{\mathbf{z}}$. Also note that their paper use the computation of upper bounds as examples while ours use the computation of lower bounds.

their equation (7), we obtain

$$\begin{aligned}\mathbf{b}_n - \sum_{k=1}^{n-1} \boldsymbol{\mu}_k^T \mathbf{b}_k + \sum_{k=1}^{n-1} \min_{\hat{\mathbf{z}}_k \in [\mathbf{l}_k, \mathbf{u}_k]} \left(\boldsymbol{\mu}_k^T \hat{\mathbf{z}}_k - \boldsymbol{\lambda}_k^T \sigma(\hat{\mathbf{z}}_k) \right) \\ + \sum_{k=1}^{n-1} \min_{\mathbf{z}_k \in [\sigma(\mathbf{l}_k), \sigma(\mathbf{u}_k)]} \left(-\boldsymbol{\mu}_{k+1}^T W_{k+1} \mathbf{z}_k + \boldsymbol{\lambda}_k^T \mathbf{z}_k \right) + \min_{\mathbf{z}_0 \in \mathcal{C}} -\boldsymbol{\mu}_1^T W_1 \mathbf{z}_0.\end{aligned} \tag{5.8.10}$$

(with the convention that $\boldsymbol{\mu}_n = -I$)

As a reminder, the formulation of our dual (5.5.5) is:

$$\begin{aligned} q(\boldsymbol{\rho}) = \min{}_{\mathbf{z}, \hat{\mathbf{z}}} \; & \hat{\mathbf{z}}_{A,n} + \Sigma_{k=1..n-1} \boldsymbol{\rho}_k^T (\hat{\mathbf{z}}_{B,k} - \hat{\mathbf{z}}_{A,k}) \\ \text{s.t.} \quad & \mathcal{P}_0(\mathbf{z}_0, \hat{\mathbf{z}}_{A,1}); \quad \mathcal{P}_k(\hat{\mathbf{z}}_{B,k}, \hat{\mathbf{z}}_{A,k+1}) \qquad \text{for } k \in [1..n-1] \end{aligned} \tag{5.8.11}$$

which can be decomposed as:

$$q(\boldsymbol{\rho}) = \sum_{k=1}^{n-1} \left(\min_{\substack{\hat{\mathbf{z}}_{B,k}, \hat{\mathbf{z}}_{A,k+1} \\ \in \mathcal{P}_k(\cdot,\cdot)}} \boldsymbol{\rho}_k^T \hat{\mathbf{z}}_{B,k} - \boldsymbol{\rho}_{k+1}^T \hat{\mathbf{z}}_{A,k+1} \right) + \min_{\substack{\mathbf{z}_0, \hat{\mathbf{z}}_{A,1} \\ \in \mathcal{P}_0(\cdot,\cdot)}} -\boldsymbol{\rho}_1^T \hat{\mathbf{z}}_{A,1}, \tag{5.8.12}$$

with the convention that $\boldsymbol{\rho}_n = -I$. We will show that when we chose the dual variables such that

$$\boldsymbol{\rho}_k = \boldsymbol{\mu}_k, \tag{5.8.13}$$

we obtain a tighter bound than the ones given by (5.8.10).

We will start by showing that the term being optimised over $\mathcal{P}_\prime$ is equivalent to some of the terms in (5.8.10):

$$\min_{\substack{\mathbf{z}_0, \hat{\mathbf{z}}_{A,1} \\ \in \mathcal{P}_0(\cdot,\cdot)}} -\boldsymbol{\rho}_1^T \hat{\mathbf{z}}_{A,1} = \min_{\substack{\mathbf{z}_0, \hat{\mathbf{z}}_{A,1} \\ \in \mathcal{P}_0(\cdot,\cdot)}} -\boldsymbol{\mu}_1^T \hat{\mathbf{z}}_{A,1} = -\boldsymbol{\mu}_1^T \mathbf{b}_1 + \min_{\mathbf{z}_0 \in \mathcal{C}} -\boldsymbol{\mu}_1^T W_1 \mathbf{z}_0 \tag{5.8.14}$$

The first equality is given by the replacement formula (5.8.13), while the second one is given by performing the replacement of $\hat{\mathbf{z}}_{A,1}$ with his expression in $\mathcal{P}_0$

We will now obtain a lower bound of the term being optimised over $\mathcal{P}_k$. Let's start by plugging in the values using the formula (5.8.13) and replace the value of $\hat{\mathbf{z}}_{A,k+1}$ using the constraint.

$$\min_{\substack{\hat{\mathbf{z}}_{B,k}, \hat{\mathbf{z}}_{A,k+1} \\ \in \mathcal{P}_k(\cdot,\cdot)}} \left(\boldsymbol{\rho}_k^T \hat{\mathbf{z}}_{B,k} - \boldsymbol{\rho}_{k+1}^T \hat{\mathbf{z}}_{A,k+1} \right) = -\boldsymbol{\mu}_{k+1}^T \mathbf{b}_{k+1} + \min_{\substack{\hat{\mathbf{z}}_{B,k} \in [\mathbf{l}_k, \mathbf{u}_k] \\ \texttt{cvx_hull}(\hat{\mathbf{z}}_{B,k}, \mathbf{z}_k)}} \left(\boldsymbol{\mu}_k^T \hat{\mathbf{z}}_{B,k} - \boldsymbol{\mu}_{k+1}^T W_{k+1} \mathbf{z}_k \right) \tag{5.8.15}$$

Focusing on the second term that contains the minimization over the convex hull, we will obtain a lower bound.

If $\boldsymbol{\lambda}_k \geq 0$, we can add a term $\boldsymbol{\lambda}_k^T(\sigma(\hat{\mathbf{z}}_k) - \sigma(\hat{\mathbf{z}}_k)) = 0$ and obtain:

$$\begin{aligned}
&\min_{\substack{\hat{\mathbf{z}}_{B,k} \in [\mathbf{l}_k, \mathbf{u}_k] \\ \texttt{cvx_hull}(\hat{\mathbf{z}}_{B,k}, \mathbf{z}_k)}} \left(\boldsymbol{\mu}_k^T \hat{\mathbf{z}}_{B,k} \; - \; \boldsymbol{\mu}_{k+1}^T W_{k+1} \mathbf{z}_k\right) \\
&= \min_{\substack{\hat{\mathbf{z}}_{B,k} \in [\mathbf{l}_k, \mathbf{u}_k] \\ \texttt{cvx_hull}(\hat{\mathbf{z}}_{B,k}, \mathbf{z}_k)}} \left(\boldsymbol{\mu}_k^T \hat{\mathbf{z}}_{B,k} \; - \; \boldsymbol{\mu}_{k+1}^T W_{k+1} \mathbf{z}_k + \boldsymbol{\lambda}_k^T(\sigma(\hat{\mathbf{z}}_{B,k}) - \sigma(\hat{\mathbf{z}}_{B,k}))\right) \\
&\geq \min_{\hat{\mathbf{z}}_{B,k} \in [\mathbf{l}_k, \mathbf{u}_k]} \left(\boldsymbol{\mu}_k^T \hat{\mathbf{z}}_{B,k} - \boldsymbol{\lambda}_k^T \sigma(\hat{\mathbf{z}}_{B,k})\right) + \min_{\substack{\hat{\mathbf{z}}_{B,k} \in [\mathbf{l}_k, \mathbf{u}_k] \\ \texttt{cvx_hull}(\hat{\mathbf{z}}_{B,k}, \mathbf{z}_k)}} \left(-\boldsymbol{\mu}_{k+1}^T W_{k+1} \mathbf{z}_k + \boldsymbol{\lambda}_k^T \sigma(\hat{\mathbf{z}}_{B,k})\right) \\
&= \min_{\hat{\mathbf{z}}_{B,k} \in [\mathbf{l}_k, \mathbf{u}_k]} \left(\boldsymbol{\mu}_k^T \hat{\mathbf{z}}_{B,k} - \boldsymbol{\lambda}_k^T \sigma(\hat{\mathbf{z}}_{B,k})\right) + \min_{\mathbf{z}_k \in [\sigma(\mathbf{l}_k), \sigma(\mathbf{u}_k)]} \left(-\boldsymbol{\mu}_{k+1}^T W_{k+1} \mathbf{z}_k + \boldsymbol{\lambda}_k^T \mathbf{z}_k\right)
\end{aligned} \tag{5.8.16}$$

Equality between the first line and the second ones comes from the fact that we are adding a term equal to zero. The inequality between the second line and the third is due to the fact that the sum of minimum is going to be lower than the minimum of the sum. The final line is obtained by noticing that, given that $\boldsymbol{\lambda}_k$ is positive, when we are minimizing $\boldsymbol{\lambda}_k^T \mathbf{z}_k$ over $\texttt{cvx_hull}(\hat{\mathbf{z}}_{B,k}, \mathbf{z}_k)$, it will always be equal to $\boldsymbol{\lambda}_k^T \sigma(\hat{\mathbf{z}}_{B,k})$.

Now if $\boldsymbol{\lambda}_k \leq 0$, we know that the term $\boldsymbol{\lambda}_k^T(\mathbf{z}_k - \sigma(\hat{\mathbf{z}}_{B,k}))$ is smaller than 0, as $\mathbf{z}_k \geq \sigma(\hat{\mathbf{z}}_{B,k})$ when $(\hat{\mathbf{z}}_{B,k}, \mathbf{z}_k) \in \texttt{cvx_hull}(\hat{\mathbf{z}}_{B,k}, \mathbf{z}_k)$. This is easily visible in Figure 5.1.

$$\begin{aligned}
&\min_{\substack{\hat{\mathbf{z}}_{B,k} \in [\mathbf{l}_k, \mathbf{u}_k] \\ \texttt{cvx_hull}(\hat{\mathbf{z}}_{B,k}, \mathbf{z}_k)}} \left(\boldsymbol{\mu}_k^T \hat{\mathbf{z}}_{B,k} \; - \; \boldsymbol{\mu}_{k+1}^T W_{k+1} \mathbf{z}_k\right) \\
&\geq \min_{\substack{\hat{\mathbf{z}}_{B,k} \in [\mathbf{l}_k, \mathbf{u}_k] \\ \texttt{cvx_hull}(\hat{\mathbf{z}}_{B,k}, \mathbf{z}_k)}} \left(\boldsymbol{\mu}_k^T \hat{\mathbf{z}}_{B,k} \; - \; \boldsymbol{\mu}_{k+1}^T W_{k+1} \mathbf{z}_k + \boldsymbol{\lambda}_k^T(\mathbf{z}_k - \sigma(\hat{\mathbf{z}}_{B,k}))\right) \\
&\geq \min_{\hat{\mathbf{z}}_{B,k} \in [\mathbf{l}_k, \mathbf{u}_k]} \left(\boldsymbol{\mu}_k^T \hat{\mathbf{z}}_{B,k} - \boldsymbol{\lambda}_k^T \sigma(\hat{\mathbf{z}}_{B,k})\right) + \min_{\mathbf{z}_k \in [\sigma(\mathbf{l}_k), \sigma(\mathbf{u}_k)]} \left(-\boldsymbol{\mu}_{k+1}^T W_{k+1} \mathbf{z}_k + \boldsymbol{\lambda}_k^T \mathbf{z}_k\right)
\end{aligned} \tag{5.8.17}$$

The first inequality is obtained by adding a negative term, and the second is obtained because the sum of minimums is lower than the minimum of the sums. We end up recovering the same lower bound than (5.8.16).

Summing up the lower bounds for all the terms in (5.8.12), as given by equation (5.8.14), (5.8.16) and (5.8.17), we obtain the formulation of Problem (5.8.10). We conclude that the bound obtained by Dvijotham et al. [53] is necessarily smaller than the bound derived using our dual. Given that we are computing lower bounds, this means that their bound is looser.

5.8.5 Initialisation of the dual

We will now show that the solution generated by the method of Wong and Kolter [171] can be used to provide initialization to our dual (5.5.5), even though both duals were derived differently. Theirs is derived from the Lagrangian dual of the Planet [58] relaxation, while ours come from Lagrangean decomposition. However, we will show that a solution that can be extracted from theirs lead to the exact same bound they generate.

As a reminder, we reproduce results proven in Wong and Kolter [171]. In order to obtain the bound, the problem they solved is:

$$\begin{aligned} \min_{\hat{\mathbf{z}}_n} \quad & \mathbf{c}^T \hat{\mathbf{z}}_n \\ \text{such that} \quad & \hat{\mathbf{z}}_n \in \tilde{\mathcal{Z}}_\epsilon(x), \end{aligned} \tag{5.8.18}$$

When $\mathbf{c}$ is just an identity matrix (so we optimise the bounds of each neuron), we recover the problem (5.4.2). $\tilde{\mathcal{Z}}_\epsilon(x)$ correspond to the contraints (5.4.2a) to (5.4.2b).

Based on the **Theorem 1**, and the equations (8) to (10) [171], the dual problem that they derive has an objective function of[2]:

$$-\sum_{i=1}^{n-1} \boldsymbol{\nu}_i^T \mathbf{b}_i - \mathbf{x}^T \hat{\boldsymbol{\nu}}_0 - \epsilon \|\hat{\boldsymbol{\nu}}_0\|_1 + \sum_{i=1}^{n-1} \sum_{j \in \mathcal{I}_i} l_{i,j} \left[\nu_{i,j}\right]_+ \tag{5.8.19}$$

with a solution given by

$$\begin{aligned} \boldsymbol{\nu}_n &= -\mathbf{c} \\ \hat{\boldsymbol{\nu}}_i &= -W_{i+1}^T D_{i+2} W_{i+2}^T \dots D_n W_n^T \mathbf{c} \\ \boldsymbol{\nu}_i &= D_i \hat{\boldsymbol{\nu}}_i \end{aligned} \tag{5.8.20}$$

where D_i is a diagonal matrix with entries

$$(D_i)_{jj} = \begin{cases} 0 & j \in \mathcal{I}_i^- \\ 1 & j \in \mathcal{I}_i^+ \\ \frac{u_{i,j}}{u_{i,j} - l_{i,j}} & j \in \mathcal{I}_i \end{cases}, \tag{5.8.21}$$

and $\mathcal{I}_i^-, \mathcal{I}_i^+, \mathcal{I}_i$ correspond respectively to blocked ReLUs (with always negative inputs), passing ReLUs (with always positive inputs) and ambiguous ReLUs.

We will now prove that, by taking as our solution of (5.5.5)

$$\boldsymbol{\rho}_k = \boldsymbol{\nu}_k, \tag{5.8.22}$$

we obtain exactly the same bound.

[2] We adjust the indexing slightly because Wong et al. note the input $\mathbf{z}_1$, while we refer to it as $\mathbf{z}_0$

The problem we're solving is (5.5.5), so, given a choice of $\boldsymbol{\rho}$, the bound that we generate is:

$$\begin{aligned} &\min_{\mathbf{z},\hat{\mathbf{z}}} \hat{\mathbf{z}}_{A,n} + \sum_{k=1..n-1} \boldsymbol{\rho}_k^T \left(\hat{\mathbf{z}}_{B,k} - \hat{\mathbf{z}}_{A,k}\right) \\ &\text{s.t.} \quad \mathcal{P}_0(\mathbf{z}_0, \hat{\mathbf{z}}_{A,1}); \qquad \mathcal{P}_k(\hat{\mathbf{z}}_{B,k}, \hat{\mathbf{z}}_{A,k+1}) \quad \text{for } k \in [1..n-1]\,. \end{aligned} \tag{5.8.23}$$

which can be decomposed into several subproblems:

$$\sum_{k=1}^{n-1} \left(\min_{\substack{\hat{\mathbf{z}}_{B,k}, \hat{\mathbf{z}}_{A,k+1} \\ \in \mathcal{P}_k(\cdot,\cdot)}} \boldsymbol{\rho}_k^T \hat{\mathbf{z}}_{B,k} \; - \; \boldsymbol{\rho}_{k+1}^T \hat{\mathbf{z}}_{A,k+1} \right) + \min_{\substack{\mathbf{z}_0, \hat{\mathbf{z}}_{A,1} \\ \in \mathcal{P}_0(\cdot,\cdot)}} -\boldsymbol{\rho}_1^T \hat{\mathbf{z}}_{A,1} \tag{5.8.24}$$

where we take the convention consistent with (5.8.22) that $\boldsymbol{\rho}_n = \boldsymbol{\nu}_n = -\mathbf{c} = -I$.

Let's start by evaluating the problem over $\mathcal{P}_0$, having replaced $\boldsymbol{\rho}_1$ by $\boldsymbol{\nu}_1$ in accordance with (5.8.22):

$$\begin{aligned} \min_{\mathbf{z}_0, \hat{\mathbf{z}}_{A,1}} \quad & -\boldsymbol{\nu}_1^T \hat{\mathbf{z}}_{A,1} \\ \text{such that} \quad & \hat{\mathbf{z}}_1 = W_1 \mathbf{z}_0 + \mathbf{b}_1 \\ & \mathbf{z}_0 \in \mathcal{C}, \end{aligned} \tag{5.8.25}$$

where in the context of [171], $\mathcal{C}$ is defined by the bounds $\mathbf{l}_0 = x - \epsilon$ and $\mathbf{u}_0 = x + \epsilon$. We already explained how to solve this subproblem in the context of our Frank Wolfe optimisation, see Equation (5.5.9)). Plugging in the solution into the objective function gives us:

$$\begin{aligned} \min_{\substack{\mathbf{z}_0, \hat{\mathbf{z}}_{A,1} \\ \in \mathcal{P}_0(\cdot,\cdot)}} \quad -\boldsymbol{\rho}_1^T \hat{\mathbf{z}}_{A,1} &= -\boldsymbol{\nu}_1^T W_1 \left(\mathbb{1}_{\boldsymbol{\nu}_1^T W_1 > 0} \cdot \mathbf{l}_0 + \mathbb{1}_{\boldsymbol{\nu}_1^T W_1 \leq 0} \cdot \mathbf{u}_0 \right) - \boldsymbol{\nu}_1^T b_1 \\ &= -\hat{\boldsymbol{\nu}}_0^T \left(\mathbb{1}_{\hat{\nu}_0 > 0} \cdot \mathbf{l}_0 + \mathbb{1}_{\hat{\nu}_0 \leq 0} \cdot \mathbf{u}_0 \right) - \boldsymbol{\nu}_1^T b_1 \\ &= -\hat{\boldsymbol{\nu}}_0^T \mathbf{x} + \epsilon \hat{\boldsymbol{\nu}}_0^T \mathbb{1}_{\hat{\nu}_0 > 0} - \epsilon \hat{\boldsymbol{\nu}}_0^T \mathbb{1}_{\hat{\nu}_0 < 0} - \boldsymbol{\nu}_1^T b_1 \\ &= -\hat{\boldsymbol{\nu}}_0^T \mathbf{x} + \epsilon \|\hat{\boldsymbol{\nu}}_0\|_1 - \boldsymbol{\nu}_1^T \mathbf{b}_1 \end{aligned} \tag{5.8.26}$$

We will now evaluate the values of the problem over $\mathcal{P}_k$. Once again, we replace the $\boldsymbol{\rho}$ with the appropriate values of $\boldsymbol{\nu}$ in accordance to (5.8.22):

$$\begin{aligned} \min_{\hat{\mathbf{z}}_{B,k}, \hat{\mathbf{z}}_{A,k+1}} \quad & \boldsymbol{\nu}_k^T \hat{\mathbf{z}}_{B,k} + \boldsymbol{\nu}_k^T \hat{\mathbf{z}}_{A,k+1} \\ \text{s.t} \quad & \mathbf{l}_k \leq \hat{\mathbf{z}}_{B,k} \leq \mathbf{u}_k \\ & \texttt{cvx_hull}_\sigma \left(\hat{\mathbf{z}}_{B,k}, \mathbf{z}_k, \mathbf{l}_k, \mathbf{u}_k \right) \\ & \hat{\mathbf{z}}_{A,k+1} = W_{k+1} \mathbf{z}_k + \mathbf{b}_{k+1}, \end{aligned} \tag{5.8.27}$$

We can rephrase the problem such that it decomposes completely over the ReLUs, by merging the last equality constraint into the objective function:

$$\begin{aligned} \min_{\hat{\mathbf{z}}_{B,k}, \mathbf{z}_k} \quad & \boldsymbol{\nu}_k^T \hat{\mathbf{z}}_{B,k} - \boldsymbol{\nu}_{k+1}^T W_{k+1} \mathbf{z}_k - \boldsymbol{\nu}_{k+1}^T \mathbf{b}_{k+1} \\ \text{s.t} \quad & \mathbf{l}_k \leq \hat{\mathbf{z}}_{B,k} \leq \mathbf{u}_k \\ & \texttt{cvx_hull}_\sigma \left(\hat{\mathbf{z}}_{B,k}, \mathbf{z}_k, \mathbf{l}_k, \mathbf{u}_k \right) \end{aligned} \tag{5.8.28}$$

Note that $\boldsymbol{\nu}_{k+1}^T W_{k+1} = \hat{\boldsymbol{\nu}}_k^T$. We will now omit the term $-\boldsymbol{\nu}_{k+1}^T \mathbf{b}_{k+1}$ which doesn't depend on the optimisation variable anymore and will therefore be directly passed down to the value of the bound.

For the rest of the problem, we will distinguish the different cases of ReLU, $\mathcal{I}^+, \mathcal{I}^-$ and $\mathcal{I}$.

In the case of a ReLU $j \in \mathcal{I}^+$, we have $\texttt{cvx_hull}_\sigma(\dots) \quad \equiv \quad \hat{\mathbf{z}}_{B,k}[j] = \mathbf{z}_k[j]$, so the replacement of $\mathbf{z}_k[j]$ makes the objective function term for j becomes $(\boldsymbol{\nu}_k[j] - \hat{\boldsymbol{\nu}}_k[j])\, \hat{\mathbf{z}}_{B,k}[j]$. Given that $j \in \mathcal{I}^+$, this means that the corresponding element in D_k is 1, so $\boldsymbol{\nu}_k[j] = \hat{\boldsymbol{\nu}}_k[j]$, which means that the term in the objective function is zero.

In the case of a ReLU $j \in \mathcal{I}^-$, we have $\texttt{cvx_hull}_\sigma(\dots) \quad \equiv \quad \mathbf{z}_k[j] = 0$, so the replacement of $\mathbf{z}_k[j]$ makes the objective function term for j becomes $\boldsymbol{\nu}_k[j] \hat{\mathbf{z}}_{B,k}[j]$. Given that $j \in \mathcal{I}^-$, this means that the corresponding element in D_k is 0, so $\boldsymbol{\nu}_k[j] = 0$. This means that the term in the objective function is zero.

The case for a ReLU $j \in \mathcal{I}$ is more complex. We apply the same strategy of picking the appropriate inequalities in `cvx_hull` as we did for obtaining Equation (5.5.12), which leads us to:

$$\begin{aligned} \min_{\hat{\mathbf{z}}_{B,k}} \quad & (\boldsymbol{\nu}_k[j] - [\hat{\boldsymbol{\nu}}_k[j]]_+ \frac{u_k[j]}{u_k[j] - l_k[j]}) \mathbf{z}_{B,k}[j] \\ & - [\hat{\boldsymbol{\nu}}_k[j]]_- \max(\mathbf{z}_{B,k}, 0) \\ & + [\hat{\boldsymbol{\nu}}_k[j]]_+ \frac{u_k[j]}{u_k[j] - l_k[j]} l_k[j] \\ \text{s.t} \quad & \mathbf{l}_k \leq \hat{\mathbf{z}}_{B,k} \leq \mathbf{u}_k \end{aligned} \tag{5.8.29}$$

Once again, we can recognize that the coefficient in front of $\mathbf{z}_{B,k}[j]$ (in the first line of the objective function) is equal to 0 by construction of the values of $\boldsymbol{\nu}$ and $\hat{\boldsymbol{\nu}}$. The coefficient in front of $\max(\mathbf{z}_{B,k}, 0)$ in the second line is necessarily positive. Therefore, the minimum value for this problem will necessarily be the constant term in the third line (reachable for $\mathbf{z}_{B,k} = 0$ which is feasible due to j being in $\mathcal{I}$). The value of this constant term is:

$$[\hat{\boldsymbol{\nu}}_k[j]]_+ \frac{u_k[j]}{u_k[j] - l_k[j]} l_k[j] = [\boldsymbol{\nu}_k[j]]_+ \, l_k[j] \tag{5.8.30}$$

Regrouping the different cases and the constant term of problem (5.8.28) that we ignored, we find that the value of the term corresponding to the $\mathcal{P}_k$ subproblem is:

$$-\boldsymbol{\nu}_{k+1}^T \mathbf{b}_{k+1} + \sum_{j \in \mathcal{I}} [\boldsymbol{\nu}_k[j]]_+ \, l_k[j] \tag{5.8.31}$$

Plugging all the optimisation result into Equation (5.8.24), we obtain:

$$\begin{aligned}
&\sum_{k=1}^{n-1} \left(\min_{\mathcal{P}_k} \boldsymbol{\rho}_k^T \hat{\mathbf{z}}_{B,k} - \boldsymbol{\rho}_{k+1}^T \hat{\mathbf{z}}_{A,k+1} \right) + \min_{\mathcal{P}_0} -\boldsymbol{\rho}_1^T \hat{\mathbf{z}}_{A,1} \\
&\quad = \sum_{k=1}^{n-1} \left(-\boldsymbol{\nu}_{k+1}^T \mathbf{b}_{k+1} + \sum_{j \in \mathcal{I}} [\boldsymbol{\nu}_k[j]]_+ \, l_k[j] \right) - \hat{\boldsymbol{\nu}}_0^T \mathbf{x} + \epsilon \|\hat{\boldsymbol{\nu}}_0\|_1 - \boldsymbol{\nu}_1^T \mathbf{b}_1 \\
&\quad = -\sum_{k=1}^{n-1} \boldsymbol{\nu}_k^T \mathbf{b}_k - \hat{\boldsymbol{\nu}}_0^T \mathbf{x} - \epsilon \|\hat{\boldsymbol{\nu}}_0\|_1 + \sum_{k=1}^{n-1} \sum_{j \in \mathcal{I}} [\boldsymbol{\nu}_k[j]]_+ \, l_k[j],
\end{aligned} \tag{5.8.32}$$

which is exactly the value given by Wong and Kolter [171] in Equation (5.8.19).

6
Discussion and Future Work

If Neural Networks are to become more commonly adopted as components of widely used systems, our methods for ensuring their robustness and the correctness of their behaviour need to be improved. This thesis presented several directions towards improved verification algorithms.

6.1 Methodology

We argue that realizing the connection between formal verification of properties on neural network and optimisation provides useful tools to improve on existing algorithms. The formulation of Chapter 3 has made it feasible for us to shed light on the difficulties of the problem and on what directions of improvements were worth pursuing. By reframing the verification of properties into optimisation problems (or more accurately bounding problems), we can use standard techniques of the field of optimization to improve our algorithms.

The use of Lagrangean dual methods [53] allows us to derive algorithm that are any-time, that is, they generate valid bounds at every step. This removes the previously existing constraint that bounding problems had to be solved to convergence in order to generate usable bounds.

The lens of optimization also allows us to establish parallels with other problems to suggest promising avenues of research. For example, in the past, the study of the various possible relaxations for conditional random fields [101] lead to significant improvements in terms of the quality of the approximate solution found and the bounds obtained. Taking inspiration of the approaches employed then can lead to insights on how we

might be able to improve the relaxation we use for neural network and the bounding procedures our algorithms are based on.

Thanks to the reformulating of existing verification method as branch and bound, we realized that all incomplete verification method could be understood as bounding procedures and used as building blocks for a branch and bound based complete verification method as we described in Chapter 3. We can directly leverage any progress made to incomplete verification methods into improvements to the field of complete verification algorithms, which is what we attempted in Chapter 5.

Mixed integer programming solvers are at the moment the most widely used tool to perform complete verification of neural networks. However, their applicability has some limitations.

The strength of mixed integer programming solver is that they are capable of easily encoding new properties of interest, as we demonstrated ourselves in Chapter 4. This enables practitioners to rapidly iterate on ideas, without having to implement the underlying solving algorithm.

However, they are not capable of encoding non piecewise-linear neural networks, such as those using softplus, sigmoid or hyperbolic tangent as activation function. Indeed, the mixed integer programming formalism only support linear and integrality constraints. Our reframing of the verification procedure as an optimization problem provides a way forward. The branch and bound algorithm does not rely on any assumption beyond access to an appropriate bounding function, which can be provided for example by the method discussed in Chapter 5. As a result, complete verification of property based on non piecewise-linear neural network can be achieved by branch and bound. In addition, verification methods based on mixed integer programming are necessarily dependent on a solver that has no specific knowledge of the problem. As we have shown in Chapter 3, by specializing our solving algorithm to our task, we have the possibility of achieving potentially significant performance gains.

6.2 Scalability

The main limitation at the moment preventing a more widespread adoption of formal methods in machine learning is their lack of scalability. Aside from robustness to very small amount of noise, there is no hope of performing successful verification for models containing more than a few thousand hidden units.

To alleviate this, several research directions present themselves. Algorithmic improvements in our optimisation methods are the most straightforward. As the computation of bounds is currently the bottleneck, faster solvers for the relaxation that we are currently

using would have a direct impact on performance, and by consequence on scalability. This is the approach we followed in Chapter 5. Another possibility lies in developing new types of tighter relaxations, such as the one proposed by Anderson et al. [8] or Singh et al. [152]. Those tighter relaxations would lead to better bounds, potentially resulting in a reduction of the depth of the required search tree to achieve verification.

Another promising direction lies in identifying better decision to make during the verification process. The benefits exhibited by smart branching in Chapter 3 demonstrate that even a slightly more advanced branching process can have a significant effect on runtime, by splitting the original domain into verifiable subdomains in less steps. Identifying better heuristics, or learning them from data, could result in further improvements.

It is notable that as part of the verification process, several bounds are computed that don't lead to verification and therefore require more splitting. Any bound that we compute that does not lead to the underlying domain being considered verified can be thought of as an inefficiency. If we had known in advance that the bounding procedure would be inconclusive, we would have done further splitting before attempting to verify it. While it is not possible to know if the property can be verified on a subdomain without running the bounding procedure, we might be able to predict it with high enough accuracy that performance gains can be achieved. Successful predictions would result in saving the algorithms calls to the bounding procedures, while incorrect predictions might lead us to perform unnecessary splitting. This however would compromise neither the soundness nor the completeness of the verification algorithm. Indeed, this would just impact the structure of the search tree, but the verification status of the property would still depend on the verification of all leaf nodes.

Another source of unnecessary computation is the uncertainty on the results we are trying to achieve. With the proposed branch and bound framework, computation used to search for counter-examples does not inform the bounding process. The bounding process does not facilitate the search for counter-examples either, aside from potentially narrowing the search domain when pruning. In each case, at least one of the process that we have to run is wasted computation that can not succeed. Having higher quality adversarial attacks that perform better at discovering counterexamples would allow us to not attempt verification for properties for which counterexamples can be surfaced.

6.3 Generality

As we advance towards making verification procedures scale, it is also beneficial to ensure that they can apply to as wide a variety of systems as possible. Chapter 4 has shown how recurrent networks, even those having variable length outputs, could be handled.

Chapter 5 has demonstrated how to derive bounding procedures necessary for verification for networks with non piecewise linear activation functions.

Allowing for more complex and meaningful constraints on the input is critical to deliver value for useful applications. While Chapter 5 has shown how to go beyond the interval constraints typical in the literature, real benefits will appear when semantic constraints can be incorporated. Incorporating generative models of allowable inputs and using those as specification might provide a stepping stone towards that goal, but will only lead to verification modulo a generative model, which would then also need to be verified.

6.4 Incorporating Properties into Training

This thesis focused mainly on the verification of already trained models. We assumed that the model was satisfying the property and were looking for methods to confirm our belief. It is however notable that in most cases, models learned with standard methods that do not include desired properties at training time will not usually respect the specification. This is expected as nothing in their training forced them to satisfy it. It is therefore crucial for progress in training procedures to be made, so that the learned model can be made to respect the specification. Scalability problems are even higher in the case of training procedures as the verification methods need to be used once per iteration, leading to millions of invocations.

While some success was already achieved in robust training of Neural Networks to obtain models producing prediction resilient to small norm perturbations, further work is necessary. Why does enforcing invariances around the training data points leads to the same invariance holding up around new data? Why do certain looser relaxations [71] lead to improved generalization of robustness when compared to tighter ones? Is the decline in nominal accuracy currently observed [71, 121, 158, 171] that comes with including additional specification unavoidable? All these questions are open and need to be addressed.

It also remain to be seen whether it is possible to incorporate more diverse specifications in the training objective. While it has been shown several times in the context of robustness to small norm perturbations, empirical validation of what other type of specifications are easy to incorporate remains missing.

6.5 Theoretical understanding

In addition to engineering progress and improvements in the system that we build, additional understanding would also be beneficial. When trying to fit a specification, machine learning methods are never able to indicate when a model might mot have the capacity to produce the desired results. It is hard to identify what is a limitation of the learning algorithms, as opposed to a limitation of the model. Additional research on understanding the complexity requirements expansion caused by imposing specifications is warranted.

References

[1] Mahdieh Abbasi and Christian Gagné. “Robustness to adversarial examples through an ensemble of specialists”. In: *ICLR workshop* (2017).

[2] Thalaiyasingam Ajanthan, Alban Desmaison, Rudy Bunel, Mathieu Salzmann, Philip HS Torr, and M Pawan Kumar. “Efficient linear programming for dense CRFs”. In: *Conference on Computer Vision and Pattern Recognition* (2017).

[3] Michael Akintunde, Alessio Lomuscio, Lalit Maganti, and Edoardo Pirovano. “Reachability analysis for neural agent-environment systems”. In: *International Conference on Principles of Knowledge Representation and Reasoning* (2018).

[4] Michael E Akintunde, Andreea Kevorchian, Alessio Lomuscio, and Edoardo Pirovano. “Verification of RNN-based neural agent-environment systems”. In: *AAAI Conference on Artificial Intelligence* (2019).

[5] Salman Alsubaihi, Adel Bibi, Modar Alfadly, and Bernard Ghanem. “Expected tight bounds for robust deep neural network training”. In: *arXiv:1905.12418* (2019).

[6] Greg Anderson, Shankara Pailoor, Isil Dillig, and Swarat Chaudhuri. “Optimization and abstraction: a synergistic approach for analyzing neural network robustness”. In: *PLDI* (2019).

[7] Peter Anderson, Xiaodong He, Chris Buehler, Damien Teney, Mark Johnson, Stephen Gould, and Lei Zhang. “Bottom-up and top-down attention for image captioning and visual question answering”. In: *Conference on Computer Vision and Pattern Recognition* (2018).

[8] Ross Anderson, Joey Huchette, Christian Tjandraatmadja, and Juan Pablo Vielma. “Strong mixed-integer programming formulations for trained neural networks”. In: *Conference on Integer Programming and Combinatorial Optimization* (2019).

[9] Julia Angwin, Jeff Larson, Surya Mattu, and Lauren Kirchner. “Machine bias”. In: *ProPublica, May* (2016).

[10] Cem Anil, James Lucas, and Roger Grosse. “Sorting out lipschitz function approximation”. In: *International Conference on Machine Learning* (2019).

[11] Anish Athalye, Nicholas Carlini, and David A. Wagner. “Obfuscated gradients give a false sense of security: circumventing defenses to adversarial examples”. In: *International Conference on Machine Learning* (2018).

[12] Francis Bach. “Duality between subgradient and conditional gradient methods”. In: *SIAM Journal on Optimization* (2015).

[13] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. “Neural machine translation by jointly learning to align and translate”. In: *International Conference on Learning Representations* (2015).

[14] Mislav Balunovic, Maximilian Baader, Gagandeep Singh, Timon Gehr, and Martin Vechev. “Certifying geometric robustness of neural networks”. In: *Neural Information Processing Systems* (2019).

[15] Solon Barocas, Moritz Hardt, and Arvind Narayanan. “Fairness in machine learning”. In: *NIPS Tutorial* (2017).

[16] Clark Barrett, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. “Splitting on demand in SAT modulo theories”. In: *International Conference on Logic for Programming Artificial Intelligence and Reasoning* (2006).

[17] Osbert Bastani, Yani Ioannou, Leonidas Lampropoulos, Dimitrios Vytiniotis, Aditya Nori, and Antonio Criminisi. “Measuring neural net robustness with constraints”. In: *Neural Information Processing Systems* (2016).

[18] Dimitri P Bertsekas and John N Tsitsiklis. *Parallel and distributed computation: numerical methods*. Prentice hall Englewood Cliffs, NJ, 1989.

[19] Arjun Nitin Bhagoji, Daniel Cullina, and Prateek Mittal. “Dimensionality reduction as a defense against evasion attacks on machine learning classifiers”. In: *arXiv:1704.02654* (2017).

[20] 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* (2019).

[21] Noam Brown and Tuomas Sandholm. “Superhuman AI for multiplayer poker”. In: *Science* (2019).

[22] Jacob Buckman, Aurko Roy, Colin Raffel, and Ian Goodfellow. “Thermometer encoding: One hot way to resist adversarial examples”. In: *International Conference on Learning Representations* (2018).

[23] Rudy Bunel, Alban Desmaison, Pushmeet Kohli, Philip H. S. Torr, and M. Pawan Kumar. “Adaptive neural compilation”. In: *Neural Information Processing Systems* (2016).

[24] Rudy Bunel, Alban Desmaison, M. Pawan Kumar, Philip H. S. Torr, and Pushmeet Kohli. “Learning to superoptimize programs”. In: *International Conference on Learning Representations* (2017).

[25] Rudy Bunel, Ilker Turkaslan, Philip HS Torr, Pushmeet Kohli, and M Pawan Kumar. “A unified view of piecewise linear neural network verification”. In: *Neural Information Processing Systems* (2018).

[26] Rudy Bunel, Matthew Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. “Leveraging grammar and reinforcement learning for neural program synthesis”. In: *International Conference on Learning Representations* (2018).

[27] John N Buxton and Brian Randell. *Software Engineering Techniques: Report on a Conference Sponsored by the NATO Science Committee*. NATO Science Committee, 1970.

[28] Nicholas Carlini and David Wagner. “Adversarial examples are not easily detected: Bypassing ten detection methods”. In: *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security* (2017).

[29] Nicholas Carlini and David Wagner. “Towards evaluating the robustness of neural networks”. In: *2017 IEEE Symposium on Security and Privacy* (2017).

[30] Nicholas Carlini, Chang Liu, Jernej Kos, Úlfar Erlingsson, and Dawn Song. "The secret sharer: measuring unintended neural network memorization & extracting secrets". In: *arxiv:1802.08232* (2018).

[31] Chenyi Chen, Ari Seff, Alain Kornhauser, and Jianxiong Xiao. "DeepDriving: Learning affordance for direct perception in autonomous driving". In: *International Conference on Computer Vision* (2015).

[32] Hongge Chen, Huan Zhang, Pin-Yu Chen, Jinfeng Yi, and Cho-Jui Hsieh. "Attacking visual language grounding with adversarial examples: A case study on neural image captioning". In: *Annual Meeting of the association for Computational Linguistics* (2018).

[33] Pin-Yu Chen, Huan Zhang, Yash Sharma, Jinfeng Yi, and Cho-Jui Hsieh. "Zoo: Zeroth order optimization based black-box attacks to deep neural networks without training substitute models". In: *ACM Workshop on Artificial Intelligence and Security* (2017).

[34] Xinlei Chen and Abhinav Gupta. "Webly supervised learning of convolutional networks". In: *Conference on Computer Vision and Pattern Recognition* (2015).

[35] Xinlei Chen and C. Lawrence Zitnick. "Mind's eye: A recurrent visual representation for image caption generation". In: *Conference on Computer Vision and Pattern Recognition* (2015).

[36] Chih-Hong Cheng, Georg Nührenberg, and Harald Ruess. "Maximum resilience of artificial neural networks". In: *Automated Technology for Verification and Analysis* (2017).

[37] Chih-Hong Cheng, Georg Nührenberg, and Harald Ruess. "Verification of binarized neural networks". In: *arXiv:1710.03107* (2017).

[38] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, et al. "Wide & deep learning for recommender systems". In: *Proceedings of the 1st workshop on deep learning for recommender systems* (2016).

[39] Minhao Cheng, Jinfeng Yi, Huan Zhang, Pin-Yu Chen, and Cho-Jui Hsieh. "Seq2Sick: Evaluating the robustness of sequence-to-sequence models with adversarial examples". In: *arXiv:1803.01128* (2018).

[40] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. "Learning phrase representations using RNN encoder-decoder for statistical machine translation". In: *Conference on Empirical Methods in Natural Language Processing* (2014).

[41] Sumit Chopra, Michael Auli, and Alexander M Rush. "Abstractive sentence summarization with attentive recurrent neural networks". In: *Annual Conference of the North American Chapter of the Association for Computational Linguistics* (2016).

[42] Moustapha Cisse, Piotr Bojanowski, Edouard Grave, Yann Dauphin, and Nicolas Usunier. "Parseval networks: Improving robustness to adversarial examples". In: *International Conference on Machine Learning* (2017).

[43] Jeremy M Cohen, Elan Rosenfeld, and J Zico Kolter. "Certified adversarial robustness via randomized smoothing". In: *International Conference on Machine Learning* (2019).

[44] Patrick Cousot and Radhia Cousot. "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints". In: (1977).

[45] Francesco Croce, Maksym Andriushchenko, and Matthias Hein. “Provable robustness of ReLU networks via maximization of linear regions”. In: *International Conference on Artificial Intelligence and Statistics* (2019).

[46] Jeffrey De Fauw, Joseph R Ledsam, Bernardino Romera-Paredes, Stanislav Nikolov, Nenad Tomasev, Sam Blackwell, Harry Askham, Xavier Glorot, Brendan O’Donoghue, Daniel Visentin, et al. “Clinically applicable deep learning for diagnosis and referral in retinal disease”. In: *Nature medicine* (2018).

[47] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. “Imagenet: A large-scale hierarchical image database”. In: *Conference on Computer Vision and Pattern Recognition* (2009).

[48] Alban Desmaison, Rudy Bunel, Pushmeet Kohli, Philip H. S. Torr, and M. Pawan Kumar. “Efficient continuous relaxations for dense CRF”. In: *European Conference on Computer Vision* (2016).

[49] Jacob Devlin, Rudy R Bunel, Rishabh Singh, Matthew Hausknecht, and Pushmeet Kohli. “Neural program meta-induction”. In: *Neural Information Processing Systems* (2017).

[50] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. “RobustFill: Neural program learning under noisy I/O”. In: *International Conference on Machine Learning* (2017).

[51] Guneet S Dhillon, Kamyar Azizzadenesheli, Zachary C Lipton, Jeremy Bernstein, Jean Kossaifi, Aran Khanna, and Anima Anandkumar. “Stochastic activation pruning for robust adversarial defense”. In: *International Conference on Learning Representations* (2018).

[52] Souradeep Dutta, Susmit Jha, Sriram Sankaranarayanan, and Ashish Tiwari. “Output range analysis for deep feedforward neural networks”. In: *NASA Formal Methods Symposium* (2018).

[53] Krishnamurthy Dvijotham, Robert Stanforth, Sven Gowal, Timothy Mann, and Pushmeet Kohli. “A dual approach to scalable verification of deep networks”. In: *Uncertainty in Artificial Intelligence* (2018).

[54] Krishnamurthy Dvijotham, Sven Gowal, Robert Stanforth, Relja Arandjelovic, Brendan O’Donoghue, Jonathan Uesato, and Pushmeet Kohli. “Training verified learners with learned verifiers”. In: *arxiv:1805.10265* (2018).

[55] Krishnamurthy Dvijotham, Marta Garnelo, Alhussein Fawzi, and Pushmeet Kohli. “Verification of deep probabilistic models”. In: *SECML NeurIPS* (2018).

[56] Krishnamurthy Dvijotham, Robert Stanforth, Sven Gowal, Chongli Qin, Soham De, and Pushmeet Kohli. “Efficient neural network verification with exactness characterization”. In: *Uncertainty in Artificial Intelligence* (2019).

[57] Javid Ebrahimi, Anyi Rao, Daniel Lowd, and Dejing Dou. “HotFlip: White-box adversarial examples for NLP”. In: *arXiv:1712.06751* (2017).

[58] Ruediger Ehlers. “Formal verification of piece-wise linear feed-forward neural networks”. In: *Automated Technology for Verification and Analysis* (2017).

[59] Ruediger Ehlers. *Planet*. `https://github.com/progirep/planet`. 2017.

[60] R Evans, J Jumper, J Kirkpatrick, L Sifre, TFG Green, C Qin, A Zidek, A Nelson, A Bridgland, H Penedones, et al. “De novo structure prediction with deeplearning based scoring”. In: *Annual Review of Biochemistry* (2018).

[61] Hao Fang, Saurabh Gupta, Forrest N. Iandola, Rupesh Kumar Srivastava, Li Deng, Piotr Dollár, Jianfeng Gao, Xiaodong He, Margaret Mitchell, John C. Platt, C. Lawrence Zitnick, and Geoffrey Zweig. "From captions to visual concepts and back". In: *Conference on Computer Vision and Pattern Recognition* (2015).

[62] Matteo Fischetti and Jason Jo. "Deep neural networks and mixed integer linear optimization". In: *Constraints* (2018).

[63] Marguerite Frank and Philip Wolfe. "An algorithm for quadratic programming". In: *Naval Research Logistics Quarterly* (1956).

[64] Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin Vechev. "AI2: Safety and robustness certification of neural networks with abstract interpretation". In: *Security and Privacy (SP), 2018 IEEE Symposium on* (2018).

[65] Ulrich Germann, Michael Jahr, Kevin Knight, Daniel Marcu, and Kenji Yamada. "Fast decoding and optimal decoding for machine translation". In: *Annual Meeting of the association for Computational Linguistics* (2001).

[66] J. Gilmer, R. P. Adams, I. Goodfellow, D. Andersen, and G. E. Dahl. "Motivating the rules of the game for adversarial example research". In: *arxiv:1807.06732* (2018).

[67] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. "Rich feature hierarchies for accurate object detection and semantic segmentation". In: *Conference on Computer Vision and Pattern Recognition* (2014).

[68] Ambros Gleixner, Michael Bastubbe, Leon Eifler, Tristan Gally, Gerald Gamrath, Robert Lion Gottwald, Gregor Hendel, Christopher Hojny, Thorsten Koch, Marco E. Lübbecke, Stephen J. Maher, Matthias Miltenberger, Benjamin Müller, Marc E. Pfetsch, Christian Puchert, Daniel Rehfeldt, Franziska Schlösser, Christoph Schubert, Felipe Serrano, Yuji Shinano, Jan Merlin Viernickel, Matthias Walter, Fabian Wegscheider, Jonas T. Witt, and Jakob Witzig. *The SCIP Optimization Suite 6.0*. Technical Report. 2018.

[69] Zhitao Gong, Wenlu Wang, and Wei-Shinn Ku. "Adversarial and clean data are not twins". In: *arXiv:1704.04960* (2017).

[70] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. "Explaining and harnessing adversarial examples". In: *International Conference on Learning Representations* (2015).

[71] Sven Gowal, Krishnamurthy Dvijotham, Robert Stanforth, Rudy Bunel, Chongli Qin, Jonathan Uesato, Timothy Mann, and Pushmeet Kohli. "On the effectiveness of interval bound propagation for training verifiably robust models". In: *SECML NeurIPS* (2018).

[72] Alex Graves and Navdeep Jaitly. "Towards end-to-end speech recognition with recurrent neural networks". In: *International Conference on Machine Learning* (2014).

[73] Kathrin Grosse, Praveen Manoharan, Nicolas Papernot, Michael Backes, and Patrick McDaniel. "On the (statistical) detection of adversarial examples". In: *arXiv:1702.06280* (2017).

[74] Ignacio E Grossmann. "Review of nonlinear mixed-integer and disjunctive programming techniques". In: *Optimization and Engineering* (2002).

[75] Monique Guignard and Siwhan Kim. "Lagrangean decomposition: A model yielding stronger Lagrangean bounds". In: *Mathematical programming* (1987).

[76] Chuan Guo, Mayank Rana, Moustapha Cisse, and Laurens Van Der Maaten. “Countering adversarial images using input transformations”. In: *International Conference on Learning Representations* (2018).

[77] Arnaud Guyader, Nicolas Hengartner, and Eric Matzner-Løber. “Simulation and estimation of extreme quantiles and extreme probabilities”. In: *Applied Mathematics & Optimization* (2011).

[78] John Harrison. “Formal methods at Intel—An overview”. In: *Second NASA Formal Methods Symposium* (2010).

[79] Warren He, James Wei, Xinyun Chen, Nicholas Carlini, and Dawn Song. “Adversarial example defense: Ensembles of weak defenses are not strong”. In: {*USENIX*} *Workshop on Offensive Technologies* (2017).

[80] Matthias Hein and Maksym Andriushchenko. “Formal guarantees on the robustness of a classifier against adversarial manipulation”. In: *Neural Information Processing Systems* (2017).

[81] Dan Hendrycks and Kevin Gimpel. “Early methods for detecting adversarial images”. In: *ICLR Workshop* (2017).

[82] Karl Moritz Hermann, Tomas Kocisky, Edward Grefenstette, Lasse Espeholt, Will Kay, Mustafa Suleyman, and Phil Blunsom. “Teaching machines to read and comprehend”. In: *Neural Information Processing Systems* (2015).

[83] Timothy Hickey, Qun Ju, and Maarten H Van Emden. “Interval arithmetic: From principles to implementation”. In: *Journal of the ACM (JACM)* (2001).

[84] Sepp Hochreiter and Jürgen Schmidhuber. “Long short-term memory”. In: *Neural Computation* (1997).

[85] Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. “Safety verification of deep neural networks”. In: *International Conference on Computer Aided Verification* (2017).

[86] Eric Jang, Shixiang Gu, and Ben Poole. “Categorical reparameterization with gumbel-softmax”. In: *International Conference on Learning Representations* (2017).

[87] Frederick Jelinek. *Statistical Methods for Speech Recognition*. MIT press, 1997.

[88] Robin Jia and Percy Liang. “Adversarial examples for evaluating reading comprehension systems”. In: *Conference on Empirical Methods in Natural Language Processing* (2017).

[89] Matt Jordan, Justin Lewis, and Alexandros G Dimakis. “Provable certificates for adversarial examples: Fitting a ball in the union of polytopes”. In: *Neural Information Processing Systems* (2019).

[90] Thomas Joy, Alban Desmaison, Thalaiyasingam Ajanthan, Rudy Bunel, Mathieu Salzmann, Pushmeet Kohli, Philip H.S. Torr, and M. Pawan Kumar. “Efficient relaxations for dense CRFs with sparse higher-order potentials”. In: *SIAM Journal on Imaging Sciences* (2019).

[91] Richard M Karp. “Reducibility among combinatorial problems”. In: *Complexity of computer computations* (1972).

[92] Andrej Karpathy and Li Fei-Fei. “Deep visual-semantic alignments for generating image descriptions”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2017).

[93] Guy Katz, Clark Barrett, David Dill, Kyle Julian, and Mykel Kochenderfer. *Reluplex*. `https://github.com/guykatzz/ReluplexCav2017`. 2017.

[94] 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* (2017).

[95] Andreea Kevorchian. "Verification of Recurrent Neural Networks". In: (2018).

[96] Diederik P. Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *International Conference on Learning Representations* (2015).

[97] Ching-Yun Ko, Zhaoyang Lyu, Tsui-Wei Weng, Luca Daniel, Ngai Wong, and Dahua Lin. "POPQORN: Quantifying robustness of recurrent neural networks". In: *International Conference on Machine Learning* (2019).

[98] Philipp Koehn, Franz Josef Och, and Daniel Marcu. "Statistical phrase-based translation". In: *Annual Conference of the North American Chapter of the Association for Computational Linguistics* (2003).

[99] Philipp Krahenbuhl and Vladlen Koltun. "Efficient inference in fully connected CRFs with gaussian edge potentials". In: *Neural Information Processing Systems* (2011).

[100] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Neural Information Processing Systems* (2012).

[101] M. Pawan Kumar, Vladimir Kolmogorov, and Philip H. S. Torr. "An analysis of convex relaxations for MAP estimation of discrete MRFs". In: *Journal of Machine Learning Research* (2009).

[102] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. "Gradient-based learning applied to document recognition". In: *IEEE* (1998).

[103] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning". In: *Nature* (2015).

[104] Sergey Levine, Peter Pastor, Alex Krizhevsky, Julian Ibarz, and Deirdre Quillen. "Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection". In: *The International Journal of Robotics Research* (2018).

[105] Fangzhou Liao, Ming Liang, Yinpeng Dong, Tianyu Pang, Xiaolin Hu, and Jun Zhu. "Defense against adversarial attacks using high-level representation guided denoiser". In: *Conference on Computer Vision and Pattern Recognition* (2018).

[106] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. "Continuous control with deep reinforcement learning". In: *International Conference on Learning Representations* (2016).

[107] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. "Microsoft COCO: Common objects in context". In: *European Conference on Computer Vision* (2014).

[108] Dong C Liu and Jorge Nocedal. "On the limited memory BFGS method for large scale optimization". In: *Mathematical programming* (1989).

[109] John W Lloyd. "Practical advantages of declarative programming." In: *GULP-PRODE* (1994).

[110] Alessio Lomuscio and Lalit Maganti. "An approach to reachability analysis for feed-forward ReLU neural networks". In: *arXiv:1706.07351* (2017).

[111] Jonathan Long, Evan Shelhamer, and Trevor Darrell. “Fully convolutional networks for semantic segmentation”. In: *Conference on Computer Vision and Pattern Recognition* (2015).

[112] Jiasen Lu, Caiming Xiong, Devi Parikh, and Richard Socher. “Knowing when to look: Adaptive attention via a visual sentinel for image captioning”. In: *Conference on Computer Vision and Pattern Recognition* (2017).

[113] Wenjie Luo, Alexander G Schwing, and Raquel Urtasun. “Efficient deep learning for stereo matching”. In: *Conference on Computer Vision and Pattern Recognition* (2016).

[114] Xingjun Ma, Bo Li, Yisen Wang, Sarah M Erfani, Sudanthi Wijewickrema, Grant Schoenebeck, Dawn Song, Michael E Houle, and James Bailey. “Characterizing adversarial subspaces using local intrinsic dimensionality”. In: *International Conference on Learning Representations* (2018).

[115] Andrew L Maas, Raymond E Daly, Peter T Pham, Dan Huang, Andrew Y Ng, and Christopher Potts. “Learning word vectors for sentiment analysis”. In: *Annual Conference of the North American Chapter of the Association for Computational Linguistics* (2011).

[116] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. “Towards deep learning models resistant to adversarial attacks”. In: *International Conference on Learning Representations* (2018).

[117] Lalit Maganti. “Verification of Recurrent Neural Networks”. In: (2017).

[118] João P Marques-Silva and Karem A Sakallah. “GRASP: A search algorithm for propositional satisfiability”. In: *IEEE Transactions on Computers* (1999).

[119] Catherine A Meadows. “Formal verification of cryptographic protocols: A survey”. In: *International Conference on the Theory and Application of Cryptology* (1994).

[120] Jan Hendrik Metzen, Tim Genewein, Volker Fischer, and Bastian Bischoff. “On detecting adversarial perturbations”. In: *International Conference on Learning Representations* (2017).

[121] Matthew Mirman, Timon Gehr, and Martin Vechev. “Differentiable abstract interpretation for provably robust neural networks”. In: *International Conference on Machine Learning* (2018).

[122] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. “Human-level control through deep reinforcement learning”. In: *Nature* (2015).

[123] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. “Deepfool: a simple and accurate method to fool deep neural networks”. In: *Conference on Computer Vision and Pattern Recognition* (2016).

[124] Yannick Moy, Emmanuel Ledinot, Hervé Delseny, Virginie Wiels, and Benjamin Monate. “Testing or formal verification: Do-178c alternatives and industrial experience”. In: *IEEE software* (2013).

[125] Nina Narodytska, Shiva Prasad Kasiviswanathan, Leonid Ryzhyk, Mooly Sagiv, and Toby Walsh. “Verifying properties of binarized deep neural networks”. In: *arXiv:1709.06662* (2017).

[126] Maria-Irina Nicolae, Mathieu Sinn, Minh Ngoc Tran, Beat Buesser, Ambrish Rawat, Martin Wistuba, Valentina Zantedeschi, Nathalie Baracaldo, Bryant Chen, Heiko Ludwig, Ian Molloy, and Ben Edwards. "Adversarial robustness toolbox v1.0.1". In: *arxiv:1807:01069* (2018).

[127] Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. "Wavenet: A generative model for raw audio". In: *arXiv preprint arXiv:1609.03499* (2016).

[128] Nicolas Papernot, Ian Goodfellow, Ryan Sheatsley, Reuben Feinman, and Patrick McDaniel. "Cleverhans v2.0.0: An adversarial machine learning library". In: *arXiv:1610.00768* (2016).

[129] Nicolas Papernot, Patrick McDaniel, and Ian Goodfellow. "Transferability in machine learning: from phenomena to black-box attacks using adversarial samples". In: *arXiv:1605.07277* (2016).

[130] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z Berkay Celik, and Ananthram Swami. "Practical black-box attacks against machine learning". In: *ACM Asia conference on computer and communications security* (2017).

[131] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. "Automatic differentiation in PyTorch". In: *NIPS Autodiff Workshop* (2017).

[132] Jonathan Peck, Joris Roels, Bart Goossens, and Yvan Saeys. "Lower bounds on the robustness to adversarial perturbations". In: *Neural Information Processing Systems* (2017).

[133] Amir Pnueli. "The temporal logic of programs". In: *Symposium on Foundations of Computer Science* (1977).

[134] Luca Pulina and Armando Tacchella. "An abstraction-refinement approach to verification of artificial neural networks". In: *International Conference on Computer-Aided Verification* (2010).

[135] Chongli Qin, Brendan O'Donoghue, Rudy Bunel, Robert Stanforth, Sven Gowal, Jonathan Uesato, Grzegorz Swirszcz, Pushmeet Kohli, et al. "Verification of non-linear specifications for neural networks". In: *International Conference on Learning Representations* (2019).

[136] Aditi Raghunathan, Jacob Steinhardt, and Percy Liang. "Certified defenses against adversarial examples". In: *International Conference on Learning Representations* (2018).

[137] Aditi Raghunathan, Jacob Steinhardt, and Percy S Liang. "Semidefinite relaxations for certifying robustness to adversarial examples". In: *Neural Information Processing Systems* (2018).

[138] Steven J Rennie, Etienne Marcheret, Youssef Mroueh, Jarret Ross, and Vaibhava Goel. "Self-critical sequence training for image captioning". In: *Conference on Computer Vision and Pattern Recognition* (2017).

[139] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. "Semantically equivalent adversarial rules for debugging NLP models". In: *Annual Meeting of the association for Computational Linguistics* (2018).

[140] Wenjie Ruan, Xiaowei Huang, and Marta Kwiatkowska. “Reachability analysis of deep neural networks with provable guarantees”. In: *International Joint Conference on Artificial Intelligence* (2018).

[141] Vicenc Rubies-Royo, Roberto Calandra, Dusan M Stipanovic, and Claire Tomlin. “Fast neural network verification via shadow prices”. In: *arXiv:1902.07247* (2019).

[142] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. “ImageNet Large Scale Visual Recognition Challenge”. In: *International Journal of Computer Vision* (2015).

[143] Hadi Salman, Greg Yang, Huan Zhang, Cho-Jui Hsieh, and Pengchuan Zhang. “A convex relaxation barrier to tight robustness verification of neural networks”. In: *Neural Information Processing Systems* (2019).

[144] Hadi Salman, Greg Yang, Jerry Li, Pengchuan Zhang, Huan Zhang, Ilya Razenshteyn, and Sebastien Bubeck. “Provably robust deep learning via adversarially trained smoothed classifiers”. In: *Neural Information Processing Systems* (2019).

[145] Rico Sennrich, Barry Haddow, and Alexandra Birch. “Neural machine translation of rare words with subword units”. In: *Annual Meeting of the association for Computational Linguistics* (2016).

[146] Ravi Shekhar, Sandro Pezzelle, Yauhen Klimovich, Aurélie Herbelot, Moin Nabi, Enver Sangineto, and Raffaella Bernardi. “FOIL it! Find one mismatch between image and language caption”. In: *arxiv:1705.01359* (2017).

[147] Hanif D Sherali and Warren P Adams. “A hierarchy of relaxations and convex hull characterizations for mixed-integer zero—one programming problems”. In: *Discrete Applied Mathematics* (1994).

[148] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. “Mastering the game of Go with deep neural networks and tree search”. In: *Nature* (2016).

[149] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. “A general reinforcement learning algorithm that masters chess, shogi, and go through self-play”. In: *Science* (2018).

[150] Gagandeep Singh, Timon Gehr, Matthew Mirman, Markus Püschel, and Martin Vechev. “Fast and effective robustness certification”. In: *Neural Information Processing Systems* (2018).

[151] Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin Vechev. “An abstract domain for certifying neural networks”. In: *Proceedings of the ACM on Programming Languages* (2019).

[152] Gagandeep Singh, Rupanshu Ganvir, Markus Püschel, and Martin Vechev. “Beyond the single neuron convex barrier for neural network certification”. In: *Neural Information Processing Systems* (2019).

[153] Yang Song, Taesup Kim, Sebastian Nowozin, Stefano Ermon, and Nate Kushman. “Pixeldefend: Leveraging generative models to understand and defend against adversarial examples”. In: *International Conference on Learning Representations* (2018).

[154] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. "Sequence to sequence learning with neural networks". In: *Neural Information Processing Systems* (2014).

[155] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. "Intriguing properties of neural networks". In: *International Conference on Learning Representations* (2014).

[156] Gerald Tesauro. "Temporal difference learning and TD-Gammon". In: *Communications of the ACM* (1995).

[157] Vincent Tjeng, Kai Xiao, and Russ Tedrake. "Evaluating robustness of neural networks with mixed integer programming". In: *International Conference on Learning Representations* (2019).

[158] Dimitris Tsipras, Shibani Santurkar, Logan Engstrom, Alexander Turner, and Aleksander Madry. "Robustness may be at odds with accuracy". In: *International Conference on Learning Representations* (2019).

[159] Yusuke Tsuzuku, Issei Sato, and Masashi Sugiyama. "Lipschitz-margin training: Scalable certification of perturbation invariance for deep neural networks". In: *Neural Information Processing Systems* (2018).

[160] Jonathan Uesato, Brendan O'Donoghue, Aaron van den Oord, and Pushmeet Kohli. "Adversarial risk and the dangers of evaluating against weak attacks". In: *International Conference on Machine Learning* (2018).

[161] Subhashini Venugopalan, Marcus Rohrbach, Jeffrey Donahue, Raymond Mooney, Trevor Darrell, and Kate Saenko. "Sequence to sequence-video to text". In: *Conference on Computer Vision and Pattern Recognition* (2015).

[162] Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. "Show and tell: A neural image caption generator". In: *Conference on Computer Vision and Pattern Recognition* (2015).

[163] Chenglong Wang, Rudy Bunel, Dvijotham Krishnamurthy, Po-Sen Huang, Edward Grefenstette, and Pushmeet Kohli. "Knowing when to stop: Evaluation and verification of conformity to output-size specifications". In: *Conference on Computer Vision and Pattern Recognition* (2019).

[164] Jingwen Wang, Wenhao Jiang, Lin Ma, Wei Liu, and Yong Xu. "Bidirectional attentive fusion with context gating for dense video captioning". In: *Conference on Computer Vision and Pattern Recognition* (2018).

[165] Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. "Efficient formal safety analysis of neural networks". In: *Neural Information Processing Systems* (2018).

[166] Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. "Formal security analysis of neural networks using symbolic intervals". In: *27th {USENIX} Security Symposium ({USENIX} Security 18)* (2018).

[167] Stefan Webb, Tom Rainforth, Yee Whye Teh, and M Pawan Kumar. "A statistical approach to assessing neural network robustness". In: *International Conference on Learning Representations* (2019).

[168] Tsui-Wei Weng, Huan Zhang, Pin-Yu Chen, Jinfeng Yi, Dong Su, Yupeng Gao, Cho-Jui Hsieh, and Luca Daniel. "Evaluating the robustness of neural networks: An extreme value theory approach". In: *International Conference on Learning Representations* (2018).

[169] Tsui-Wei Weng, Huan Zhang, Hongge Chen, Zhao Song, Cho-Jui Hsieh, Duane Boning, Inderjit S Dhillon, and Luca Daniel. "Towards fast computation of certified robustness for relu networks". In: *International Conference on Machine Learning* (2018).

[170] Wayne L Winston and Munirpallam Venkataramanan. *Introduction to Mathematical Programming*. Vol. 1. Thomson Learning, 2002.

[171] Eric Wong and Zico Kolter. "Provable defenses against adversarial examples via the convex outer adversarial polytope". In: *International Conference on Machine Learning* (2018).

[172] Eric Wong, Frank Schmidt, Jan Hendrik Metzen, and J. Zico Kolter. "Scaling provable adversarial defenses". In: *Neural Information Processing Systems* (2018).

[173] Weiming Xiang, Hoang-Dung Tran, and Taylor T Johnson. "Output reachable set estimation and verification for multi-layer neural networks". In: *IEEE Transactions on Neural Networks and Learning Systems* (2017).

[174] Weiming Xiang, Hoang-Dung Tran, and Taylor T Johnson. "Reachable set computation and safety verification for neural networks with ReLU activations". In: *American Control Conference* (2018).

[175] Kai Y Xiao, Vincent Tjeng, Nur Muhammad Shafiullah, and Aleksander Madry. "Training for faster adversarial robustness verification via inducing relu stability". In: *International Conference on Learning Representations* (2019).

[176] Cihang Xie, Jianyu Wang, Zhishuai Zhang, Zhou Ren, and Alan Yuille. "Mitigating adversarial effects through randomization". In: *International Conference on Learning Representations* (2018).

[177] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron C. Courville, Ruslan Salakhutdinov, Richard S. Zemel, and Yoshua Bengio. "Show, attend and tell: Neural image caption generation with visual attention". In: *International Conference on Machine Learning* (2015).

[178] Weilin Xu, David Evans, and Yanjun Qi. "Feature squeezing mitigates and detects Carlini/Wagner adversarial examples". In: *arXiv:1705.10686* (2017).

[179] Puyudi Yang, Jianbo Chen, Cho-Jui Hsieh, Jane-Ling Wang, and Michael I Jordan. "Greedy attack and Gumbel attack: Generating adversarial examples for discrete data". In: *arXiv preprint arXiv:1805.12316* (2018).

[180] Li Yao, Atousa Torabi, Kyunghyun Cho, Nicolas Ballas, Christopher Pal, Hugo Larochelle, and Aaron Courville. "Describing videos by exploiting temporal structure". In: *Proceedings of the IEEE international Conference on Computer Vision* (2015).

[181] Haonan Yu, Jiang Wang, Zhiheng Huang, Yi Yang, and Wei Xu. "Video paragraph captioning using hierarchical recurrent neural networks". In: *Conference on Computer Vision and Pattern Recognition* (2016).

[182] Xiaoyong Yuan, Pan He, Qile Zhu, and Xiaolin Li. “Adversarial examples: Attacks and defenses for deep learning”. In: *IEEE transactions on neural networks and learning systems* (2019).

[183] Radosiaw R Zakrzewski. “Verification of a trained neural network accuracy”. In: *IJCNN* (2001).

[184] Huan Zhang, Tsui-Wei Weng, Pin-Yu Chen, Cho-Jui Hsieh, and Luca Daniel. “Efficient neural network robustness certification with general activation functions”. In: *Neural Information Processing Systems* (2018).

[185] Huan Zhang, Pengchuan Zhang, and Cho-Jui Hsieh. “RecurJac: An efficient recursive algorithm for bounding jacobian matrix of neural networks and its applications”. In: *AAAI Conference on Artificial Intelligence* (2019).