Automated machine learning for neural network verification

König, H.M.T.

Chapter 2

Preliminaries

In this chapter, we will introduce the most important concepts and methods on which this thesis are built, including adversarial attacks, existing approaches to formal verification as well as meta-algorithmic techniques, such as automated algorithm configuration and running time prediction. Notice that the research articles presented in Chapter 1.2 are, in part, based in the content of this chapter.

2.1 Adversarial Attacks

In recent years, deep learning methods based on neural networks have been increasingly adopted within various safety-critical domains and use contexts, ranging from manoeuvre advisory systems in unmanned aircraft to face recognition systems in mobile phones ([50, 2, 53]). Concurrently, it is now well known that neural networks are vulnerable to *adversarial attacks* [101], where a given input is manipulated, often in subtle ways, such that it is misclassified by the network.

Adversarial examples or negatives are network inputs that are indistinguishable from regular inputs, but cause the network to produce misclassifications [101]. These adversarial examples can be created by applying a hardly perceptible perturbation to the original instance that maximises model error while staying close to the initial example. The most prevalent distance metrics used to evaluate adversarial distortion are the l_1 [13, 16], l_2 [101] and l_∞ [37, 89] norms. Prominent methods for creating adversarial examples include *projected gradient descent* (PGD) [77] and *fast gradient sign method* (FGSM) [37], which, in essence, seek to identify the input pixels that have the largest impact on the model prediction, and modify those until the model produces

a misclassification. Notice that adversarial examples could also occur due to natural distortions, such as signal noise or changing conditions in the environment.

In the case of image recognition tasks, the perturbation required to trigger a misclassification, whether it is adversarially crafted or arises accidentally, can be so small that it remains virtually undetectable to the human eye.

2.2 Formal Neural Network Verification

Various methods have been proposed to establish the robustness of neural networks against adversarial attacks. Some of these methods perform *heuristic* attacks, where adversarial examples are found using approximate, iterative strategies to solve the underlying optimisation problem [37, 67, 14]; however, these methods are empirical in the sense that they do not paint a full picture of the robustness of a given network to adversarial attacks, as one defense mechanism might still be circumvented by another, possibly new class of attacks.

In light of this, several approaches have been developed to more thoroughly verify neural networks [94, 6, 27, 55, 26, 35, 115, 12, 104, 8]. These *formal* verification methods can assess the robustness of a given network in a principled fashion, which means that they yield provable guarantees on certain properties of input-output combinations. For a classifier, a property can be that instances, which are in close distance to a certain input x, belong to the same class as x.

This specific type of assessment refers to *local robustness verification*, a broadly studied verification task, in which a network is systematically tested against various input perturbations under pre-defined norms, where the most commonly considered norm is the l_∞-norm [37, 89], representing the maximum discrepancy between two inputs. The l_∞-norm is most suitable for adversarial testing, as it ensures that perturbations are minimally perceptible yet uniformly applied across all input dimensions. It should be noted that recent work has proposed to move beyond pixel-based perturbations, which account for realistic sensor errors, to *semantic* perturbations; *i.e.*, linear transformations representing changes in contrast, luminosity, scaling, rotation, and other factors [83, 39]. This specific type of verification, though, falls outside the scope of this thesis.

In general, formal verification algorithms can be characterised by three criteria: *soundness, completeness* and computational cost. A sound algorithm will only report that a property holds if the property actually holds. An algorithm that is complete will correctly state that a property holds whenever it holds. While it is favourable to produce verifiers that can certify every given instance in a dataset, there is a trade-off

between the completeness of a verification algorithm and its scalability in terms of computational complexity. The neural network verification problem is highly complex and has been proven to be NP-complete [55]. This complexity arises from the need to determine whether any input from a given domain can produce an output that violates specified constraints, which translates into solving a non-convex optimization problem due to the presence of non-linear activation functions in the neural network. Therefore, it is not surprising that for large networks and/or instance sets the problem quickly becomes practically intractable.

Consequently, some verification algorithms forego completeness to improve computational efficiency by resorting to approximations [6, 26, 35, 115, 12]. These approximations, however, do not always return the actual solution to a given verification problem but can result in mismatches or cases where the solution remains unknown. Other incomplete methods seek to add random noise to the inputs during training to smooth a neural network classifier and then derive the certified robustness for this smoothed classifier [69, 19]. While these approaches scale to larger network architectures, their robustness guarantees remain probabilistic, *i.e.*, the method estimates the probability that the predicted label remains most likely even under small perturbations to the input. In contrast to incomplete verification methods, however, this does not provide sound verification, as there is no formal guarantee that the robustness property actually holds. Furthermore, randomised smoothing has been found to come at the cost of classifier accuracy [82]. As can be seen from this example, increased scalability of a verification method usually comes at the cost of performance loss in other areas.

2.2.1 Problem Definition

Formally, the local robustness verification problem we study in the remainder of this thesis can be described as follows. Consider a feed-forward neural network classifier comprising n layers with a k_0-dimensional input and a k_n-dimensional output. Let $\boldsymbol{x} \in \mathcal{D}_{\boldsymbol{x}} \subseteq \mathbb{R}^{k_0}$ denote the input and $\boldsymbol{y} \in \mathcal{D}_{\boldsymbol{y}} \subseteq \mathbb{R}^{k_n}$ denote the class label associated with that input (in a certain representation), where $\mathcal{D}_{\boldsymbol{x}}$ and $\mathcal{D}_{\boldsymbol{y}}$ represent the domains of possible values for $\boldsymbol{x}$ and $\boldsymbol{y}$, respectively. The neural network represents a function $\boldsymbol{f}$ that aims to model the relation between input $\boldsymbol{x}$ and the associated class label $\boldsymbol{y}$, expressed as $\hat{\boldsymbol{y}} = \boldsymbol{f}(\boldsymbol{x})$, where $\hat{\boldsymbol{y}}$ represents the output of the neural network based on input $\boldsymbol{x}$. Note that the model can also produce misclassifications, in those cases $\hat{\boldsymbol{y}} \neq \boldsymbol{y}$.

To address the verification problem, we need to validate the specified conditions for input-output relationships of $\boldsymbol{f}$. In the context of local robustness verification for

image classification networks, the goal is to determine whether input instances within distance ϵ of a specific input $\boldsymbol{x_0}$ belong to the same output class as $\boldsymbol{x_0}$. This problem can be formulated as follows (based on [73]):

$$\forall \boldsymbol{x} : \|\boldsymbol{x} - \boldsymbol{x}_0\|_p \leq \epsilon \implies \boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{f}(\boldsymbol{x}_0)$$

Here, $\boldsymbol{x}$ represents a possible input, for which we can compute the distance to $\boldsymbol{x_0}$ based on an ℓ_p distance metric. If the neural network always predicts correctly within the pre-defined distance ϵ, the network is verified to be robust with respect to the verification property. Otherwise, $\boldsymbol{x}$ has been found to be an adversarial example, rendering the neural network unsafe, *i.e.*, non-robust.

2.2.2 Incomplete Verification

The most scalable and efficient incomplete verification approaches attempt to produce sound bounds for the output nodes of a given network by identifying linear relationships between each hidden or input neuron and the output neurons. Therefore, in the case of ReLU activation functions, each ReLU neuron whose inputs span over both the positive and negative domain must be relaxed, which typically involves approximating the ReLU function by means of linear bounds; see *e.g.*, [119]. These neurons are referred to as being *unstable*. There also exist relaxations for other commonly used non-linearities, such as sigmoid or hyperbolic tangent [65, 119].

To find a linear relationship between the input and the output of a network, the output bounds of the last layer are back-propagated, such that the input bounds of each layer are replaced recursively by the output bounds from the previous layer [99, 119]. These techniques are referred to as *linear bound propagation* (LBP) methods. Once the linear connections have been established, the easier linear optimisation problem can be solved to obtain the final bounds of the output layer using Linear Programming (LP) solvers. Alternatively, it has been proposed to employ *symbolic interval propagation* (SIP) to calculate linear bounding equations of the output [8, 110]. In comparison to LBP techniques, SIP-based approaches propagate input variables symbolically and, thus, are able to identify inter-dependencies between them, which may lead to overall tighter bounds.

Finally, adversarial attacks can also be considered an incomplete verification approach, as the existence of an adversarial example proves the violation of the given property [37].

2.2.3 Complete Verification

Early work on complete verification of neural networks utilised *satisfiability modulo theories (SMT)* solvers [55, 56, 90, 91, 92], which determine whether a set of logic constraints is satisfiable [84]. The resulting verification problems are NP-complete and challenging to solve in practice. Some SMT-based verification algorithms, such as those proposed by Katz et al. [55, 56], employ the well-known simplex algorithm [21] for assigning values to the SMT variables.

Alternatively, it is possible to formulate the verification task as a constraint optimisation problem using *mixed integer programming (MIP)* [8, 25, 75, 104]. MIP solvers essentially optimise an objective function subject to a set of constraints. Generally, optimisation problems are well studied, and by approaching verification tasks from that angle, techniques and insights from well-developed areas of computer science and operations research can be leveraged. MIP-based verification algorithms assign variables to each node in the network and, more specifically, encode non-linearities by means of binary variables indicating whether a node is in an active or inactive state; for ReLU nodes, this means whether a the pre-activation value of the node exceeds a value of zero. Approaches differ in the way perturbations are encoded into the program as well as in the specification of their objective function. MIP problems, similar to SMT problems, can be challenging to solve and tend to be computationally expensive (in terms of CPU time and memory). Further details on MIP-based verification will be provided in Chapter 4.

To overcome the computational complexity of SMT and MIP, it has been proposed to use the well-known *branch-and-bound* algorithm [68] for solving the verification problem, regardless of whether it is modelled as MIP or SMT [11, 12, 22, 27, 109]. Neural network verification algorithms based on branch-and-bound consist of two main steps: (*i*) branching and (*ii*) bounding. Branching involves splitting the domain of one or more variables (based on the nodes in the network) into sub-problems of the original problem; for instance, a ReLU node can be split into its positive and negative domain, which creates new sub-problems for each of these cases, respectively. These (relaxed) sub-problems are then solved by cheap but incomplete verification algorithms, such as LBP techniques, which determine a lower bound to the verification problem, while upper bounds are found via falsification algorithms [24]. By repeating these steps, the bounds, *i.e.*, the upper and lower limits on the possible value of a solution to the verification problem, are tightened in each iteration. There exist many different branching schemes and bounding algorithms, which vary in tightness of the bounds

and performance in terms of running time.

To formulate the constraints used in the above-mentioned methods, the non-linear activation functions of a neural network are usually relaxed. This is mostly done by approximating the original non-linear activation function by at least two linear functions, forming upper and lower bounds [27, 99, 112, 113, 119]. Employing the linear bounds as relaxation to the activation function increases the feasible region of each variable in the model, and as the nodes in each layer are dependent on the previous layer, the bounds on each consecutive layer become looser. The approximation, thus, provides a trade-off, as loose and fast bounds lead to large feasible regions while obtaining tight bounds tends to be computationally expensive. The way in which non-linearities are approximated presents a key distinguishing factor between complete verification algorithms.

Alternatively, *symbolic interval propagation* has been proposed to compute bounds on the output range of the network for a given input and use those as additional constraints [8, 40, 110, 111]. The output range of the output layer is obtained by propagating the bounds through the network, which renders it unnecessary to encode the entire network and use computationally expensive solvers. Symbolic interval propagation lends itself as an incomplete verification method, but it can also complement complete methods, improving their efficiency by reducing the size of the feasible region of the problem, compared to the looser approximation described above.

Other bound approximation methods include *polyhedra*, *zonotope* and *star-set abstraction*. Polyhedra abstraction produces one lower bound for the approximation based on the trained network, instead of two bounds, as used in symbolic interval propagation, where the latter results in tighter bounds [99, 119]. Zonotope abstraction is similar to polyhedra abstraction and is able to model dependencies between the zonotope representation of different network layers [35, 98]. In contrast to polyhedron transformations, the zonotope transformations scale polynomially, and optimisation is efficient. Star sets are a generalisation of the zonotope abstraction, as they are not restricted to being symmetric [4]. They are similar to zonotopes; however, optimisation is less efficient, as it requires solving a linear program.

Throughout this thesis, we will use and study several verification algorithms stemming from various paradigms. Notice that we focus on complete verifiers. An overview of all considered methods as well as their reference and location in this thesis are presented in Table 2.1.

Table 2.1: Overview of verification methods considered in this thesis.

Verifier	Reference	Paradigm	Chapter
BaB	Bunel et al. [12]	Branch-and-bound	3
BaDNB	DePalma et al. [22]	Branch-and-bound	3,5
$\alpha\beta$-CROWN	Wang et al. [111]	Branch-and-bound	3,5,6
Marabou	Katz et al. [56]	SMT	3
MIPVerify	Tjeng et al. [104]	MIP	4
MN-BaB	Ferrari et al. [29]	Branch-and-bound	3
Neurify	Wang et al. [110]	SIP	3
nnenum	Bak et al. [4]	Star-set abstraction	3
Venus	Botoeva et al. [8]	MIP	4
VeriNet	Henriksen & Lomuscio [40]	SIP	3,5,6

2.3 Automated Algorithm Configuration

In general, the algorithm configuration problem can be described as follows: Given an algorithm A (also referred to as the *target algorithm*) with parameter configuration space Θ (arising from the domains of individual parameters), a set of problem instances Π and a cost metric $c : \Theta \times \Pi \to \mathbb{R}$, find a configuration $\theta^* \in \Theta$ that minimises cost c across the instances in Π:

$$\theta^* \in \arg\min_{\theta \in \Theta} \sum_{\pi \in \Pi} c(\theta, \pi) \tag{2.1}$$

The general workflow of the algorithm configuration procedure starts with picking a configuration $\theta \in \Theta$ and an instance $\pi \in \Pi$. Next, the configurator initialises a run of algorithm A with configuration θ on instance π with CPU time cutoff k and measures the resulting cost $c(\theta, \pi)$. The configurator uses this information about the performance of the target algorithm to find a configuration that performs well on the training instances. Once its configuration budget (e.g., time budget) is exhausted, it returns the current incumbent, *i.e.*, the best configuration found so far. Finally, when running the target algorithm with configuration θ^*, it should result in lower cost (such as average running time) across the benchmark set.

Automated algorithm configuration has been shown to work effectively in the context of SAT solving [43, 47], scheduling [18], mixed-integer programming [44, 76], answer set solving [34], AI planning [107] and machine learning [103, 31].

2.4 Algorithm Portfolios & Selection

In cases where the performance of an algorithm varies greatly from one instance to another and where the performance of several different algorithms (or algorithm configurations) complements each other across an instance distribution, one can make use of *algorithm portfolios* [36, 42]. Such portfolios combine multiple algorithms (or algorithm configurations) in such a way that a much broader range of performance characteristics can be exploited, compared to running a single algorithm on its own. Added this paragraph with formal description.

A widely recognised approach for portfolio construction is Hydra [116], which has been shown to work effectively, *e.g.*, in the context of MIP-based neural network verification [61]. Hydra employs a greedy algorithm that scores a configuration based on its actual performance if it surpasses the portfolio on the current instance; otherwise, it assigns the portfolio's performance cost. Consequently, potential configurations are evaluated solely based on their contribution to improving the portfolio, guiding the configurator to prioritise instances on which the current portfolio underperforms. The portfolio construction proceeds as follows: Let P_i denote the portfolio after iteration i, starting with $P_0 := \{\}$, which is the empty portfolio. In each iteration i, the configurator identifies a new configuration θ_i, which is then added to the portfolio. In the first iteration, this results in $P_1 := \{\theta_1\}$. From the second iteration onward, the performance metric is adjusted such that if the incumbent configuration underperforms the portfolio on a specific instance, its score is replaced with the portfolio's performance. Following each iteration, the newly generated configuration θ_i is evaluated, and the portfolio is updated according to a predefined portfolio updating strategy, yielding $P_i = P_{i-1} \cup \{\theta_i\}$. Notice that this framework can be extended to allow adding multiple configurations per iteration, however, this requires adjustments to the performance metric and the updating strategy.

Algorithm portfolios can employ all algorithms in a parallel fashion or, alternatively, provide the basis for per-instance algorithm selection mechanisms. The latter are based on instance-specific features, which are used to train a statistical model subsequently used for selecting the algorithm to be run on a given problem instance, *e.g.*, based on performance predictions for each individual algorithm in the portfolio [118]. In the former case, all algorithms are run in parallel on a given problem instance, and the portfolio terminates once one algorithm has returned a solution. This implicitly ensures that we always benefit from the best-performing algorithm in the portfolio; however, it comes at the cost of increased use of parallel resources when compared to

per-instance selection from a portfolio of algorithms. Thus, when evaluating a parallel algorithm portfolio, it is important to ensure that all algorithms together do not exceed the global computing budget used by a single baseline algorithm.

2.5 Running Time Prediction

In the context of NP-complete problems, such as SAT, MIP or TSP, it has been shown that running time can vary drastically depending on the instance and algorithm used [48]. While there is only very little understanding of the reasons for this behaviour, it is possible to predict running times of previously unseen SAT, MIP and TSP problem instances reasonably well based on cheaply computable instance features by fitting a statistical model on the running time of a given algorithm [48]. Running time prediction has various applications that seek to minimise costs and improve the efficiency of machine learning systems; those range from algorithm selection (as mentioned in the previous section) to scheduling [58]. Furthermore, running time prediction provides insights into the relationship between instance characteristics and algorithm running time and, thus, informs us about instance complexity.

Following the notation of [48], we characterise a problem instance using a list of m features $\mathbf{z} = [z_1, \ldots, z_m]$, selected from a specified feature space $\mathcal{F}$. These features are typically computed by specialised code to extract relevant characteristics for any given problem instance. In the context of neural network verification, such features could be, for example, the number of unstable nodes or the prediction margin.

Consider $\mathbb{R}$ the space of real numbers indicating an algorithm's performance measure, such as the running time in seconds on a given machine. For an algorithm A and a distribution of instances described by a feature space $\mathcal{F}$, we are interested in modelling the running time as a function $f : \mathcal{F} \to \mathbb{R}$ that maps the feature vector $\mathbf{z} \in \mathcal{F}$ to the performance measure of the algorithm.

To model the performance of an algorithm A on an instance set Π, we run A on all instances $\pi_i \in \Pi$ and record the resulting performance values y_i. We capture the m-dimensional feature vector $\mathbf{z}_i$ of the instance used in the i-th run to be used as our vector of predictor variables. The training data for our performance model is then $\{(\mathbf{z}_1, y_1), \ldots, (\mathbf{z}_n, y_n)\}$, where y_i (with $1 \leq i \leq n$) is the performance (*e.g.*, running time) of A on instance π_i. Lastly, we use $\mathbf{y}$ for the vector of performance values $[y_1, \ldots, y_n]$.