Characterizing Neural Network Verification for Systems with NN4SysBench

Haoyu He [1] Tianhao Wei [2] Huan Zhang [2] Changliu Liu [2] Cheng Tan [1]

Abstract

We present NN4SysBench, a benchmark suite for neural network verification, comprised of benchmarks from *neural networks for systems* (or *NN4Sys*). NN4Sys is booming: there are hundreds of proposals of using neural networks in computer systems—databases, OSes, and networked systems—that are safety critical. We observe that NN4Sys has some unique characteristics that today's neural network verification tools overlooked. This benchmark aims at bridging the gap between NN4Sys and NN-verification by tailoring impactful NN4Sys instances to benchmarks that today's NN-verification tools can work on.

1. Introduction

Applying deep learning techniques on tasks in computer systems attracts much attentions recently. There are many proposals to replace system components with neural networks. We call these, *neural networks for systems* or *NN4Sys*. NN4Sys have been used for database indexes (Kraska et al., 2018), congestion control (Jay et al., 2019), database query optimization (Krishnan et al., 2018), memory prefetching (Hashemi et al., 2018), memory allocator (Maas et al., 2020), and I/O latency prediction in OS (Hao et al., 2020).

Deploying NN4Sys however faces a challenge in practice; that is, neural networks are black boxes and may produce unpredictable outcomes. For example, a neural network based scheduler may attempt to schedule invalid jobs (Mao et al., 2016). A learned index may output a faraway data position for non-existing keys. A learned cardinality estimation may predict a smaller number for a larger query range.

Meanwhile, debugging and fixing neural networks for certain behaviors are hardly conceivable thus far. Despite the challenge, there is a hope: *neural network verification* (NN-verification) can provably check if networks satisfy user-defined specifications. This enables developers to verify if a trained NN4Sys follows some basic safety properties. In fact, people have already used NN-verification for examining desired properties in networked systems (Eliyahu et al., 2021; Dethise et al., 2021).

Indeed, we believe NN-verification works well with NN4Sys, for two reasons. First, the network sizes of NN4Sys are usually small (see also Eliyahu et al. (2021, Table 1)), which are suitable for expensive verification. Second, the specifications of NN4Sys are unambiguous hence are straightforward to develop, owing to the rigorous semantics of systems. Thus, we argue that NN4Sys should couple with NN-verification whenever it can, in order to build readily deployable networks in practice.

Though promising, using verification in NN4Sys today is limited. To see why, we studied two NN4Sys applications—database learned indexes (Kraska et al., 2018) and cardinality estimation (Kipf et al., 2018)—and summarize the difficulties of applying today's NN-verification to NN4Sys. In particular, we want to illustrate the unique characteristics of NN4Sys and highlight some challenges for today's NN-verification. We brief the four characteristics that we observe. (This is by no means a comprehensive list.)

- *Small number of input dimensions.* Compared with vision models with high-dimensional inputs (in thousands), both NN4Sys cases we studied have fewer than tens of input dimensions that can be perturbed. This suggests that, considering a single specification entry (an input-output constraint), NN4Sys instances are easy to verify.
- *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 (§3.1).
- *Hierarchical models.* NN4Sys sometimes uses multiple neural networks in a hierarchical structure which together serve one task (an example is Figure 1). An end-to-end verification ideally can check them in one pass.
- *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 (§3.2).

[1]Khoury College of Computer Sciences, Northeastern University [2]Department of Computer Science, CMU. Correspondence to: Cheng Tan <c.tan@northeastern.edu>.

1st *Workshop on Formal Verification of Machine Learning*, Baltimore, Maryland, USA. Colocated with ICML 2022. Copyright 2022 by the author(s).

Some characteristics mentioned are not well supported by today's verification tools. To bridge NN4Sys and NN-verification, we propose a benchmark suite, *NN4SysBench*. NN4SysBench is designed to include impactful NN4Sys that already exists, plus specifications that we create. So far, we have two benchmarks, learned index (§3.1) and cardinality estimation (§3.2).

NN4SysBench is tailored for today's verification tools. Benchmark models and specifications are customized to follow verification conventions and assumptions. For example, in learned index, we use a single neural network to replace the original Recursive Model Index that combines multiple networks (§3.1); in learned cardinality estimation, we develop a dual-model to simulate monotonicity comparison between two inferences (§3.2). We wish that NN4SysBench can demonstrate the power of NN-verification, meanwhile also illustrates how NN4Sys works, which may hint better ways for verifying NN4Sys.

2. Background and related work

2.1. Neural networks for systems

People introduce a broad range of neural networks for computer systems. This paper studies two of them: database learned index (§2.2) and learned cardinality estimation (§2.3). Learned index was initially proposed by Kraska et al. (2018), which try to replace classic database indexes with neural networks. Cardinality estimation (Liu et al., 2015) is used by database query optimizer to predict sizes of query results, and Kipf et al. (2018) proposed to use neural networks for the estimation.

Beyond databases, many NN4Sys proposals exist in other system areas. For example, neural networks are used for congestion control (Jay et al., 2019) and datacenter network traffic optimization (Chen et al., 2018; Salman et al., 2018) in networked systems. In operating systems, there are systems using neural networks for predicting I/O latency (Hao et al., 2020), page prefetching, and job scheduling (Qiu et al., 2021). In particular, reinforcement learning is popular to train NN4Sys (Haj-Ali et al., 2019; Mao et al., 2019).

2.2. Learned index

Indexes are common data structures in systems, which enable fast data accesses. For example, a classic index data structure is B-Tree. It allows database users to quickly find data stored on the underlying storage. Learned index structure (Kraska et al., 2018) is an alternative index structure that uses neural networks for better lookup time and smaller memory footprints than B-Tree. In databases, index's inputs are database keys, and the outputs are data positions stored on disk. The data are sorted by keys. Learned indexes (namely neural networks) should learn the key distribution of the databases to give accurate predictions of data positions. Recursive Model Index (RMI) is the first learned index structure (Kraska et al., 2018) that we elaborate in section 3.1. Also, there is a line of work (Ding et al., 2020b;a; Tang et al., 2020; Marcus et al., 2020) to optimize the performance of learned indexes and extend learned indexes to different environments, for example, multi-core machines (Tang et al., 2020).

2.3. Learned cardinality estimation

Cardinality estimation (CE) plays a significant role in query optimization of database systems. It aims at estimating the size of sub-plans of each query and guiding the optimizer to select the optimal join operations. Performance of cardinality estimation has great impact on the quality of the generated query plans. With the proliferation of widespread applications of neural networks, recent works (Wang et al., 2020) try to apply machine learning methods to estimate the cardinality of queries. Kipf et al. (2018) proposed a multi-set MLP-based architecture (MSCN) to frame the cardinality estimation problem as a typical deep learning task. Despite the promising results, learned cardinality has a drawback that it neglects the internal semantic logic of each query as a result of encoding queries into numerical vectors. Our CE benchmark is built upon this. Multiple other proposals also use neural networks for cardinality estimation (Wang et al., 2020).

2.4. Benchmarks in verification

Benchmarking in verification has developed in response to the empirical research within different fields, which is in favor of benchmarks that are diverse in structure and difficulty, and are representatives of the use cases. In diverse problem settings, people made great efforts to build benchmarks for hardware (Gupta, 1992), software (D'Silva et al., 2008), and constraints (Gent & Walsh, 1999). These well curated benchmarks have been invaluable resources to advance a research community.

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 Bak et al. (2021), 9 benchmarks were proposed and most of them are designed to fit in the field of computer vision. Xu et al. (2020a) devised a benchmark generator on various tasks with networks, most of which are composed of fully-connected layers. Our work is different from these literatures in that we concentrate in the field of neural network verification for systems. Our benchmark provides supplementary tasks and architectures to the verification community, and shows the potential to advance research in neural network verification.

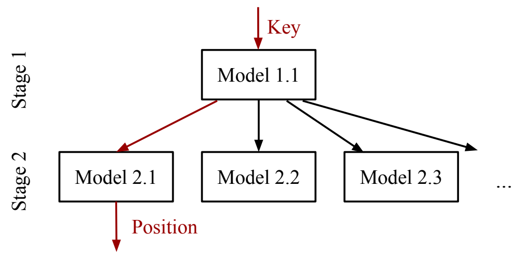

Figure 1. A two-stage Recursive Model Index (RMI). In this example, this RMI takes a “Key” as an input, chooses “Model 1.1” and “Model 2.1” for prediction, and finally produces the “Position” which is supposed to be the data location indexed by “Key”.

3. NN4SysBench

In this section, we introduce the two benchmarks in NN4SysBench: learned index (§3.1) and learned cardinality estimation (§3.2). We summarize their characteristics in section 3.3, and further elaborate how we choose benchmark parameters in section 3.4.

3.1. Learned index

Learned indexes have been extensively studied by using different ML approaches (Marcus et al., 2020). Below we introduce the first learned index using neural networks, named *Recursive Model Index* (RMI), which is depicted in Figure 1. RMI has multiple stages and each stage has one or multiple models (neural networks). During a lookup, RMI picks one model in each stage to run; models in upper stages (starting from stage 1) decide the model in the next stage; and a final stage model predicts the data position for the queried key. As the best practice (Kraska et al., 2018), people use two-stage RMIs.

3.1.1. Specification

The correctness property of a RMI is to ensure that models *always* produce data positions within a certain error-bound, so that RMIs can always find existing data in the database (a required property for any index structures). Original RMIs achieve this by evaluating all existing keys on the trained neural networks, and replace those inaccurate ones with traditional B-Trees. But, this approach does not provide guarantees for *non-existing* keys—the predicted data positions can be arbitrary. As discussed in the RMI paper (Kraska et al., 2018, §3.4), original RMIs can handle non-existing keys by forcing all neural network models to be monotonic or using exponential search techniques, which requires extra effort.

In NN4SysBench, the specifications for learned index require that the neural networks predict with bounded errors for *all* keys (including non-existing keys). This applies to range queries whose upper/lower bound of the range might be non-existing keys. In particular, one specification entry reads as follows,

$$\forall k \in [\mathcal{K}[i], \mathcal{K}[i{+}1]], F(k) \in [DB(\mathcal{K}[i]){-}\epsilon, DB(\mathcal{K}[i{+}1]){+}\epsilon]$$

where k is a key, $\mathcal{K}$ is the sorted list of existing keys, $F(\cdot)$ is the learned index, DB is the database key-position mapping (ground truth), and ϵ is the error bound. The number of specification entries equals the number of keys in the database.

3.1.2. Network

Vanilla RMIs cannot be verified as a whole by today’s verification tools because the tools assume that verification only applies to a single neural network. One attempt is to merge models from all stages into a gigantic model by adding gluing neurons and manipulating their weights to simulate RMIs (Wei et al., 2021). This is a benchmark in VNN-COMP’21 (Bak et al., 2021). However, it suffers from numerical instability problems during verification due to the manipulated neurons.

In NN4SysBench, we train a single neural network for learned index benchmark, which is much more expensive than RMIs as discussed by Kraska et al. (2018, §2.3). In particular, we borrow training approaches from Ouroboros (Tan et al., 2021) to train a single neural network that learns very well in a 150K-key lognormal dataset. NN4SysBench has two learned index 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.

3.2. Learned cardinality estimation

Cardinality Estimation (CE) predicts the number of return rows from a database (referred as the cardinality) given a SQL query or subquery. Consider a relation R with n attributes $\{A_1, A_2, \cdots, A_n\}$, and a query q over R with a conjunctive of d predicates. The cardinality of query q over a this relation R can be represented as:

$$\text{SELECT COUNT(*) FROM } q,$$

$$q := R \text{ WHERE } \theta_1 \text{ AND } \cdots \theta_d,$$

where $\theta_i (i \in [1, d])$ can be joins like $A_1 = A_3$ or predicates like $A_1 = a$, $A_2 \geq b$, $a, b \in \mathbb{R}$. The goal of a cardinality estimation function $F(\cdot)$ is to approximate the value of CE given a query. Usually, learned cardinality methods apply neural networks to parameterize F and the input of F is numerical vectors featurized from q.

3.2.1. Specification

To build specifications for learned cardinality estimation, we first introduce properties that correlate with the task semantics in the specification language. Specifications include:

1. $F(q) \leq |R|$.
2. $F(q) \geq 0$.
3. $F(q) \geq F(q \wedge \theta_i), i \in [1, d]$.
4. $F(q_1) \geq F(q_2)$, if q_1 and q_2 only differ in that they have unique attributes A_i limited in overlapped ranges such that $\theta_{q_1} := A_i \leq a, \theta_{q_2} := A_i \leq b$.
5. $n \leq F(q) \leq m$, where $q = \neg\theta_{q_1} \wedge q_2$, q_1 and q_2 only differ in that they have unique attributes A_i limited in overlapped ranges such that $\theta_{q_1} := A_i \leq a$, $\theta_{q_2} := A_i \leq b$. The cardinality of q_1 and q_2 is n, m, respectively.

The first two properties require the prediction to be no less than 0 and no greater than the number of total data pieces in a relation, which are very basic rules that a cardinality estimator should follow. Although these two properties seem trivial, they are necessary for verifying learned cardinality estimator because of the black-box characteristics of deep neural networks. Property 3 follows a natural intuition that if we add one more predicate to a query, then the cardinality of this query should be less than the original query. To better explain the proposed specifications, we illustrate properties 4 and 5 with two examples in the following. Assume

$$q_1 = \text{FROM table t WHERE t.year} < 2015, CE(q_1) = n,$$

$$q_2 = \text{FROM table t WHERE t.year} < 2020, CE(q_2) = m,$$

It is logically reasonable that n should be no more than m because there should be no less than 0 data piece between year 2015 and 2020. Therefore, we can build a specification that $F(q_1) \geq F(q_2)$, leading to property 4. Note that constructing specifications of property 4 does not need the true value of cardinality to a query because it verifies the monotonicity of model predictions. To the best of our knowledge, our benchmark is the first to concern the monotonicity of neural networks and we will describe how we implement this in Section 3.2.2. For property 5, we can construct another specification, where the pre-condition requires the value of t.year to be in the range of $[2015, 2020]$, and the post-condition limits the prediction to be in the range of $[n, m]$. These specifications are different from previous works in that they range the inputs of the networks following the task semantics and practical logical rules, while previous works performed random perturbation (e.g. adding noise to an image).

Specification generation. We leverage the mechanism in Kipf et al. (2018) to feature every query into a vector that includes binary value and numerical value. Represent a query q as a collection (T_q, J_q, P_q) of a set of tables $T_q \subset T$, a set of joins $J_q \subset J$, and a set of predicates $P_q \subset P$. For each table t and join j, we encode them as a unique one-hot vector. For predicates in the form of an attribute in a value range (*att*, *op*, *val*), *att* and *op* are featurized as one-hot vector and the *val* is normalized as a numerical value between 0 and 1. Given a set of SQL queries with the real cardinality that are used to train cardinality estimators, we can perform automatic generation following our proposed properties in Section 3.2.1 by tuning the featured vector.

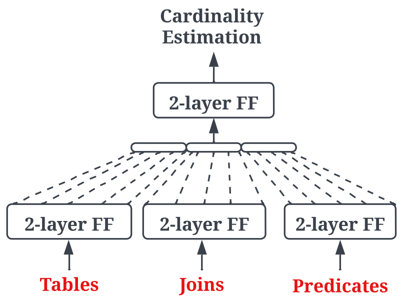

Figure 2. A multi-set networks with three separate modules for embedding tables, joins, and predicates as E_t, E_j, E_p and a final output module that take the concatenation of E_t, E_j, E_p as input.

In order to support monotonicity specification (the property 4), we implement a *dual-model* architecture that simulates two network inferences at the same time (details in §3.2.2). 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.)

$$q_1.year < 2015 < q_2.year \implies y_1 < y_2$$

where $q_1.year$ and $q_2.year$ are two inputs of the same inference to the dual-model; $y_1 = F(q_1)$ and $y_2 = F(q_2)$ are the outputs of the same dual-model inference.

3.2.2. Network

We use the representative *MSCN* (Kipf et al., 2018) architecture as the estimator of our learned cardinality task. *MSCN* is a multi-set feed-forward network (FFN) where tables, joins, and predicates are represented as separate modules, comprised of one two-layer neural network per set element with concatenation operations. This architecture (See Figure 2) itself is appropriate for advancing the development of today's verification tools because some of them do not support the concatenation operation.

Dual-model for monotonicity. To enable monotonicity verification for today's tools, we duplicate the trained MSCN model and connect the two models side-by-side as a single new model, called a dual-model (See Figure 3). The dual-model's inputs and outputs are doubled compared to the original MSCN. We use a split operator to split inputs into two and send them to the two identical "internal models": the first half of the inputs go to the first model and

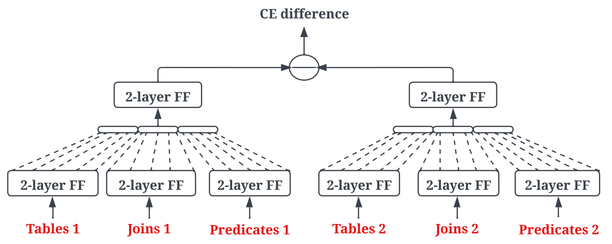

Figure 3. The dual model do parallel inferences on a couple of inputs and produces the difference of predictions.

the second half to the second model. Dual-model's output is the difference between the estimated cardinality for the first-half inputs, and the second-half inputs.

3.3. Benchmark characteristics

Different from existing verification benchmarks like MNIST, CIFAR10, and ACAS Xu (Katz et al., 2017), NN4SysBench has four unique characteristics. First, NN4Sys has a small number of input dimensions; this applies to both studied cases. Learned index has a single input dimension and learned cardinality estimation has less than ten inputs that could be perturbed (others are one-hot encoding). Compared with thousands of input dimensions in vision models, these numbers are tiny. This is because each input dimension in NN4Sys usually represents a factor in systems and there are not many meaningful factors in practice.

Second, NN4SysBench specifications have many entries, which appears in both studied cases. The number of specification entries in learned index is 150K because that is the number of keys in the database. Similar scenarios occur for learned cardinality estimation. We believe this is broadly applicable to NN4Sys, which is rooted in having a comprehensive specification that covers the entire input space (like the database key space).

Third, NN4Sys sometimes needs hierarchical structure that combines multiple models. This happens in the RMI of learned index (§3.1). In practice, people sometimes train multiple models and organize them in a certain structure to achieve better accuracy or shorter training time. The idea is that each model only learns a small portion of the whole problem (hence is easier to train), and people can cherry-pick models that behave well and discard the ones that do not. So far, NN-verification tools lack the ability to verify multiple models end-to-end.

Finally, monotonicity is a desired property that learned cardinality estimation needs, but hasn't been well supported. As mentioned (§3.2.1), the property 4 in cardinality estimation specifications is the monotonicity requirement. It says that a query with a larger range should return the same or more number of rows. Monotonicity is a common case in NN4Sys, for example, I/O latency prediction in which latencies should monotonically increase regarding queue lengths. However, verifying monotonicity is not supported by today's verification tools.

3.4. Benchmark parameters

To construct a difficulty-adaptable benchmark, we apply several parameters to tune the difficulty. General parameters for both tasks in NN4SysBench include: (1) the depth and width of neural networks, (2) the number of entries in specifications, (3) the ratio of safe and unsafe specifications. There is an extra parameter for learned cardinality estimation, (4) if verifying monotonicity specifications for dual-models. In NN4SysBench, we provide multiple versions of neural networks for different difficulties (tuning parameter (1) and (4)). We also generate a large set of specifications (in hundreds) with different numbers of entries (ranging from 1 to hundreds of thousands) to vary difficulties (tuning parameters (2), (3), and (4)).

4. Experiments

In this section, we experiment NN4SysBench with the state-of-the-art neural network verifier, and show its performance on different difficulties.

Setup. We run all our experiments on a `g5.4xlarge` EC2 machine, with 16 vCPUs, 64GB memory, and an NVIDIA A10G GPU. We use α,β-CROWN (Zhang et al., 2018; Xu et al., 2020b; Salman et al., 2019; Xu et al., 2021; Wang et al., 2021) as our verifier, which is the winner of VNNCOMP'21 (Bak et al., 2021).

4.1. Learned index

For learned index, we study how the depth of neural networks and the number of specification entries (namely, tuning parameter (1) and (2) in §3.4) may vary verification time. We run the verifier on two feed-forward networks of the same width (of 128) but with different depths: one has four layers; the other has six layers. The number of specification entries range from 1 to 800. All the specification entries are verified safe. Figure 4 shows the results. We learn that the verification time grows linearly regarding the growth of specification entries. This is expected because the current verifier checks one entry at a time. Also, the deeper the network, the slower the verification. But, the verification time does not growing proportionally regarding the depth of a neural network: verifying the deeper 6-layer network costs less than 1.5× of the shallower 4-layer network. Our hypothesis is that because learned indexes are easy to verify, the setup overheads (which are stable for all networks) are non-negligible, hence the runtime difference is not as large.

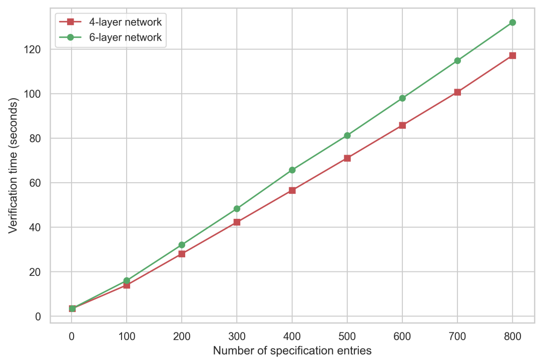

Figure 4. Verification time on specifications that have different number of entries. For the purpose of experiments, we limit the number of entries to be 800 instead of 150K in our benchmark.

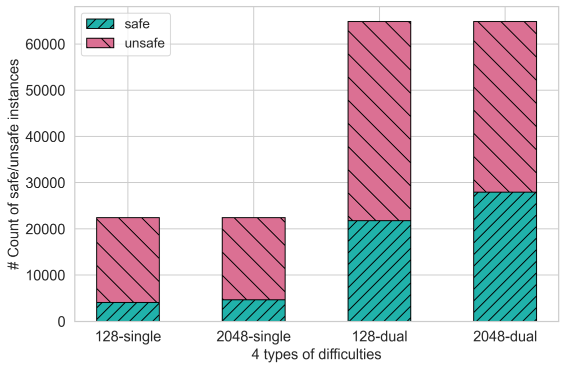

Figure 5. Count of safe and unsafe instances across the four difficulties, verified by α, β-CROWN.

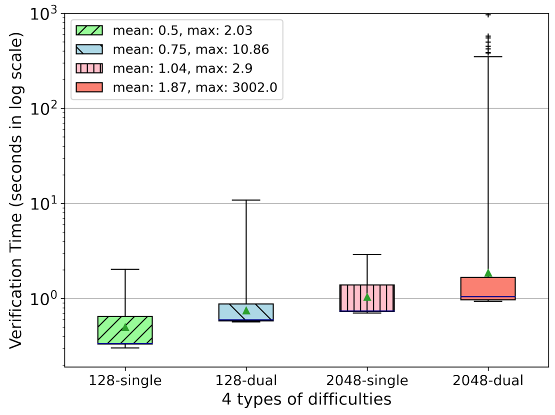

Figure 6. Verification time on specifications of various difficulties.

4.2. Learned cardinality

Based on training data and testing data (i.e. SQL queries) used in (Kipf et al., 2018), we generate 22384 specifications following property 1, 2, 3, and 5 of the learned cardinality (§3.2) and 64864 specifications following property 4. In this experiment, we tune parameter (1) and (4) described in Section 3.4 to form specifications in four difficulties (128-single, 128-dual, 2048-single, 2048-dual), where networks have different hidden state width 128/2048 and the same network will be copied as a dual-model when verifying specifications of property 4.

We use the verification time per instance of specifications to measure how hard is one instance to the existing verifier, so as to evaluate the effectiveness of our parameters to tune the difficulty. According to experimental results demonstrated in Figure 5, no more than half of the specifications are safe (i.e., successfully verified) for each group. For model with either 128 or 2048 dimension, the proportion of safe instances for monotonicity specifications are higher than normal specifications. Also, whether the specification is safe does not show clear correlation with the model size. Figure 6 reports the distribution of verification time cost by specifications in the four difficulties. We observe that by increasing the model size or performing monotonicity verification raise the verification time, proving the effectiveness of our benchmark parameters.

5. Future work, discussion, and summary

Future work. We're working on adding more NN4Sys instances to NN4SysBench, for example, learned schedulers, I/O latency predictors, and learned memory allocators. We also plan to include other categories of specifications, and develop sophisticated specifications that have more structures than a simple parallel OR. In the near future, we will conduct a more comprehensive performance analysis of the existing verification approaches on NN4SysBench.

Hints for NN-verification from NN4Sys. Looking forward, we believe the following optimizations can significantly boost neural network verification of NN4Sys. First, *batch verification* will help verifying multiple (potentially, many) specification entries at the same time. Second, it would be better to have *native support for verifying monotonicity* than our dual-model design (§3.2.2). Third, *end-to-end verification of multiple models* can substantially improve the training time of NN4Sys because as learned index shows, training a RMI is much cheaper than training a single network with the same accuracy (§3.1).

Summary. 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

Bak, S., Liu, C., and Johnson, T. T. The second international verification of neural networks competition (vnn-comp 2021): Summary and results. *ArXiv*, abs/2109.00498, 2021.

Chen, L., Lingys, J., Chen, K., and Liu, F. Auto: Scaling deep reinforcement learning for datacenter-scale automatic traffic optimization. In *Proc. SIGCOMM*, 2018.

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.

Gent, I. P. and Walsh, T. Csplib: A benchmark library for constraints. In Jaffar, J. (ed.), *Principles and Practice of Constraint Programming – CP'99*, pp. 480–481, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. ISBN 978-3-540-48085-3.

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.

Hashemi, M., Swersky, K., Smith, J., Ayers, G., Litz, H., Chang, J., Kozyrakis, C., and Ranganathan, P. Learning memory access patterns. In *International Conference on Machine Learning*, 2018.

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.

Katz, G., Barrett, C., Dill, D. L., Julian, K., and Kochenderfer, M. J. Reluplex: An efficient smt solver for verifying deep neural networks. In *Proc. CAV*, 2017.

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.

Krishnan, S., Yang, Z., Goldberg, K., Hellerstein, J., and Stoica, I. Learning to optimize join queries with deep reinforcement learning. *arXiv preprint arXiv:1808.03196*, 2018.

Liu, H., Xu, M., Yu, Z., Corvinelli, V., and Zuzarte, C. Cardinality estimation using neural networks. In *Proceedings of the 25th Annual International Conference on Computer Science and Software Engineering*, pp. 53–59, 2015.

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., 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. 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*, 2020.

Qiu, Y., Liu, H., Anderson, T., Lin, Y., and Chen, A. Toward reconfigurable kernel datapaths with learned optimizations. In *Proc. HotOS*, 2021.

Salman, H., Yang, G., Zhang, H., Hsieh, C.-J., and Zhang, P. A convex relaxation barrier to tight robustness verification of neural networks. *Advances in Neural Information Processing Systems*, 32:9835–9846, 2019.

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, C., Zhu, Y., and Guo, C. Building verified neural networks with specifications for systems. In *Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems*, 2021.

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.

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, NN4Sys benchmark. https://github.com/stanleybak/vnncomp2021/blob/main/benchmarks/nn4sys/, 2021.

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, 2020a.

Xu, K., Shi, Z., Zhang, H., Wang, Y., Chang, K.-W., Huang, M., Kailkhura, B., Lin, X., and Hsieh, C.-J. Automatic perturbation analysis for scalable certified robustness and beyond. *Advances in Neural Information Processing Systems*, 33, 2020b.

Xu, K., Zhang, H., Wang, S., Wang, Y., Jana, S., Lin, X., and Hsieh, C.-J. Fast and Complete: Enabling complete neural network verification with rapid and massively parallel incomplete verifiers. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=nVZtXBI6LNn.

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.