
NN4SYSBENCH: CHARACTERIZING NEURAL NETWORK VERIFICATION FOR COMPUTER SYSTEMS

Shuyi Lin¹ Haoyu He¹ Tianhao Wei² Kaidi Xu³ Huan Zhang⁴ Gagandeep Singh⁴ Changliu Liu²
Cheng Tan¹

ABSTRACT

We present NN4SysBench, a benchmark suite for neural network verification that is composed of applications from the domain of computer systems. We call these *neural networks for computer systems* or *NN4Sys*. NN4Sys is booming: there are many proposals for using neural networks in computer systems—for example, databases, OSes, and networked systems—many of which are safety critical. Neural network verification is a technique to formally verify whether neural networks satisfy safety properties. We however observe that NN4Sys has some unique characteristics that today’s verification tools overlook and have limited support. Therefore, this benchmark suite aims at bridging the gap between NN4Sys and the verification by using impactful NN4Sys applications as benchmarks to illustrate computer systems’ unique challenges. We also build a compatible version of NN4SysBench, so that today’s verifiers can also work on these benchmarks with approximately the same verification difficulties.

1 INTRODUCTION

AI safety is critical in the field of deep learning and neural networks. One pivotal technique to guarantee safety is *neural network verification* (Katz et al., 2017; Liu et al., 2019), or *NN-verification*. It offers systematic and rigorous guarantees to ensure the reliability and robustness of neural networks. In recent years, formally verifying neural networks has gained significant attention, and various benchmark suites (VNN, 2023) have emerged to assess the verification efficiency for different problems.

These benchmark suites provide a platform for researchers and practitioners to compare different verification methodologies, ultimately helping people use neural networks safely. For example, ACAS XU (Julian et al., 2016; Katz et al., 2017) is a benchmark to evaluate using neural networks as an air-to-air collision avoidance system, which provides turning suggestions to avoid an intruder aircraft. As another example, MNIST (LeCun et al., 1998) and CIFAR-10 (Krizhevsky et al., 2009; Gehr et al., 2018) are benchmarks to measure robustness of vision models trained on MNIST and CIFAR-10 datasets, making sure the networks are resilient to adversarial attacks.

However, there has been limited attention on verifying *neural networks for computer systems* (NN4Sys). NN4Sys are neural networks that replace traditional components in computer systems, like OSes, databases, and networked systems.

People have demonstrated that NN4Sys can significantly improve the performance of a system (Kraska et al., 2018; Mao et al., 2017). Meanwhile, since computer systems are critical infrastructures, NN4Sys has strong safety requirements. But, unlike traditional ML tasks, NN4Sys has unique characteristics which brings new challenges for today’s neural network verifiers. (We detail the motivation and the challenges in §2.) Thus, there is a strong need to connect NN4Sys and NN-verification.

To bridge the gap between NN4Sys and NN-verification, we introduce *NN4SysBench*, a benchmark suite for NN-verification to verify impactful NN4Sys applications. NN4SysBench encompasses a diverse set of applications from a wide spectrum of systems, including databases indexes (Kraska et al., 2018), distributed system schedulers (Mao et al., 2019b), and Internet congestion control (Jay et al., 2019). Each benchmark has a set of specifications that covers different verification difficulties. Over the past three years, NN4SysBench has been used as part of the benchmarks in an annual NN-verification competition, contributing to the evaluation of the state-of-the-art verifiers.

Goals and non-goals. As mentioned earlier, NN4Sys brings new challenges to verifiers, and until today, some NN4Sys properties still haven’t been supported. So, in order to use in practice, NN4SysBench has two versions—a vanilla version and a compatible version—targeting different goals:

- *Vanilla version*: this version contains the original neural networks proposed by their original authors, and the specifications that should have been verified assuming an oracle verifier.

¹Northeastern University ²CMU ³Drexel University ⁴UIUC.
Correspondence to: Cheng Tan <c.tan@northeastern.edu>.

- *Compatible version*: we customize the networks to simulate the original proposals, and remove the unsupported operators and network architectures, so that today’s verifier can verify its specifications.

The primary goal of NN4SysBench is to demonstrate the characteristics of NN4Sys, with the overarching aim of helping and accelerating advancements in future neural network verifiers. To achieve this, the vanilla version is built to exemplify verification challenges that NN4Sys poses, thereby serving as motivations for new verification techniques. Meanwhile, the compatible version is used for evaluating the performance of today’s verifiers. It provides an approximation of the verification difficulties inherent in various NN4Sys applications. Beyond its immediate usefulness, we envision that the compatible NN4SysBench will also serve as a stepping stone towards the eventual establishment of verifying vanilla NN4Sys.

However, NN4SysBench is not designed for the following points: first, NN4SysBench is not built for detecting unsafe behaviors in the original NN4Sys proposals. In our benchmarks, we replicate the original proposal by retraining the neural networks. A specification violation on our networks does not necessarily imply that the original proposal is unsafe. Second, NN4SysBench is not designed for proposing perfect specifications. We do not claim that the specifications used in our benchmarks are what NN4Sys must satisfy. The specifications are subjective to our understanding of the applications, and many specifications are “better-to-obey” rather than “have-to-obey”. In fact, it is an open question to design high-quality specifications for NN4Sys. Third, NN4SysBench does not contribute to NN-verification. It does not invent any new verification methods or algorithms.

Contributions. Our contributions are as follows.

- We design and implement NN4SysBench, a benchmark suite that bridge NN4Sys and NN-verification. We propose safety specifications for these NN4Sys applications.
- We customize the existing NN4Sys to form a compatible version of NN4SysBench, which enables today’s verifiers to experiment on NN4Sys applications.
- We highlight some NN4Sys characteristics that we observe, some of which significantly differ from classic machine learning tasks.
- We experiment NN4SysBench with two state-of-the-art verifiers, $\alpha\beta$ -CROWN (Zhang et al., 2018; Wang et al., 2021) and Marabou (Katz et al., 2019). They perform differently on different benchmarks, which indicates that NN4SysBench covers various verification difficulties.

We wish that NN4SysBench can illustrates how NN4Sys works, meanwhile also demonstrate the potentials of NN-verification for computer systems. NN4SysBench may further hint better ways for verifying NN4Sys applications.

2 MOTIVATION: VERIFYING NN4SYS

Background: NN4Sys. People have introduced a broad range of NN4Sys applications in computer systems. These applications either use neural networks to replace traditional data structures for performance (e.g., learned index (Marcus et al., 2020a)), or leverage neural networks to make better decisions (e.g., learned scheduler (Sabek et al., 2022)). Below we list a few NN4Sys applications in databases, OSES, and networked systems.

In databases, people proposed learned index (Kraska et al., 2018; Ding et al., 2020b;a; Tang et al., 2020; Marcus et al., 2020a), learned cardinality estimation (Kipf et al., 2018), and learned query optimizers (Marcus et al., 2020b; 2019). In operating systems, there are systems using neural networks for predicting I/O latency (Hao et al., 2020), page prefetching (Akgun et al., 2021), load balancing (Fingler et al., 2023), and job scheduling (Qiu et al., 2021). Likewise, neural networks are used for networked systems, including congestion control (Jay et al., 2019), datacenter network traffic optimization (Chen et al., 2018; Salman et al., 2018), resource allocation and scheduling (Mao et al., 2016; Xu et al., 2017; Mao et al., 2019b), optimizing video streaming (Mao et al., 2017), and packet classification (Liang et al., 2019). And, Haj-Ali et al. (2019) and Mao et al. (2019a) survey NN4Sys applications.

Why verifying NN4Sys? Despite having better (average) performance, NN4Sys is not widely deployed in practice. One major reason is that neural networks are black boxes and what has been learned by the neural networks is unclear to developers. Thus, neural networks may produce unexpected results for unseen inputs, and risk the stability and correctness of the systems. For example, a neural network based scheduler may attempt to schedule invalid jobs (Mao et al., 2016); a learned video streaming system may pick the worst bitrate even if the network condition is good (Eliyahu et al., 2021); a learned index may output a faraway data position for non-existing keys (Wei et al., 2023); and a learned cardinality may violate monotonicity and predict a smaller number for a larger query range (Wang et al., 2020).

Meanwhile, testing cannot address the abovementioned problem, as the input space is usually infinite (think of the database key space) and testing cannot cover all possible inputs. Despite the challenge of infinite input spaces, there is a hope: neural network verification can provably check networks for a continuous range of inputs. This enables developers to verify if a trained NN4Sys follows a safety properties for any inputs under some predefined conditions, the number of which can be infinite. In fact, people have already used NN-verification for examining desired properties of networked systems (Eliyahu et al., 2021; Dethise et al., 2021).

Vision: NN4Sys + NN-verification. We believe NN-verification works well with NN4Sys. First, unlike traditional ML tasks (e.g., image classification, speech recognition) which use large models (e.g., LLM), the size of neural networks in NN4Sys is manageable for verification. This is also observed by [Eliyahu et al. \(2021\)](#). Second, owing to the rigorous semantics of systems, the specifications of NN4Sys are unambiguous. Thus, we argue that NN4Sys should couple with NN-verification whenever it can, in order to build readily deployable networks in practice.

NN4Sys brings new verification challenges. Though promising, adopting verification in NN4Sys is non-trivial. This is because NN4Sys has its own characteristics, some of which turn out to be challenging for today’s neural network verifiers. Below, we list the unique characteristics of NN4Sys to highlight some challenges to apply NN-verification on NN4Sys applications.

- *Large number of specification entries.* We observe that NN4Sys usually has many specification entries. This is because safety properties need to cover the entire input space to be comprehensive. For example, the learned index in our benchmark has 150K entries.
- *Monotonicity specification.* Beyond normal specifications of specifying input-output constraints, NN4Sys also requires monotonicity properties. As an example, learned cardinality estimation requires results to be monotonically increasing while query ranges increase.
- *Probabilistic specification.* NN4Sys sometimes needs specifications to be probabilistic—it can be either the guarantees hold probabilistically (e.g., in probabilistic data structures) or developers want to give some leeway to networks (e.g., allowing false predictions).
- *Temporal specification.* In many cases, NN4Sys require properties related to time. For example, congestion control protocols should eventually increase the packet sending rate when facing good network conditions, but increasing doesn’t have to happen immediately.
- *Hierarchical models.* NN4Sys sometimes uses multiple neural networks in a hierarchical structure which together serve one task, for example, RMI ([Kraska et al., 2018](#)). Ideally, verification tools can check them end-to-end in one pass.

To clarify the challenges, here are several nuances to mention. First, we are not claiming that every challenge mentioned is impossible to verify; some are merely inefficient due to the absence of targeted support. Similarly, even if some challenges are not directly supported, there are alternative methods to verify them by making modifications to networks or specifications—as demonstrated in how we build the compatible version of NN4SysBench. It’s also worth noting that the verification of certain challenges is exclusive to specific verifiers and lacks a universal verifica-

tion interface. Lastly, there are indeed some challenges that are currently unverifiable (e.g., variable tensor indexing), presenting challenges to today’s verification techniques.

Bridging the gap with NN4SysBench. To address the divide between NN4Sys and NN-verification, we propose NN4SysBench, as a benchmark suite for NN4Sys. NN4SysBench is designed to include impactful NN4Sys that already exists, plus specifications that we create. Also, each benchmark is available in two versions: vanilla and compatible. So far, NN4SysBench includes six benchmarks that span a wide array of applications. These have been implemented in diverse computer systems, each providing unique functionalities and specifications. Next, we introduce NN4SysBench.

3 NN4SYSBENCH

NN4SysBench contains six NN4Sys applications, detailed in Figure 1. In the following, we will first introduce these applications (§3.1), then their specifications (§3.2), and final the overall organization of NN4SysBench (§3.3).

3.1 Neural networks for computer systems

The six applications in NN4SysBench are: learned index, learned bloom filter ([Kraska et al., 2018](#)), learned cardinalities ([Kipf et al., 2018](#)), learned congestion control ([Jay et al., 2019](#)), learned adaptive bitrate ([Mao et al., 2017](#)), and learned scheduler ([Mao et al., 2019b](#)). Below, we introduce them by their training methods.

Supervised learning. There are three NN4Sys applications that use supervised learning to learn:

- Database learned index (abbreviated as *LearnedIndex*): database index is a data structure to accelerate data retrieval that maps database keys to the corresponding data positions on the underlying storage. In learned index, people use ML models to replace traditional index data structures like B-Tree; the inputs of the learned index are database keys; and the outputs are data positions.
- Learned bloom filter (abbreviated as *BloomFilter*): a bloom filter is a probabilistic data structure that has been widely used in many computer systems. Bloom filters test whether an element (for example, a string) is in a pre-defined set. Bloom filters allow false positives: it may return true for an element that is not in the set. The inputs are being-tested elements, and the outputs are booleans, whether the elements are in the set.
- Learned cardinalities (abbreviated as *CardEsti*): database cardinality estimation predicts the number of rows returned by a database query (i.e., a SQL statement), which will then influence the query optimization plans. Traditional cardinality estimation relies on heuristics and do-

benchmark	bench size	training	network			operators		specifications			
			size	depth	in_dim	compatible	vanilla	reach	mono	prob	temp
LearnedIndex	small	SL	33K	8	1	Gemm,ReLU	–	✓			
	large		66K	12	1						
CardEsti	small	SL	103K	19	154	Gemm,ReLU, MatMul,Split,	–	✓			
	large		25M	18	154						
(dual-model)	small	SL	103K	21	308	ReduceSum,Slice, Concat,Sigmoid			✓		
	large		25M	20	308						
BloomFilter	–	SL	133K	8	1	Gemm,ReLU	–				✓
CongestCtrl	small	RL	0.6K	6	30	Gemm,Tanh	–	✓	✓		
	medium		1.5K								
	large		4.1K								
(dual-model)	small	RL	0.6K	8	60	Gemm,Tanh,Split	–		✓	✓	✓
	medium		1.5K								
	large		4.1K								
(chain-model)	small	RL	0.6K	40	151	Gemm,Tanh Split,Concat	–			✓	✓
	medium		1.5K								
	large		4.1K								
LearnedSched	–	RL	3K	217	4.3K	Gemm,ReLU, Split,Reshape, MatMul(const)	+LeakyReLU, +Slice,+Gather, +MatMul(var)	✓			
AdaptBitrate	small	RL	103K	12	48	Gemm, ReLU,	+Slice, +MatMul, +Conv1d	✓	✓	✓	
	medium		264K	11							
	large		527K	11							
(dual-model)	small	RL	103K	18	96	Reshape, Concat	+Flatten, +ReduceSum, +Gather		✓	✓	✓
	medium		264K								
	large		527K								

Figure 1. NN4SysBench application overview. We get network parameters from onnx-tool (onn, 2023): “size” is the number of trained parameters contained in the network; “depth” is the longest path in the network’s computational graph; “in_dim” is the size of the flattened input tensor. In “training”, “SL” means supervised learning; and “RL” means reinforcement learning. In “operators”, “vanilla” column indicates the operators used by the original models but not by the compatible models. In “specifications”, “reach”, “mono”, “prob”, “temp” represent reachability, monotonicity, probabalistic, and temporal specifications, respectively.

main knowledge; whereas, the learned cardinalities learn from the trace data. The inputs of learned cardinalities are SQL queries, and the outputs are estimated number of returned rows.

Reinforcement learning. The other three use reinforcement learning to learn:

- Learned Internet congestion control (abbreviated as *CongestCtrl*): congestion control protocols are used to ensure efficient data transmission and prevent packet congestion in a network. Learned congestion control uses reinforcement learning to study the patterns of congestion and non-congestion conditions, so it can predict if there is a congestion and what to do next (e.g., if a congestion exists, machines should lower their packet sending rates.) The inputs are network conditions, and the outputs are packet sending rates in the near future.
- Learned adaptive bitrate (abbreviated as *AdaptBitrate*): in video streaming, adaptive bitrate algorithms are used on client-side video players to decide what video resolution (e.g., 720P, 1080P, 4K) to download for the next chunk of the video. Learned adaptive bitrate uses neural networks to select future video chunks based on observations collected by client video players. The inputs are the states of the current video player and client-side measurements of the network, and outputs are the resolution

for the next video chunk.

- Learned distributed system scheduler (abbreviated as *LearnedSched*): a distributed computing system like Spark manages a cluster of machines and runs multiple jobs; each job contains multiple tasks, some of which depend on other tasks. A learned scheduler uses neural networks to learn workload-specific scheduling algorithms to optimize some high-level objective, such as minimizing average job completion time. The inputs are the status of the current jobs, tasks, and the cluster; the outputs are the next tasks to run.

Benchmark selection. To ensure benchmark relevance and quality, we considered the following criteria in selecting our benchmarks. First, to provide a comprehensive view of NN4Sys, we aim to cover a broad spectrum of applications in computer systems, including databases (LearnedIndex and CardEsti), networked systems (CongestCtrl and AdaptBitrate), distributed systems (LearnedSched), and generic data structures used in systems (BloomFilter). Second, we focus on highly cited works to ensure the benchmarks are impactful in their fields. Third, we prioritize the open-sourced projects, so we have accurate guidance to replicate the networks in a modern DL framework like PyTorch. Finally, we tend to choose benchmarks that have been previously subjected to verification efforts by other people (Eliyahu et al., 2021; Wei et al., 2023), as these NN4Sys applications

benchmark	spec	description
LearnedIndex	reach	All predicted data locations are error-bounded.
CardEsti	reach	The predicted cardinalities are close to the ground-truth cardinalities from the database.
	mono	A larger-ranged query returns larger cardinality.
BloomFilter	prob	The false positive and false negative rates are bounded.
CongestCtrl	reach	When observing good (bad) networking conditions, the sender does not decrease (increase) packet sending rates.
	mono	When observing better networking conditions, the sender increases packet sending rates by either the same or a larger amount.
	temp	When the networking condition changes from bad to good, the sender eventually increases packet sending rates.
LearnedSched	reach	(1) if job A's input depends on job B's output, B is not finished, then A should not be scheduled. (2) A user cannot get their jobs scheduled earlier by requiring more resources for them.
AdaptBitrate	reach	When facing good (bad) downloading conditions, the video streaming system should not pick the worst (best) video resolution.
	mono	Better downloading conditions implies better resolutions.

Figure 2. A high-level description of the specifications for the applications in NN4SysBench.

are what people want to verify and we can borrow intuitions from their specification design.

3.2 NN4Sys specifications

Specifications are safety properties that developers expect the NN4Sys to satisfy. For example, the specification for the learned index aims to guarantee that the outputs (i.e., the predicted data positions of a key) closely align with the actual data positions, within a predefined error bound. This ensures that the learned index consistently locates existing data within the database, an essential requirement for any index structure. The correctness and safety properties of various NN4Sys applications vary significantly and are specific on the requirements of applications and the problems they solve. NN4SysBench has a diverse collection of NN4Sys, hence having a diverse set of specifications. We brief the specifications for each NN4Sys in Figure 2.

In NN4SysBench, we design the specifications for each NN4Sys based on our understanding of the original systems. We also borrow specifications from prior verification work (Eliyahu et al., 2021; Wu et al., 2022) to some of these NN4Sys. We classify specifications to four categories: reachability, monotonicity, probabilistic, and temporal specifications.

Reachability specification. In some NN4Sys, developers want to bound the outputs of a neural network regarding a range of inputs, for example in learned indexes. We call this input-output properties, *reachability specifications*, which we define as follows.

Formally, consider a neural network as a function f to which inputs denote as $x \in \mathcal{D}_x$ and outputs are $y \in \mathcal{D}_y$, where $\mathcal{D}_x$ and $\mathcal{D}_y$ are the domain and range of f . Users can define a reachability specification by providing a pair of domains

as $\langle x \in \mathcal{X}, y \in \mathcal{Y} \rangle$, where $\mathcal{X} \subseteq \mathcal{D}_x$ and $\mathcal{Y} \subseteq \mathcal{D}_y$. And a reachability specification is written as:

$$\forall x, x \in \mathcal{X} \implies y = f(x) \in \mathcal{Y} \quad (1)$$

Reachability specification is also called *neural network robustness* in verifying vision models (Müller et al., 2020; Zhang et al., 2018; Cohen et al., 2019) and others.

Monotonicity specification. Monotonicity is a widely used correctness property in systems. For example, when the network condition becomes better, a video streaming system should not decrease the video quality. As another example, a database cardinality estimation estimates the number of database rows returned by a SQL statement. Users want that if two SQL statements query the same database and one queries a subset of the other, then the NN4Sys output (that is, the number of returned rows) of the subset query should be smaller than the other's. We define monotonicity specification as follows.

For a network f , two inputs x_0 and x_1 in some input domain $\mathcal{X}$, a monotonically increasing specification reads as:

$$\forall x_0, x_1 \in \mathcal{X}, \forall i, x_0[i] \geq x_1[i] \implies f(x_0)[j] \geq f(x_1)[j] \quad (2)$$

where j is an output dimension provided by users.

Probabilistic specification. This specification describes a NN4Sys obeying rules with some given probability. It is useful when either the guarantees hold probabilistically (e.g., in probabilistic data structures) or users want to give some leeway to NN4Sys (e.g., allowing few false predictions). For example, learned bloom filters require probabilistic specification because they allow a small number of false positives. We define probabilistic specifications below.

Given a network f and its input and output x and y , users can define a probabilistic specification by specifying a reachability specification $\langle \mathcal{X}, \mathcal{Y} \rangle$ and a probability $\mathcal{P}$ indicating how likely the probabilistic specification holds. The probabilistic specification is written as:

$$\forall x, x \in \mathcal{X} \implies \Pr(y \in \mathcal{Y} \mid y = f(x)) > \mathcal{P} \quad (3)$$

Temporal specification. Many NN4Sys require properties related to time, especially those applications interacting with environments. For example, a congestion control protocol should eventually increase the packet sending rate, if the network remains in good conditions. Note that the NN4Sys's decisions (here, the sending rates) may not change immediately, but the expected behavior should happen in a finite number steps.

3.3 Benchmark organization

In NN4SysBench, each application (e.g., learned index) forms a *benchmark set*. Each set contains multiple *benchmark cases*. A case represents a verification problem—given

a trained neural network and a set of specifications, does the network satisfy the specifications.

NN4SysBench comprises six benchmark sets, each corresponding to a NN4Sys application in Figure 1. Within each set, we’ve provided multiple pre-trained networks of varying sizes, reflecting a spectrum of verification challenges. Specifications are derived from our predefined templates, and users of NN4SysBench have the flexibility to configure both a random seed and the number of specifications to generate—these are two parameters in benchmark configurations. Besides network sizes, different specification templates also present varying levels of complexity.

Tuning benchmark difficulty. NN4SysBench serves as performance tests for neural network verifiers, requiring a diverse range of difficulties to effectively tell the capabilities of the verifiers. To provide varied levels of difficulties, we alter the networks and adjust the specifications by tuning several influencing factors, including network depth, network size, the number of perturbed dimensions in specifications, and the range of specification perturbations. The interplay between these factors and the verification difficulty is studied in section 5. This investigation helps us fine-tune the benchmark difficulty to examine the verifiers’ performance under varying conditions.

Building benchmarks from scratch. Beyond “ready-to-verify” benchmark cases, we also provide infrastructures for users who want to update the network architectures or modify specifications. Users can build their own benchmark cases with the desired neural networks and specifications. In particular, we provide the training code for each application based on PyTorch and specification templates written in VNN-LIB format (VNN, 2019).

4 DESIGNING COMPATIBLE BENCHMARKS

As mentioned earlier (§2), some of the networks and specifications are not supported by current neural network verifiers. To make NN4SysBench accessible to and compatible with today’s NN-verification, we build the *compatible NN4SysBench*. In particular, we customize the networks and specifications to satisfy the assumptions that verifiers make—the rules used by the International Verification of Neural Networks Competition (VNN, 2023). Below we detail how we approximate the unsupported characteristics.

Dual-model to simulate monotonicity. In order to support monotonicity specification (§3.2), we implement a *dual-model* architecture that simulates two network inferences at the same time. For example, in learned cardinalities (Kipf et al., 2018), we duplicate the pre-trained model and connect the two models side-by-side as a single new model (see Figure 3). The dual-model’s inputs and outputs are doubled compared to the original model. By using a split operator,

we divide the inputs in half: the first half is channeled to the first model, while the latter half is directed to the second model. Dual-model’s output is the difference between the estimated cardinality for the first-half and the second-half inputs.

With the dual-model, a monotonicity specification can be expressed as, for example: (except for *year*, other properties of q_1 and q_2 are the same.)

$$q_1.\text{year} < 2015 < q_2.\text{year} \implies y_1 - y_2 \leq 0$$

where q_1 and q_2 serve as inputs to the first and the second models respectively, while y_1, y_2 are the outputs from the the first and the second models, respectively.

Chain-model to approximate k -step temporal specifications. Multiple NN4Sys applications require to incorporate temporal specifications into their operations. Take, for instance, the case of Aurora (Jay et al., 2019), a system designed to tackle Internet congestion control with neural networks. In this context, users anticipate observing a gradual increase in the packet sending rate as network congestion subsides, a change that might not manifest instantaneously but should eventually happen, as long as the network conditions remain stable. These temporal specifications, which we formally defined in §3.2, play a crucial role in achieving desired system performance. However, today’s verifiers haven’t yet supported temporal specifications, highlighting a significant gap in verification capabilities to ensure the safety of NN4Sys applications operating within dynamic environments.

To simulate the temporal specifications, we construct *chain-models* by replicating the original network k times and concatenating them. This chain-model serves as a representation of the original model’s behavior over a span of k consecutive steps. Notably, the output from one model step serves as the input for the subsequent step, as illustrated in Figure 3. Consequently, for a chain-model, the cumulative inputs amount to k times the inputs of the original model, while the final output corresponds to the output of the ultimate model in the sequence. This approach allows us to model the temporal specifications required for congestion control, which specify that after k successive steps of favorable network conditions, the model should opt to increase the packet sending rate. Chain-model is not new; it has been used before in NN-verification by Eliyahu et al. (2021) and Wu et al. (2022).

Counting units for probabilistic specifications. Accurately verifying probabilistic specifications like learned bloom filters (Mitzenmacher, 2018; Vaidya et al., 2020) is challenging. Instead, we simulate this verification by (i) splitting the input space into tiny equal-sized units, (ii) verifying each unit separately, and (iii) counting the verified safe units over all units as the probability of verified safe space.

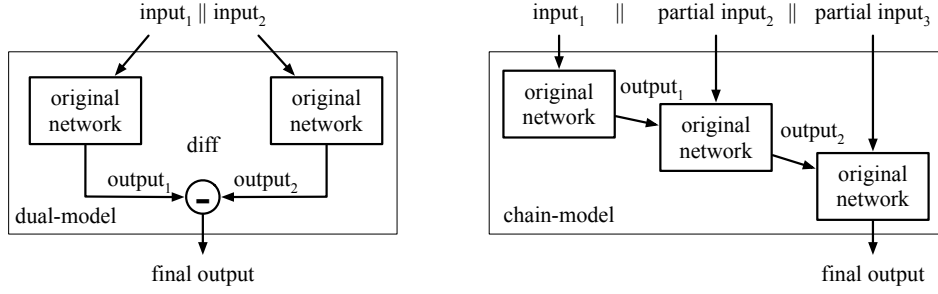


Figure 3. Dual-model and chain-model overview. In input tensors, “||” represents tensor concatenation. In the dual-model, the “diff” operator is based on the output of the original neural network. (Take CardEsti as an example: the output is the number of returned rows, so the “diff” is the subtraction operator.) In the chain-model, the “output” from the prior network and the “partial input” together form the complete input tensor for the next network in the chain.

Of course, this is a conservative approximation because a verified unsafe unit might have most of the space that is safe. The size of the unit will be a parameter to trade off verification performance and the accuracy of probabilistic specifications. For our learned bloom filter benchmark, we choose a dataset that has a two dimensional input (a geolocation with a latitude and a longitude). We split the input space evenly to 10K (i.e., 100×100) units.

Merging hierarchical models for learned index. It is hard to verify vanilla RMIs (Kraska et al., 2018) as a whole because an RMI has a hierarchical structure and is comprised of many models. One attempt is to merge models from all stages into a gigantic model by adding gluing neurons and manipulating their weights to simulate RMIs. This is an idea used in a VNN-COMP’21 benchmark (Wei et al., 2021). However, it fails as the glued model suffers from numerical instability problems due to the manipulated neurons.

In NN4SysBench, we use a straightforward approach to approximate learned index: we use a single neural network to learn what an RMI learns. Note that compared with RMIs, neural networks are much more expensive and difficult to train, as discussed by Kraska et al. (2018, §2.3). To address this, we borrow training approaches from Ouroboros (Wei et al., 2023) to train a single neural network that learns very well in a 150K-key lognormal dataset. NN4SysBench provides learned index in two sizes. One is a four-layer fully connected network with 128 neurons each layer; the other is a six-layer network with a width of 128.

5 EXPERIMENTAL EVALUATION

There are two questions we want to answer:

- How do state-of-the-art verifiers perform on NN4SysBench?
- How does verification difficulty change according to different benchmarks?

Verifiers. We experiments with two verifiers:

- $\alpha\beta$ -CROWN: $\alpha\beta$ -CROWN (abc, 2023) is a verifier based on an efficient linear bound propagation framework and branch and bound. It can be accelerated efficiently on GPUs and can scale to large networks.
- Marabou: Marabou (mar, 2023) is an SMT-based verifier that transforms neural network verification to constraint satisfaction problems, and solves the problem using a specialized SMT solver (github commit a2077b46).

Note that the verifier performance comparison is not entirely apple-to-apple, as $\alpha\beta$ -CROWN uses GPUs and Marabou does not.

Benchmarks. Each benchmark comprises (1) a trained neural network and (2) a set of specifications. We name a benchmark with the combination of these two. For example, CongestCtrl_small_chain_spec4 represents a benchmark from application CongestCtrl: the model size is small; it is a chain-model (§4); the corresponding specification is spec4. We briefly describe specifications below:

- LearnedIndex: all specifications are reachability specifications. The differences are the number of spec entries.
- CardEsti: the specifications for dual-model are monotonicity specifications, for normal models are reachability specifications.
- BloomFilter: the specifications are approximated probabilistic specifications (§4)
- CongestCtrl: spec101/102/2 are reachability specifications; spec3 is monotonicity specifications for dual-models; spec4 is temporal specifications for chain-models.
- LearnedSched: the specifications (spec1/2) are reachability specifications.
- AdaptBitrate: the specifications (spec1/2) are reachability specifications; spec3 is monotonicity specifications for dual-models.

Please see Figure 2 for the meaning of each type of specifications regarding different NN4Sys.

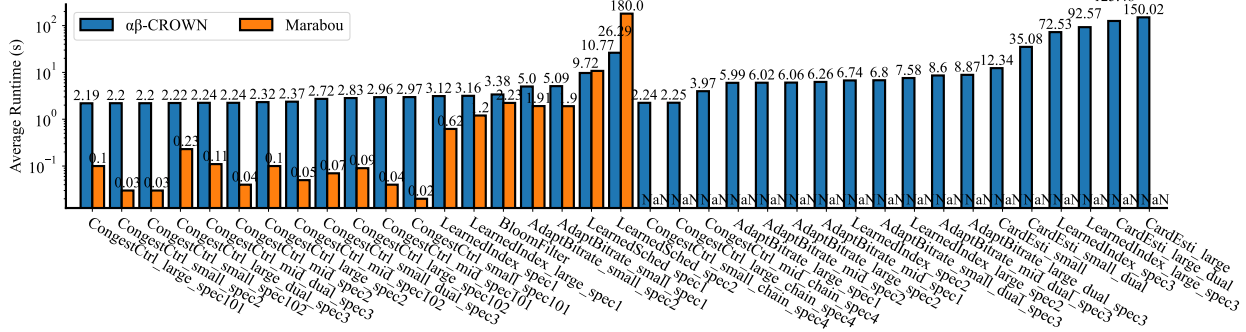


Figure 4. Verification runtime for different benchmarks. The x-axis are the benchmarks; the y-axis is the average verification runtime of ten cases (in log-scale). Bars on the left are the benchmarks solved by both verifiers; on the right are the ones only solved by $\alpha\beta$ -CROWN. The bars are sorted by the verification time, and the exact runtime is marked on top of the bars.

NN4SysBench configs. NN4SysBench allows users to configure the number of specifications for each benchmark, and randomly generate specifications (§3.3). In the following experiments, we fix the random seed to generate the same set of benchmarks for two verifiers, and we configure NN4SysBench to generate 10 specifications from one specification template. That means for each pair of application and specification template, we generate 10 cases.

Experiment setup. We run experiments on a machine with 32-Core Intel Xeon Gold 6338 CPU, 256GB memory, and an NVIDIA A30(24GB) GPU. The OS is Ubuntu 22.04. Verifier Marabou runs with Python 3.10.9, PyTorch 1.12.1, yaml 0.2.5, numpy 1.23.5, and onnx 1.13.1; $\alpha\beta$ -CROWN runs with Python 3.9.17, PyTorch 1.11.0, yaml 0.2.5, numpy 1.21.0, onnx 1.13.1, and onnx2pytorch 0.4.1. We run the two verifiers with different software versions because the verifiers require so.

5.1 NN4Sys verification performance

We experiment all the above mentioned benchmarks on the two verifiers, $\alpha\beta$ -CROWN and Marabou. We measure how much time each verifier spends on the verification. In this experiment, we collect the end-to-end verification time for each benchmark. Note that we configure NN4SysBench to generate 10 cases for each benchmark, so the runtime for each benchmark is the average of ten runs. If there is a timeout (180 seconds), we add 180 seconds to the overall time. Figure 4 shows the results.

As Figure 4 shows, different benchmarks have significantly different verification runtime (notice that y-axis is in log-scale). This highlights that NN4SysBench can distinguish the verification capabilities. Also, all benchmarks can be verified by at least one of the verifier, indicating that our compatible version of benchmarks indeed works with today’s verifiers. Finally, we can see that different verifiers perform differently on different benchmarks. For example,

Marabou can solve some small networks very efficiently; meanwhile, $\alpha\beta$ -CROWN can solve some large networks that Marabou cannot address.

Even in compatible version of NN4SysBench, there are several benchmarks that neither verifier is able to verify, due to unsupported operators. In particular, the $\alpha\beta$ -CROWN does not support scatter operator, which are used in chain-models, such as `LearnedSched_chain` and `AdaptBitrate_chain` models. Meanwhile, Marabou does not support `Add(const)`, `Matmul(var)`, `slice` operator, and some other operators, making it unable to verify a number of benchmarks.

5.2 Benchmark difficulty

NN4SysBench is designed to explore different verification difficulties. On the one hand, this demonstrates the intrinsic complexity of each NN4Sys application. On the other hand, various difficulties allow NN4SysBench, as a benchmark suite, provide more information for different verifiers. Next, we study what’s the difficulties of different NN4Sys applications, and different specifications. We use the verification time of $\alpha\beta$ -CROWN as a metric to tell the benchmark difficulty.

We experiment with two NN4Sys applications (`CongestCtrl` and `AdaptBitrate`) with multiple specification templates. We study their difficulties regarding their verification time and model sizes. Figure 5 shows the results.

There are three trends in Figure 5. First, the overall trend is that the larger the model, the longer the verification time. Hence, the verification difficulty increases when the model size increases. This is not surprising, as the number of non-linear operations decides the potential branches during verification. Second, for these two applications, monotonicity specifications are harder to verify than reachability specifications. For example, verifying `AdaptBitrate_small_dual_spec3` takes

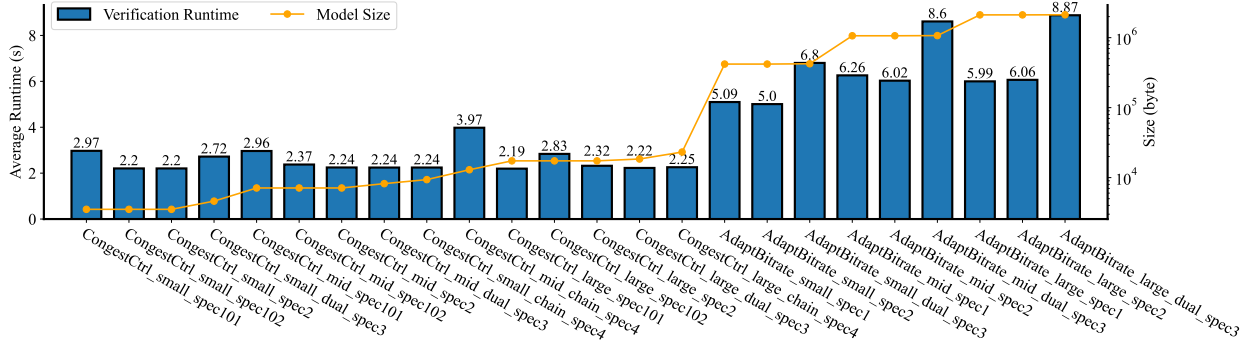


Figure 5. Verification time versus model size for CongestCtrl and AdaptBitrate. The bars represent the benchmark verification time from $\alpha\beta$ -CROWN (the left y-axis), and the line indicates the ONNX model size in bytes (the right y-axis). The benchmarks are sorted by their model size.

longer than verifying `AdaptBitrate_small_spec2`, even though they have similar model sizes. Third, safe specifications are harder to verify than unsafe specifications. For example, given the same model, verifying `CongestCtrl_small_spec101`, which contains 10 safe cases and 0 unsafe cases, is longer than verifying `CongestCtrl_small_spec102`, which contains 0 safe cases and 10 unsafe cases.

6 RELATED WORK

Benchmarks in verification. Benchmarking in verification has developed in response to the empirical research within different fields (Gupta, 1992; D’Silva et al., 2008). Compared to these developed fields, neural network verification is a new field under development, and verification competitions have been positive driven force to develop high-quality benchmarks. In particular, International Verification of Neural Networks Competition (Brix et al., 2023), VNN-COMP, has been a representative competition to introduce and compare state-of-the-art methods in NN-verification. It also has benchmarks spanning different applications and scenarios.

Carvana UNet (vnn, 2022) is a benchmark proposed in VNN-COMP that introduces networks and specifications for semantic segmentation in autonomous driving. To benchmark the ability that current verifiers handle practical neural network architectures, Carvana UNet covers comparably complex networks including Conv2d layers, AveragePool layers and TransposedConv Upsampling (Tran et al., 2021) layers followed by batch normalization. ERAN benchmark is proposed to understand how the choice of activation function affects certifiability. Although most of NN Verification methods focus their analysis on ReLU based networks, modern network architectures, such as EfficientNet (Tan & Le, 2019), are based on non-pieewise-linear activation functions. The ERAN benchmark aims at comparing the certifiability of networks based on pieewise-linear and non-

pieewise-linear activation functions. Xu et al. (2020) summarizes a benchmark suite on various tasks with networks, most of which are composed of fully-connected layers.

NN4SysBench is different from these literatures as it focuses on verification for neural networks within system applications, advocating to bridge the gap between current NN-verification methods and growing machine learning for system applications. The proposed benchmark provides supplementary tasks, architectures, and specifications to the verification community, and shows the potential to advance research in NN-verification for systems.

7 FUTURE WORK AND CONCLUSION

Future work. We’re working on adding more NN4Sys instances to NN4SysBench, for example, I/O latency predictors (Hao et al., 2020), OS kernel load balancing (Fingler et al., 2023), and learned memory allocator (Maas et al., 2020). One interesting topic is to study and provide a set of high-level specification interfaces for NN4Sys that covers most safety properties required by computer systems.

We also plan to include other categories of specifications, and develop sophisticated specifications that have more structures than a simple parallel OR in VNN-LIB. In particular, we plan to build compound specifications which are a combination of multiple different categories of specifications to serve the same safety property. For example, a compound of monotonicity and error-bounded existing keys can substitute the current specifications for learned index, with much fewer specification entries.

Conclusion. We present a benchmark suite, NN4SysBench, with a hope to bridge NN-verification and NN4Sys. NN4SysBench is designed for today’s verification tools while hinting at the future verification of NN4Sys.

REFERENCES

- The Verification of Neural Networks Library (VNN-LIB). www.vnnlib.org, 2019.
- Benchmark of VNN-COMP 2022. https://github.com/ChristopherBrix/vnncomp2022_benchmarks/tree/main/benchmarks, 2022.
- 4th International Verification of Neural Networks Competition (VNN-COMP’23). <https://sites.google.com/view/vnn2023>, 2023.
- alpha-beta-CROWN. <https://github.com/Verified-Intelligence/alpha-beta-CROWN>, 2023.
- Marabou. <https://github.com/NeuralNetworkVerification/Marabou>, 2023.
- onnx-tool. <https://github.com/ThanatosShinji/onnx-tool>, 2023.
- Akgun, I. U., Aydin, A. S., Shaikh, A., Velikov, L., and Zadok, E. A machine learning framework to improve storage system performance. In *Proceedings of the 13th ACM Workshop on Hot Topics in Storage and File Systems*, pp. 94–102, 2021.
- Brix, C., Müller, M. N., Bak, S., Johnson, T. T., and Liu, C. First three years of the international verification of neural networks competition (vnn-comp), 2023.
- Chen, L., Lingys, J., Chen, K., and Liu, F. Auto: Scaling deep reinforcement learning for datacenter-scale automatic traffic optimization. In *Proc. SIGCOMM*, 2018.
- Cohen, J., Rosenfeld, E., and Kolter, Z. Certified adversarial robustness via randomized smoothing. In *International Conference on Machine Learning*, pp. 1310–1320. PMLR, 2019.
- Dethise, A., Canini, M., and Narodytska, N. Analyzing Learning-Based Networked Systems with Formal Verification. In *Proceedings of INFOCOM’21*, 2021.
- Ding, J., Minhas, U. F., Yu, J., Wang, C., Do, J., Li, Y., Zhang, H., Chandramouli, B., Gehrke, J., Kossmann, D., et al. Alex: an updatable adaptive learned index. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020a.
- Ding, J., Nathan, V., Alizadeh, M., and Kraska, T. Tsunami: A learned multi-dimensional index for correlated data and skewed workloads. *arXiv preprint arXiv:2006.13282*, 2020b.
- D’Silva, V. V., Kroening, D., and Weissenbacher, G. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27:1165–1178, 2008.
- Eliyahu, T., Kazak, Y., Katz, G., and Schapira, M. Verifying learning-augmented systems. In *Proc. SIGCOMM*, 2021.
- Fingler, H., Tarte, I., Yu, H., Szekely, A., Hu, B., Akella, A., and Rossbach, C. J. Towards a machine learning-assisted kernel with lake. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pp. 846–861, 2023.
- Gehr, T., Mirman, M., Drachsler-Cohen, D., Tsankov, P., Chaudhuri, S., and Vechev, M. Ai2: Safety and robustness certification of neural networks with abstract interpretation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pp. 3–18. IEEE, 2018.
- Gupta, A. Formal hardware verification methods: A survey. In *Computer-Aided Verification*, pp. 5–92. Springer, 1992.
- Haj-Ali, A., Ahmed, N. K., Willke, T., Gonzalez, J., Asanovic, K., and Stoica, I. Deep reinforcement learning in system optimization. *arXiv preprint arXiv:1908.01275*, 2019.
- Hao, M., Toksoz, L., Li, N., Halim, E. E., Hoffmann, H., and Gunawi, H. S. Linnos: Predictability on unpredictable flash storage with a light neural network. In *Proc. OSDI*, 2020.
- Jay, N., Rotman, N., Godfrey, B., Schapira, M., and Tamar, A. A deep reinforcement learning perspective on internet congestion control. In *International Conference on Machine Learning*. PMLR, 2019.
- Julian, K. D., Lopez, J., Brush, J. S., Owen, M. P., and Kochenderfer, M. J. Policy compression for aircraft collision avoidance systems. In *2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)*, pp. 1–10. IEEE, 2016.
- Katz, G., Barrett, C., Dill, D. L., Julian, K., and Kochenderfer, M. J. Reluplex: An efficient smt solver for verifying deep neural networks. In *International Conference on Computer Aided Verification*, pp. 97–117. Springer, 2017.
- Katz, G., Huang, D. A., Ibeling, D., Julian, K., Lazarus, C., Lim, R., Shah, P., Thakoor, S., Wu, H., Zeljić, A., et al. The marabou framework for verification and analysis of deep neural networks. In *Computer Aided Verification: 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I 31*, pp. 443–452. Springer, 2019.

- Kipf, A., Kipf, T., Radke, B., Leis, V., Boncz, P., and Kemper, A. Learned cardinalities: Estimating correlated joins with deep learning. *arXiv preprint arXiv:1809.00677*, 2018.
- Kraska, T., Beutel, A., Chi, E. H., Dean, J., and Polyzotis, N. The case for learned index structures. In *Proc. SIGMOD*, 2018.
- Krizhevsky, A., Hinton, G., et al. Learning multiple layers of features from tiny images. 2009.
- LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.
- Liang, E., Zhu, H., Jin, X., and Stoica, I. Neural packet classification. In *Proc. SIGCOMM*. 2019.
- Liu, C., Arnon, T., Lazarus, C., Strong, C., Barrett, C., and Kochenderfer, M. J. Algorithms for verifying deep neural networks. *arXiv preprint arXiv:1903.06758*, 2019.
- Maas, M., Andersen, D. G., Isard, M., Javanmard, M. M., McKinley, K. S., and Raffel, C. Learning-based memory allocation for c++ server workloads. In *Proc. ASPLOS*, 2020.
- Mao, H., Alizadeh, M., Menache, I., and Kandula, S. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM workshop on hot topics in networks*, pp. 50–56, 2016.
- Mao, H., Netravali, R., and Alizadeh, M. Neural adaptive video streaming with pensieve. In *Proc. SIGCOMM*, pp. 197–210, 2017.
- Mao, H., Negi, P., Narayan, A., Wang, H., Yang, J., Wang, H., Marcus, R., Addanki, R., Khani, M., He, S., et al. Park: An open platform for learning augmented computer systems. 2019a.
- Mao, H., Schwarzkopf, M., Venkatakrishnan, S. B., Meng, Z., and Alizadeh, M. Learning scheduling algorithms for data processing clusters. In *Proc. SIGCOMM*. 2019b.
- Marcus, R., Negi, P., Mao, H., Zhang, C., Alizadeh, M., Kraska, T., Papaemmanouil, O., and Tatbul, N. Neo: A learned query optimizer. *arXiv preprint arXiv:1904.03711*, 2019.
- Marcus, R., Kipf, A., van Renen, A., Stoian, M., Misra, S., Kemper, A., Neumann, T., and Kraska, T. Benchmarking learned indexes. *arXiv preprint arXiv:2006.12804*, 2020a.
- Marcus, R., Negi, P., Mao, H., Tatbul, N., Alizadeh, M., and Kraska, T. Bao: Learning to steer query optimizers. *arXiv preprint arXiv:2004.03814*, 2020b.
- Mitzenmacher, M. A model for learned bloom filters and optimizing by sandwiching. *Advances in Neural Information Processing Systems*, 31, 2018.
- Müller, C., Singh, G., Püschel, M., and Vechev, M. Neural network robustness verification on gpus. *arXiv preprint arXiv:2007.10868*, 2020.
- Qiu, Y., Liu, H., Anderson, T., Lin, Y., and Chen, A. Toward reconfigurable kernel datapaths with learned optimizations. In *Proc. HotOS*, 2021.
- Sabek, I., Ukyab, T. S., and Kraska, T. Lsched: A workload-aware learned query scheduler for analytical database systems. In *Proceedings of the 2022 International Conference on Management of Data*, pp. 1228–1242, 2022.
- Salman, S., Streiffer, C., Chen, H., Benson, T., and Kadav, A. Deepconf: Automating data center network topologies management with machine learning. In *Proceedings of the 2018 Workshop on Network Meets AI & ML*, 2018.
- Tan, M. and Le, Q. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International Conference on Machine Learning*, pp. 6105–6114. PMLR, 2019.
- Tang, C., Wang, Y., Dong, Z., Hu, G., Wang, Z., Wang, M., and Chen, H. Xindex: a scalable learned index for multicore data storage. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 308–320, 2020.
- Tran, H.-D., Pal, N., Musau, P., Lopez, D. M., Hamilton, N., Yang, X., Bak, S., and Johnson, T. T. Robustness verification of semantic segmentation neural networks using relaxed reachability. In *International Conference on Computer Aided Verification*, pp. 263–286. Springer, 2021.
- Vaidya, K., Knorr, E., Kraska, T., and Mitzenmacher, M. Partitioned learned bloom filter. *arXiv preprint arXiv:2006.03176*, 2020.
- Wang, S., Zhang, H., Xu, K., Lin, X., Jana, S., Hsieh, C.-J., and Kolter, J. Z. Beta-CROWN: Efficient bound propagation with per-neuron split constraints for complete and incomplete neural network verification. *Advances in Neural Information Processing Systems*, 34, 2021.
- Wang, X., Qu, C., Wu, W., Wang, J., and Zhou, Q. Are we ready for learned cardinality estimation? *arXiv preprint arXiv:2012.06743*, 2020.
- Wei, T., Tan, C., and Changliu, L. VNN-COMP 2021 benchmark. <https://github.com/stanleybak/vnncomp2021/blob/main/benchmarks/nn4sys/>, 2021.

- Wei, T., Jia, Z., Liu, C., and Tan, C. Building verified neural networks for computer systems with ouroboros. *Proceedings of Machine Learning and Systems*, 5, 2023.
- Wu, H., Barrett, C., Sharif, M., Narodytska, N., and Singh, G. Scalable verification of gnn-based job schedulers. *Proceedings of the ACM on Programming Languages*, 6 (OOPSLA2):1036–1065, 2022.
- Xu, D., Shriver, D., Dwyer, M. B., and Elbaum, S. G. Systematic generation of diverse benchmarks for dnn verification. *Computer Aided Verification*, 12224:97 – 121, 2020.
- Xu, Z., Wang, Y., Tang, J., Wang, J., and Gursoy, M. C. A deep reinforcement learning based framework for power-efficient resource allocation in cloud rans. In *2017 IEEE International Conference on Communications (ICC)*. IEEE, 2017.
- Zhang, H., Weng, T.-W., Chen, P.-Y., Hsieh, C.-J., and Daniel, L. Efficient neural network robustness certification with general activation functions. *Advances in Neural Information Processing Systems*, 31:4939–4948, 2018. URL <https://arxiv.org/pdf/1811.00866.pdf>.