CHAPTER 9

Complete neural network verification

Complete verification is a hard problem due to the nonlinearity of deep neural networks. Even for a simple ReLU network, Katz et al. (2017) showed that complete verification is NP-complete. Therefore earlier works that formulate complete verification as Mixed Integer Programming (MIP) usually failed to handle networks beyond 100 neurons. However, significant progress has been made in the past few years and nowadays good verification tools can already handle some larger networks beyond 1 million parameters. In this chapter, we will discuss the main techniques used in the current complete verifiers.

9.1 Mixed integer programming

For ReLU networks, the complete verification problem can be formulated into the mixed integer programming (MIP) problem. The key insight is using binary variables to indicate the activation pattern of ReLU functions. Following the notation used in the previous sections, for a feed-forward network f we can formulate the neural network verification problem (7.8) into the following MIP problem:

$$\begin{aligned}
&\min\ c^T f \quad \text{s.t.}\ \ i \in [L],\ j \in [n_i];\\
&z^{(i)} = W^{(i)}\hat{z}^{(i-1)} + b^{(i)};\ \ f = z^{(L)};\ \ \hat{z}^{(0)} = x \in \mathcal{C};\\
&\hat{z}_j^{(i)} \geq z_j^{(i)};\ \ \hat{z}^{(i)}j \leq u_j^{(i)} s_j^{(i)};\ \ \hat{z}_j^{(i)} \leq z_j^{(i)} - l_j^{(i)}(1 - s_j^{(i)});\\
&\hat{z}_j^{(i)} \geq 0;\ \ z_j^{(i)} \in [l_j^{(i)}, u_j^{(i)}];\ \ s_j^{(i)} \in \{0, 1\};
\end{aligned} \tag{9.1}$$

where $s_j^{(i)}$ indicates the two statuses of ReLU: (1) *inactive*: when $s_j^{(i)} = 0$, constraints on $z_j^{(i)}$ simplify to $\hat{z}_j^{(i)} = 0$; or (2) *active*: when $s_j^{(i)} = 1$, we have $\hat{z}_j^{(i)} = z_j^{(i)}$. Here $l_j^{(i)}$ and $u_j^{(i)}$ are precomputed intermediate lower and upper bounds on preactivation $z_j^{(i)}$ such that $l_j^{(i)} \leq z_j^{(i)}(x) \leq u_j^{(i)}$ for $x \in \mathcal{C}$. Note that (9.1) requires exponential time to solve exactly due to the integer variable $s_i^{(j)}$ (activation pattern of each ReLU), but meanwhile there exists software packages such as Gurobi, which can handle small-scale instances relatively well. For example, for problems with tens of neurons, it is still possible to directly apply Gurobi to conduct complete verification. Unfortunately, the complexity of this problem can increase exponentially with the number of ReLU neurons, so it can take hours to run even on a small network, unless the network is trained with a strong regularization such as a certified dense (Wong and Kolter, 2018; Xiao et al., 2019b).

9.2 Branch and bound

To speed up complete verification, most of the modern algorithms adopt the Branch-and-Bound (BaB) method (Bunel et al., 2018). Note that BaB is also a crucial component in off-the-shelf MIP solvers, and by developing a specialized BaB procedure for verification can improve the efficiency. BaB-based verification framework is a recursive process consisting of two main steps, branching and bounding. For *branching*, BaB-based methods will divide the bounded input domain $\mathcal{C}$ into subdomains $\{\mathcal{C}_i | \mathcal{C} = \bigcup_i \mathcal{C}_i\}$, each of these domains corresponds to a new independent verification problem. For instance, we can split a ReLU unit $\hat{z}^{(k)} = \text{ReLU}(z^{(k)})$ to negative and positive cases as $\mathcal{C}_0 = \{x \mid z^{(k)} \geq 0\}$ and $\mathcal{C}_1 = \{x \mid z^{(k)} < 0\}$ for a ReLU-based network, where each of these domains are the set of input that satisfies a certain constraint. For each subdomain $\mathcal{C}_i$, BaB-based methods perform *bounding* to obtain a relaxed but sound lower bound $\underline{f}_{\mathcal{C}_i}$. A tightened global lower bound over $\mathcal{C}$ can then be obtained by taking the minimum values of the subdomain lower bounds from all the subdomains: $\underline{f} = \min_i \underline{f}_{\mathcal{C}_i}$.

Based on this splitting approach, BaB will be performed recursively to tighten the approximated global lower bound. This can be implemented into a tree structure, where each node corresponds to a subdomain C. Starting from the root note where C is the entire input domain (e.g., an ℓ_p ball around some data point x_0), BaB splits the node(domain) based on the sign of one ReLU neuron. After the split, we will create two children for a node, each of them consists a subdomain, and compute the lower bound for the subdomain. This is done until either (i) the global lower bound $\underline{f}$ becomes larger than 0 and thus the property is proven or (ii) a violation (e.g., identification of an adversarial example) is located in a subdomain to disprove the property.

Soundness of BaB. We say that the verification process is sound if we can always trust the "yes" (the property is successfully verified) answer given by the verifier. When a BaB-based verification process outputs "yes", it means all the leaf nodes output "yes". Therefore, we can easily see that if the bounding method used for each subdomain C_i is sound, then the whole BaB-based verifier will also be sound.

Completeness of BaB. In a BaB-based verifier, it will keep splitting nodes until all the nodes are proved or disapproved. To guarantee the completeness of the procedure, we have to ensure that the bounding method will become a complete verifier for each subdomain after splitting all the ReLU neurons. This is true when applying the LP-based bounding method or linear bound propagation methods to each subdomain.

Bounding with Linear Programming (LP). For neural network verification, a classical bounding method used in BaB-based verification is the *Linear Programming bounding procedure*. Specifically, one can transform the original verification problem into a linear programming problem. This is done by relaxing each ReLU activation function $\hat{z} = \max(z, 0)$ into linear constraints. If the ReLU neuron has been splitted, then it has already been replaced by a linear constraint (either $z \le 0, \hat{z} = 0$ or $z > 0, \hat{z} = z$). For each unsplitted neurons, it can be relaxed into the following three constraints:

$$\hat{z} \ge 0 \tag{9.2}$$

$$\hat{z} \ge z \tag{9.3}$$

$$\hat{z} \le \frac{u}{u-l}(z-l), \tag{9.4}$$

where u, l are the lower and upper bound of z, which can be computed by a simple forward bound propagation or using the linear propagation-based method mentioned in the previous section. Eq (9.4) is known as the "triangle relaxation" for ReLU neurons, since it essentially bounds the ReLU by a convex triangle region. After doing this for all the unsplitted neurons, the verification problem will become a linear programming problem and can thus be solved by off-the-shelf LP solvers. Using the above mentioned LP-based bounding method with BaB can lead to a complete verifier, since when all the ReLU neurons are splitted, there will be no relaxation to the original problem, and the LP will get the exact solution of the verification problem for the given subdomain.

Branching in BaB. Since the branching step determines the shape of the search tree, the main challenge is to efficiently choose a good leaf to split, which can significantly reduce the total number of branches and running time. For example, (Bunel et al., 2018) includes a simple branching heuristic, which assigns each ReLU node a score to estimate the improvement for tightening $\underline{f}$ by splitting it, and splits the node with the highest score.

Different implementations usually use slightly different branching heuristics.

9.3 Branch-and-bound with linear bound propagation

Although linear programming (LP) can provide tight bounds for each subdomain when conducting BaB, they are usually time consuming to solve and cannot fully exploit the parallelization provided by GPUs or TPUs. Therefore many recent works applied incomplete verification solvers mentioned in the previous chapter to efficiently rule out some regions and provide guidance to BaB (Bunel et al., 2020a; Xu et al., 2021). In particular, Xu et al. (2021) combines BaB with a linear propagation-based incomplete verifier CROWN to conduct complete verification.

The main challenge of using linear propagation-based incomplete verifiers to BaB is to combine the split constraint ($z \geq 0$ or $z < 0$ for a neuron z) into the linear propagation procedure. One simple way to do that is to just use it as a pre-activation bound for each neuron (l, u in (9.4)), but this is insufficient as the pre-activation constraint does not eliminate all the inputs outside the domain. Therefore, algorithms such as (Xu et al., 2021) incorporates the split constraints using Lagragian multipliers. More specifically, linear propagation methods lead to the following minimization problem for verification:

$$\min_{x \in C} f(x) \geq \min_{x \in C} w^T Dz,$$

where z is the values of some layer of neural network and D is the diagonal matrix derived from bounding each neurons (see Lemma 1 in Chapter 8). The split constraint can then be added into this formulation as

$$\min_{x \in C} f(x) \geq \min_{x \in C} w^T Dz, \quad Sz \leq 0, \tag{9.5}$$

where S is a diagonal matrix with $+1, -1, 0$ on the diagonal, indicating the split constraints (postive, negative or no split). Optimizing (9.5) will then lead to the optimal solution within the constraint set. To deal with the constraint minimization problem, a standard approach is to transform it into an uncinstrained problem with Lagrangian multipliers:

$$\min_{x \in C} f(x) \geq \max_{\beta \geq 0} \min_{x \in C} w^T Dz + \beta Sz, \tag{9.6}$$

where β are the multipliers. We can thus merge the two linear terms in the objective together, which reduced to the linear propagation method, with the only change being introducing a few Lagrange multipliers (β) where we need to use another gradient ascent on β to find the optimal value.

The main advantage of using linear propagation-based method in complete verification is that those methods are highly parallelizable, which can make the overall procedure efficient when running on GPU.

9.4 Empirical comparison

To give a full comparison between existing complete verification methods, we show the experimental results reported in a recent paper by Wang et al. (2021c). They evaluate verification algorithms using the benchmark containing three CIFAR-10 models (Base, Wide, and Deep) with 100 examples each. Each data example is associated with an ℓ_∞ norm perturbation budget ϵ and a target label for verification (referred to as a *property* to verify). The comparison includes the following methods:

- BaBSR (Bunel et al., 2020b), a basic BaB- and LP-based verifier;
- MIPplanet (Ehlers, 2017), a customized MIP solver for NN verification, where unstable ReLU neurons are randomly selected for splitting;
- ERAN (Singh et al., 2018a,b, 2019a,b), an abstract interpretation-based verifier, which performs well on this benchmark in VNN-COMP 2020;
- GNN-Online Lu and Kumar (2020), a BaB- and LP-based verifiers using a learned Graph Neural Network (GNN) to guide the ReLU splits;
- BDD+ BaBSR (Bunel et al., 2020a), a verification framework based on Lagrangian decomposition on GPUs (BDD+) with BaBSR branching strategy;
- OVAL (BDD+ GNN) (Bunel et al., 2020a; Lu and Kumar, 2020), a strong verifier in VNN-COMP 2020 using BDD+ with GNN guiding the ReLU splits; A.set BaBSR;
- Big-M+A.set BaBSR (De Palma et al., 2021a), which are dual-space verifiers on GPUs with a tighter linear relaxation than triangle LP relaxations;
- Fast-and-Complete (Xu et al., 2021), which uses CROWN on GPUs as the incomplete verifier in BaB without neuron split constraints;
- BaDNB (BDD+ FSB) (De Palma et al., 2021b), a complete verifier using BDD+ on GPUs with FSB branching strategy. β-CROWN BaB can use either BaBSR or FSB branching heuristic, and we include both in the evaluation.

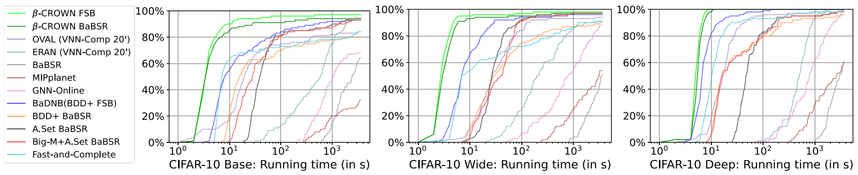

Figure 9.1 Percentage of solved properties with growing running time. β-CROWN FSB (light green (light gray in print version)) and β-CROWN BaBSR (dark green (dark gray in print version)) clearly lead in all three settings and solve over 90% properties within 10 seconds.

The experiments are run on a machine with a single NVIDIA RTX 3090 GPU (24 GB GPU memory), an AMD Ryzen 9 5950X CPU and 64 GB memory. All methods use a 1-hour timeout threshold. Fig. 9.1 plots the percentage of solved properties over time. β-CROWN FSB achieves the fastest average running time compared to all other baselines with minimal timeouts and also clearly leads on the cactus plot. When using a weaker branching heuristic, β-CROWN BaBSR still outperforms all baselines.