Automated machine learning for neural network verification

König, H.M.T.

Chapter 5

Dynamic Algorithm Termination for Branch-and-Bound-based Neural Network Verification

As mentioned in previous chapters, much recent work has been concerned with the development of more efficient verification algorithms, *e.g.*, by employing the Branch and Bound (BaB) method for solving the verification problem [22, 11, 109, 12, 27] or by tightening bounds in the problem formulation using symbolic interval propagation [8, 40, 110, 111] and abstraction [4, 99, 119, 35, 98] techniques. However, even in light of recent developments, neural network verification remains a challenging and expensive computational task, especially as network complexity and dataset size increase.

Recall that neural network verification can be divided into *local* and *global* verification [100]. As in previous chapters, we focus on local robustness verification in this study. Local robustness verification typically considers a trained neural network, along with a set of inputs and a verification property specification. Considering the computational complexity of the verification problem, the required computational resources can grow substantially with the size of the network to be verified and the set of inputs.

We propose a novel approach enabling the efficient allocation of compute resources during the verification procedure. Specifically, we introduce a method to classify verification instances as solvable or unsolvable within a predefined time budget based on cheaply computable features. Furthermore, we operationalise these predictions to

terminate the verification procedure early for instances where a solution can not be obtained within the given time budget. Thereby, we avoid spending compute resources on attempting to verify instances that ultimately do not inform us about the robustness of the network. We evaluated our approach on a broad set of state-of-the-art verification algorithms and benchmarks, including benchmarks from recent VNN competitions [10, 87, 3], and show that we can reliably terminate verification runs for instances that are unsolvable within a given cutoff time without solving considerably fewer instances overall. In summary, the contributions of this chapter are as follows:

- We present features of branch-and-bound-based neural network verification instances that enable predictions about their solubility within a given time budget;
- we introduce a novel method based on those features that reliably identifies instances that cannot be solved within a given time budget;
- we evaluate our method on a broad set of benchmarks and across multiple verification tools;
- we show how this approach can be leveraged to terminate unsolvable instances early in the verification process, leading to savings of 64% in terms of running time on average with a comparable number of solved instances relative to the current state-of-the-art approach.

5.1 Method

As previously explained, solving the neural network verification task is computationally challenging. In addition, it is not known what makes certain instances harder to solve than others. Therefore, it cannot be decided *a priori* whether an instance could be solved successfully within a given time budget, potentially leading to ineffective resource allocation; *i.e.*, allocating compute time to unsolvable instances. In light of this, we leverage running time prediction techniques to classify instances as solvable or unsolvable within a given time budget. This allows us to greatly accelerate the verification procedure, by ensuring that resources are only spent on solvable instances.

5.1.1 Problem Formulation

As we are interested in the task of terminating instances that cannot be solved within a given time budget, we consider a binary classification problem. Hence, the vector $\mathbf{y}$

introduced in Chapter 2 contains as performance measures binary variables describing whether an instance has been solved or not.

Our goal is to reduce the computational burden demanded by the verification procedure, ensuring the most effective use of resources by not spending the full budget on unsolvable instances. In addition, we need to avoid classifying solvable instances as unsolvable; otherwise, the number of certified instances would be reduced, which might lead to inaccurate conclusions about the robustness of a given neural network.

5.1.2 Dynamic Algorithm Termination for BaB-based Neural Network Verification

As explained in Chapter 2, to perform running time prediction, we need to define a feature space $\mathcal{F}$ from which we can obtain a list of features $\mathbf{z}$ characterising a problem instance Specifically, we utilise cheaply computable features that, in part, relate directly to the internal operations of the given verifier.

We distinguish between *static* and *dynamic* features, where the former are computed only once and do not change during the solving process. Examples of static features include the lower bound obtained by an incomplete verification method at the beginning of the verification process. Conversely, dynamic features aim to capture the dynamically changing state of the verification algorithm at any point in time; examples include the current number of nodes in the BaB tree and the current global bounds. Generally, static features reflect the inherent complexity of the verification instance, while dynamic features capture the progress made thus far in solving the query. A detailed discussion of the features we have developed is provided later in this section.

To best leverage the evolving nature of our dynamic features, we propose a novel method that dynamically terminates verification queries when a classification model determines that the given instance will not be solved in the remaining time budget. We give a schematic overview of our method in Algorithm 1. The procedure is parameterised by the frequency t_{freq} at which the current progress of the verification process is assessed to predict whether the given instance will be solved in the remaining time, and by the maximum allotted running time per instance, t_{cutoff}. We refer to the points in time at which the verification query is examined as a *checkpoint*. For each checkpoint, we train a classifier $\mathcal{C}_t$ with t denoting the time of the checkpoint. $\mathcal{C}_t$ is trained on the feature values of the verification instances in the training set at time t along with their corresponding label indicating whether the instances were solved within t_{cutoff} seconds. In addition, we trained the classifier on verification instances from our training set that

were successfully solved before the current checkpoint with their feature values when the verification process was completed; thereby, the classifier can learn, which feature values define a completed instance.

Furthermore, our proposed method is configurable via a confidence parameter θ, which defines the threshold that the prediction value for the positive class must exceed such that an instance is labelled accordingly. The incorporation of this parameter ensures that a user can choose whether the algorithm should stop potentially unsolvable instances as soon as possible ($\theta = 0.5$) or whether, in case of doubt, more information should be collected. The verification algorithm is then terminated only in case of a highly confident classifier prediction ($\theta = 0.99$). We note that θ can also be understood as a tuning parameter between *exploitation* and *exploration*. Therefore, θ should be chosen according to the user's needs, prioritising either a substantial reduction of the computational burden or a higher number of certified instances.

In summary, our method operates as follows. Given t_{freq}, t_{cutoff}, θ, a verification algorithm, a neural network and a training set of verification instances, we initially collect feature values for each training instance at every checkpoint t by executing the verification algorithm on each query for t_{cutoff} seconds. In addition, we record whether the instance was solved or not. Subsequently, we train a classification model C_t for every checkpoint t on the collected data. During classification, given a verification query, we start by executing the verification algorithm for t_{freq} seconds to collect an initial set of features for the given instance. Thereafter, we employ the classifier for the first checkpoint to predict whether the instance will be solved in the remaining time budget. If the confidence of this prediction exceeds θ, we terminate the verification run for the given instance and record its result as *unknown*; otherwise, we continue the verification process for t_{freq} seconds and update the dynamic instance features accordingly. Next, we query the classification model for the following checkpoint and decide whether to terminate the run. We repeat this process until the verification algorithm solves the instance under consideration or reaches the given cutoff time.

5.1.3 Static Instance Features

To perform running time prediction, we need to define instance features that allow us to make performance predictions for a given algorithm. We begin by introducing our proposed static features.

Prediction margin (Δ). This feature is defined as the difference between the two highest class scores. *i.e.*, given a neural network f with input $x_0 \in \mathcal{X}$ and corresponding

Algorithm 1 Dynamic termination for BaB-based neural network verification

Input: Verification instance (x_0, ϵ); maximum per-instance running time t_{cutoff}; dynamic termination frequency t_{freq}; set of classifiers $\mathcal{C} = \{\mathcal{C}_{t_{\text{freq}}}, \mathcal{C}_{2t_{\text{freq}}}, \ldots, \mathcal{C}_{t_{\text{cutoff}}}\}$; confidence parameter θ; verification algorithm VERIFY$((x, \epsilon), t_{\text{freq}})$ that pauses after t_{freq} seconds to return the features and result of the instance.
Output: result or *unknown*
solved, features $\leftarrow$ VERIFY$((x_0, \epsilon), t_{\text{freq}})$
$t_{\text{elapsed}} \leftarrow t_{\text{freq}}$
while $\neg$ solved **and** $t_{\text{elapsed}} < t_{\text{cutoff}}$ **do**
    if $\mathcal{C}_{t_{\text{elapsed}}}$(features) $> \theta$ **then**
        return *unknown*
    else
        solved, features $\leftarrow$ VERIFY$((x_0, \epsilon), t_{\text{freq}})$
        $t_{\text{elapsed}} \leftarrow t_{\text{elapsed}} + t_{\text{freq}}$
    end if
end while
return solved

correct label $y_0 \in \mathcal{Y}$, we have $\Delta := f_{y_0}(x_0) - \max_{y \in \mathcal{Y} \setminus y_0} f_y(x_0)$, where f_y refers to the output for class y. The prediction margin can be seen as a proxy for the closeness of the input image to the decision boundary. It heuristically captures how much change in the input space is required to change the neural network's prediction and, thus, the likelihood of an adversarial attack succeeding. This feature has recently been used in the context of adversarially robust model selection [64].

Initial Incomplete Bound. Each verifier we consider first attempts to solve the verification instance using an incomplete method. We utilised the resulting global upper and lower bounds as features.

Improved Incomplete Bound. If the initial problem bounds do not suffice to solve the problem, Oval and $\alpha\beta$-CROWN follow up with a tighter bounding method to further optimise last layer bounds. The initial and improved bounds give an estimate of how much improvement on the lower bound is realisable through (incomplete) bound optimisation methods. Furthermore, these bounds are the starting point for BaB and, thus, indicate the improvements required during BaB for solving the problem.

Initial Percentage of Safe Constraints. While VeriNet does not employ bound optimisation, the first call to the LP solver with the initial SIP bounds can already determine that some (or all) linear equations are unsatisfiable; these output constraints do then not have to be examined further during BaB. Thus, the percentage of initial safe constraints also provides an indication of the additional computation VeriNet will require subsequently.

Adversarial Attack Margin. Each of the considered verifiers initially carries out an adversarial attack that seeks to minimise the margin between the correct and incorrect classes. If the attack remains unsuccessful, its output can still be utilised to estimate the upper bound of the verification problem. Therefore, we included the adversarial attack margin, *i.e.*, the difference between the two highest scoring classes on the adversarial candidate, as an estimation of the upper bound of the given verification instance.

Number of Unstable Neurons. Lastly, we also included the absolute number of unstable neurons in our feature set. This number does not only indicate how many non-linearities have to be approximated but also bounds the maximum depth of the BaB tree.

5.1.4 Dynamic Instance Features

The *dynamic* features of BaB-based verification instances are subject to change during the BaB process, as they capture the progress made while solving the given problem instance.

Branch Characteristics. We included the number of *visited* branches that are already bounded as well as the total number of branches, also including those that have been created through branch splits but still need to be bounded. We further included the fraction of verified branches; these correspond to the leaves of the BaB tree and do not need to be split further. Once this number reaches a value of 1, the verification system has proven that the property holds.

Current Global Bounds. Furthermore, we included the current global bounds of the BaB tree. When the MIP formulation of the problem was solved by $\alpha\beta$-CROWN, we also recorded the resulting global bounds. This constitutes another way of capturing the progress of the given query, as once any global bound changes its sign, the verification process has been completed.

Depth of the BaB Tree. One important characteristic of the BaB tree that indicates instance complexity is its current depth, as it indicates how many neuron splits are present in the leaf nodes.

Number of GPU Batches. For Oval and $\alpha\beta$-CROWN, which perform the BaB algorithm in batches on a GPU, we included the number of batches that have been already computed. This feature enables a running time predictor to relate the BaB features to the internal operations of the verifiers.

Batch Computation Time. In addition, we computed the time used for the

computation of the last completed batch; this number indicates the computational hardness of the problem instance at hand, also in relation to the execution environment used for running the verifier. If feature collection occurs while a batch is still being processed, we additionally considered the computation time already spent on that batch.

5.1.5 Classification Model

For each checkpoint, we trained a random forest classifier with 200 decision trees and otherwise default hyperparameter settings. It has been shown in the past that random forests perform very well in the context of running time prediction tasks [48]. We also experimented with automatic hyperparameter configuration using *auto-sklearn* [30], but did not observe substantial improvements. Before training and classification, all features were standardised, *i.e.*, we removed the mean of each feature and scaled it to have unit variance, using the mean and standard deviation of each feature over the training set.

5.2 Experiments

We evaluated our approach on several benchmarks, which we will introduce in the following, along with details on the performance data collection and feature computation process.

Each benchmark was run on a compute cluster node equipped with two Intel Xeon Platinum 8480+ processors with 56 cores and a cache size of 105MB, 2TB of RAM and four NVIDIA H100 GPUs with 80GB of video memory, running Rocky Linux 9.4. Each run utilised 28 CPU cores, one GPU and 448GB of RAM.

5.2.1 Benchmarks

We considered a wide and diverse set of benchmarks taken from the ERAN repository [86, 99] and the VNN Competition [10, 87, 3], which have been commonly used by the neural network verification community [60, 10, 87, 3, 111, 99].

For our evaluation, we included two usage scenarios. First, we considered an approach aligned with an end-user's needs in assessing the robustness of a neural network. Here, we verified the correctly classified images from the first 1000 test set instances. We also included a competition scenario, where we generated problem instances according to the VNN Competition instance generation protocols.

In the first scenario, we included two convolutional (*Conv Big* and *Conv Small*) and two fully connected networks (*5 100* and *8 100*) trained on the MNIST dataset that were taken from the ERAN repository. For the CIFAR-10 dataset, we considered a small ResNet proposed by Wang et al. [111] (*ResNet 2B*). We verified the first 1000 test images against l_∞ perturbations with ϵ-values chosen in line with those used in previous studies [60, 111, 96].

For the second scenario, we included benchmarks directly taken from different editions of the VNN competition [10, 87, 3]. We employed the instance generation scripts provided in the competitions to generate 500 instances per benchmark that follow specific selection criteria such as correct classification or robustness against adversarial attacks. Concretely, we included the *Marabou*, *Oval21*, *SRI ResNet* and *ViT* benchmarks that consist of networks trained on the CIFAR-10 dataset. If the benchmarks included multiple networks or ϵ value specifications, we chose the configurations that yielded the most timeouts in the VNN competition, *i.e.*, the presumably most challenging problem instances.

Lastly, we included two benchmarks from the VNN Competition that consider the more complex CIFAR-100 and Tiny ImageNet datasets [87]. For both datasets, we chose the medium-size models for our evaluation. With this collection of networks and benchmarks, we ensured to include instances that have been studied extensively in the literature and that are challenging to solve by state-of-the-art verification tools.

5.2.2 Evaluation Setup

We first collected all performance data and feature values by running the verification tools and saving the result of the verification query, the consumed running time and the values of the considered instance features during the verification procedure. In Table 5.1, we report the number of solved instances and the running time for each verification tool and benchmark Missing values indicate that the benchmarks could not be used with the respective verification tools, due to unsupported network architectures.

We then evaluated our method by simulating it on the collected data. Generally, we followed a 5-fold cross validation protocol. To ensure that our training and testing sets were representative, we included in each fold the same proportion of verification instances solved before the first checkpoint, after the first checkpoint and unsolved instances; however, we only report metrics on instances that ran beyond the first checkpoint, as otherwise, we would predict timeouts after the instance has already been solved.

Benchmark	# Inst.	$\alpha\beta$-CROWN		VeriNet		Oval	
		# Solved	Time [GPU h]	# Solved	Time [GPU h]	# Solved	Time [GPU h]
5 100	960	868	31.44	580	66.65	430	90.77
8 100	947	767	41.32	501	76.64	387	94.69
Conv Big	929	918	1.50	868	11.29	842	15.34
Conv Small	980	979	1.26	931	11.96	958	6.06
ResNet 2B	703	619	15.16	576	22.72	-	-
Marabou	500	193	51.73	176	54.28	187	53.32
Oval21	500	210	50.47	158	58.45	201	52.50
ViT	500	251	41.86	-	-	-	-
SRI ResNet A	500	198	51.74	133	62.01	-	-
CIFAR-100	500	361	24.73	279	40.25	-	-
Tiny ImageNet	500	421	14.63	356	29.47	-	-

Table 5.1: Overview of benchmarks used in our evaluation, including the number of certified instances and running time for each verification tool. Instances are from the first 1000 test set images for the first 5 benchmarks and otherwise from the VNN Competition [10, 87, 3] instance selection procedure. All experiments used a per-instance timeout of 600 seconds and GPU acceleration.

We evaluated the performance of our method in terms of *accuracy*, *true positive rate* (TPR) and *false positive rate* (FPR). The TPR reflects the fraction of correctly classified timeouts out of all unsolved instances while the FPR indicates the fraction of solved instances wrongly classified as timeouts out of all solved instances. On the convolutional networks for $\alpha\beta$-CROWN, some folds did not include true negatives or true positives. If these folds were used as the holdout set, we excluded them when computing the average TPR and FPR. In addition, we also compared our method to the standard verification procedure in terms of the overall number of solved instances (including those completed before the first checkpoint) and the required running time.

To run the verification algorithms, we used the configurations provided by the respective authors for their entries in the VNN Competitions. We chose a maximum running time of 600 seconds in wall-clock time per instance ($t_{\text{cutoff}} = 600s$) and predicted whether the instance will be solved within the remaining time budget every 10 seconds ($t_{\text{freq}} = 10s$). Lastly, we set the decision threshold θ to 0.99 to ensure that our method solves as many instances as possible.

	$\alpha\beta$-CROWN			VeriNet			Oval		
Benchmark	Acc.	TPR	FPR	Acc.	TPR	FPR	Acc.	TPR	FPR
5 100	0.99	0.95	0.00	0.89	0.87	0.04	0.97	0.96	0.00
8 100	0.99	0.99	0.00	0.92	0.91	0.02	0.99	0.99	0.07
Conv Big	0.47	0.43	0.00	0.88	0.74	0.00	0.78	0.75	0.05
Conv Small	0.82	1.00	0.20	0.81	0.39	0.00	0.79	0.09	0.00
ResNet 2B	0.98	0.98	0.00	0.77	0.71	0.00	-	-	-
Marabou	0.99	0.99	0.10	0.93	0.95	0.53	0.96	0.96	0.13
Oval21	0.97	0.98	0.05	0.89	0.88	0.07	0.96	0.95	0.03
ViT	1.00	1.00	0.00	-	-	-	-	-	-
SRI ResNet A	0.99	1.00	0.02	0.91	0.90	0.00	-	-	-
CIFAR-100	0.99	1.00	0.03	0.85	0.79	0.00	-	-	-
Tiny ImageNet	0.98	0.99	0.03	0.88	0.67	0.00	-	-	-

Table 5.2: Results for timeout prediction with continuous feature collection in terms of accuracy, true positive and false positive rate as averages over five folds. We display results for $\theta = 0.99$, *i.e.*, the confidence threshold that must be reached before an instance is terminated.

5.3 Results and Discussion

In the following, we present results from our experimental evaluation of our dynamic algorithm termination method for the various verification algorithms we considered, and we show how our approach can be leveraged to allocate available resources more efficiently by terminating instances that will result in timeouts earlier in the verification process.

5.3.1 Classification Metrics

We report the classification metrics of our proposed method in Table 5.2 as averages over all five folds. We obtained very high TPR scores while maintaining a FPR close to 0 for most verifiers and benchmarks. Concretely, on average, our classifier correctly identified 85% of timeouts while incorrectly classifying 5% of solvable instances. Noticeably, across all verifiers, there were several benchmarks with TPRs above 90% and FPRs of almost zero. Lower TPR scores on the *Conv Big* and *Conv Small* benchmarks were due to the relatively small number of timeouts occurring in these benchmarks, leading to a lack of training examples for this class. Similarly, we observed higher FPR scores for the *Marabou* benchmark. This is likely due to the small number of queries solved after the first checkpoint, again resulting in less diverse training data.

	$\alpha\beta$-CROWN				VeriNet				Oval			
Benchmark	Time [GPU h]		# Solved		Time [GPU h]		# Solved		Time [GPU h]		# Solved	
5 100	21.97	(70%)	868	(±0)	18.81	(28%)	576	(-4)	7.88	(9%)	430	(±0)
8 100	17.86	(43%)	766	(-1)	16.75	(22%)	500	(-1)	3.57	(4%)	386	(-1)
Conv Big	1.01	(68%)	918	(±0)	5.48	(49%)	868	(±0)	6.36	(41%)	841	(-1)
Conv Small	1.00	(80%)	969	(-10)	11.30	(94%)	931	(±0)	6.08	(100%)	958	(±0)
ResNet 2B	4.30	(28%)	619	(±0)	10.45	(46%)	576	(±0)	-	-	-	-
Marabou	2.47	(5%)	192	(-1)	6.36	(12%)	168	(-8)	4.97	(9%)	185	(-2)
Oval21	7.57	(15%)	207	(-3)	15.04	(26%)	155	(-3)	10.02	(19%)	199	(-2)
ViT	2.00	(5%)	251	(±0)	-	-	-	-	-	-	-	-
SRI ResNet A	3.86	(7%)	197	(-1)	8.71	(14%)	133	(±0)	-	-	-	-
CIFAR-100	5.10	(21%)	360	(-1)	19.37	(48%)	279	(±0)	-	-	-	-
Tiny ImageNet	4.58	(31%)	420	(-1)	19.92	(68%)	355	(-1)	-	-	-	-

Table 5.3: Results for dynamic termination of verification queries with $\theta = 0.99$. We display the running time and the number of solved instances accumulated over five folds. In parentheses, we provide the fraction of running time used and the difference in the number of solved instances compared to the standard verification procedure.

5.3.2 Dynamic Algorithm Termination

We display the results of our method in terms of total cumulative running time and number of solved instances aggregated over all folds for each benchmark and verifier in Table 5.3.

Most importantly, we obtained substantial speed-ups, while only a small amount of solvable instances was terminated prematurely. On average, our approach solved comparably many instances in 36% of the original running time. Notably, the largest acceleration occurred on the Marabou benchmark, where up to 95% of the standard running time could be saved. However, we also observed moderate penalties in terms of the absolute difference of solved instances for the *Marabou* benchmark on VeriNet and the *Conv Small* benchmark on $\alpha\beta$-CROWN, due to the reasons stated earlier. Moreover, on several benchmarks, all solvable instances were certified using substantially reduced running time; *e.g.*, the *5 100* benchmark for Oval and $\alpha\beta$-CROWN or the *ResNet A* benchmark for VeriNet.

Overall, we found that our approach substantially improves neural network verification in terms of running time of several verification algorithms on a broad range of benchmarks.

5.4 Conclusions and Future Work

In this study, we have shown that the computational resources demanded by neural network robustness verification can be greatly reduced by identifying and terminating runs on verification instances that will not be solved within their remaining time budget.

This answers the third research question (RQ3) of whether and to which extent we can predict the running time of a given verification algorithm for a specific problem instance: although it seems impossible to precisely extrapolate the running time to previously unseen instances, we show that a timeout can be reliably predicted for a given instance and time budget. Concretely, we showed that our method accelerates the verification procedure by 64% on average compared to the current state-of-the-art approach across a diverse set of benchmarks from the verification literature, while certifying a comparable number of instances. To predict whether an instance will be solved, we leveraged running time prediction techniques that employ novel static and dynamic features capturing both characteristics of the verification instance as well as features related to the internal operations of the given verifier.

The success of the proposed method was enabled by several design decisions. First, we leveraged the evolving nature of our dynamic features by regularly predicting timeouts throughout the verification procedure. Moreover, we included a confidence parameter θ that controls the threshold the prediction value of the timeout class must exceed before an instance is terminated. Using this parameter, a user can adjust the method to either prioritise savings in compute resources or a higher number of solved instances. We show that for a high value of θ our method substantially accelerates the verification procedure while solving comparably many instances as the standard verification.

In future work, we seek to extend our approach to further BaB-based verification approaches (*e.g.*, MN-BaB). Furthermore, we plan to investigate if our proposed features could be applied in other contexts, such as algorithm selection or satisfiability prediction. Lastly, we are interested in further studying the running time prediction capabilities of our features, possibly enabling empirical scaling models of BaB-based verification.