Building Verified Neural Networks with Specifications for Systems

Cheng Tan, Yibo Zhu, and Chuanxiong Guo
ByteDance Inc.

Abstract

Neural networks (NNs) are beneficial to many services, and we believe systems—such as OSes, databases, networked systems—are not an exception. However, applying NNs in these *critical* systems is challenging: people have to risk getting unexpected outcomes from NNs since NN behaviors are not well-defined. To tame these uncertain behaviors, we introduce a framework OUROBOROS which enables system developers to build *verified* NNs that follow user-defined specifications. These specifications comprise input and output constraints which characterize the behaviors of a NN. We do a case study on database learned indexes to demonstrate that training verified NN models is possible. Though many challenges remain, OUROBOROS enables us, for the first time, to apply NNs in critical systems *with confidence*.

ACM Reference Format:
Cheng Tan, Yibo Zhu, and Chuanxiong Guo . 2021. Building Verified Neural Networks with Specifications for Systems. In *ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '21), August 24–25, 2021, Hong Kong, China.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3476886.3477508

1 Introduction

Neural networks (NNs) have been widely used, and many applications and services benefit from applying them. We believe systems (like OSes, databases, networked systems) are not an exception. But, one major challenge to adopt NNs in these critical systems is NN's uncertainty: NNs do not have well-defined behaviors and they may produce unexpected results [8, 12]. Such uncertainty is particularly dangerous for critical components in a system, where any unexpected behavior may result in an incorrect system state.

NNs are complicated black boxes that are difficult to reason. Restraining NN's uncertainty by adding constraints either directly to NNs or to training process is hardly conceivable, as people have limited understanding of NN internals. In fact, it is notoriously hard to explain NN's behaviors, and explainable artificial intelligence [3, 6] is an active research topic.

Despite the uncertainty nature of NNs, there is a technique—*neural network verification* (NN-verification)—that can help. NN-verification verifies whether a NN satisfies some given properties. In particular, given a pair of input and output constraints (denoted as $\mathcal{X}$ and $\mathcal{Y}$, respectively), NN-verification checks whether a NN holds the following property: if an input x meets the input constraints ($x \in \mathcal{X}$), then the corresponding output y must satisfy the output constraints ($y \in \mathcal{Y}$).

In principle, NN-verification can help verify (hence build) NNs that meet pre-specified properties. But, in practice, NN-verification has two major limitations. First, NN-verification is powerful but expensive [1, 12]. Though researchers have made significant progress [4, 8, 13, 22, 23] in accelerating the verification, NN-verification still takes a long time (in hours or even days) to verify large NNs. Second, it is challenging to specify the expected NN behaviors by input and output constraints only. This is especially true for those tasks which do not have canonical outputs for unseen inputs. As an example, for recommending movies to a new user, it is unclear what input/output constraints should be considered as "expected".

However, we observe and argue that *NNs for systems* [7, 10, 11, 14, 15, 20, 21] are a perfect fit for NN-verification: NNs for systems are usually small and have clear semantics for inputs and outputs. Take database learned indexes [10] as an example. Their NN models are tiny with fewer than 100 neurons, hence are cheap to verify. Furthermore, NNs have clear semantics and their expected behaviors are unambiguous—NN's inputs are database keys and outputs are data positions on the underlying storage (e.g., disks); we expect that NN's outputs (predicted positions) should not be too far away from data's true positions.

We believe our observation of NN for systems is generally true, as systems often require NNs to run fast (hence NNs must be tiny) and systems' inputs and outputs have clear semantics (hence NNs have unambiguous expected behaviors). In addition, we believe many system components can benefit from applying NNs, including memory allocation [14], database query planning [11], memory prefetching [7], circuits design [21], and datacenter scheduling [15].

To build verified NNs for systems, our main idea is to use NN-verification as a verifier and check whether a trained NN satisfies user-defined properties, which we call a *specification*. If the NN passes the verification, we have a verified NN that

APSys '21, August 24–25, 2021, Hong Kong, China

ACM ISBN 978-1-4503-8698-2/21/08…$15.00
https://doi.org/10.1145/3476886.3477508

has well-defined behaviors characterized by the specification; otherwise, we retrain the NN.

This paper introduces a framework, OUROBOROS, that achieves the aforementioned idea. OUROBOROS takes the NN, data, and the specification as inputs to train a verified model. In the training process, OUROBOROS checks whether the current candidate NN satisfies the specification. If it does, OUROBOROS outputs the model; if it doesn't, OUROBOROS generates *specification-aware data* from the counterexamples which violate the specification, and retrains the model with these specification-aware data until it passes the verification.

We did a preliminary case study with database learned index [10], which is the first (arguably) practical NN-based index structure for databases. We reproduce the NNs described in their paper [10], and design a specification for the learned index. That is, for all keys, the NN's predicted positions in the database are at most ϵ slots away from their true positions, where ϵ is an error bound provided by users (§3.2).

Note that a verified NN does not outperform unverified ones in terms of model accuracy or inference speed. After all, they have the same architecture (i.e., NN computation graph). Instead, a verified NN has well-defined behaviors—*all* its outputs follow specifications regarding the inputs, and this is formally verified by OUROBOROS.

Though our case study shows that training a verified NN is possible, many challenges arise and lots of research questions are opened. We list a few here (more in §6):

- Our current retrain process is tedious, and there is no guarantee that users will finally have a verified model. For example, a specification might be too strict, and retrains cannot succeed. So, providing some types of guarantees to the retrain process is our future work.
- Though systems have clear semantics, describing specifications still requires significant manual efforts. It will be helpful to have common primitives for describing specifications, but how to design these primitives is unclear. Another topic is to automatically generate specifications.
- Since training verified NNs is possible, we are interested in discovering new components in critical systems that can benefit from being replaced by verified NNs, some of which were impossible due to NN's undefined behaviors.

Despite all the above challenges and open questions, OUROBOROS gives us a way, for the first time, to train a verified NN with specifications. It opens a new dimension in system design where developers can *safely* replace critical system components with NNs.

2 Background

In this section, we provide some necessary background for NN-verification (§2.1), and introduce learned index, which uses NNs as database indexes (§2.2).

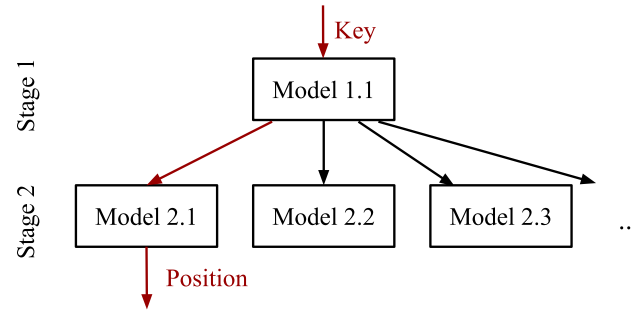

Figure 1. A two-stage Recursive Model Index (RMI). In this example, this RMI takes a "Key" as an input, chooses "Model 1.1" and "Model 2.1" for prediction, and finally produces the "Position" which is supposed to be the data location indexed by "Key" in the database.

2.1 Neural network verification

NN-verification [12] is a technique that formally verifies whether a NN satisfies a specification. A specification comprises a set of input/output constraint pairs; each pair represent a statement that if inputs (of the NN) satisfy the input constraint, then the corresponding outputs must satisfy the output constraint. If the NN meets all constraints in the specification, NN-verification accepts. Otherwise, NN-verification rejects and provides counterexamples.

Formally, consider a NN as a function f whose inputs are $x \in \mathcal{D}_x \subseteq \mathbb{R}^n$ and outputs are $y \in \mathcal{D}_y \subseteq \mathbb{R}^m$, where n and m are the input and output dimensions. We denote a pair of input/output constraint as $\langle x \in \mathcal{X}, y \in \mathcal{Y} \rangle$, where $\mathcal{X}$ and $\mathcal{Y}$ are subsets of the input and output domains ($\mathcal{X} \subseteq \mathcal{D}_x$ and $\mathcal{Y} \subseteq \mathcal{D}_y$). The problem of NN-verification is to check whether the following assertion holds for f (the NN):

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

NN-verification is already in use in practice. One example is to verify NNs in airborne collision avoidance systems [8]. These systems are used by aircraft to avoid midair collisions. In this scenario, inputs are sensor data including distance, speed, heading angle of the aircraft itself and other intruder aircraft; outputs are action advisories, including clear-of-conflict, weak left/right, strong left/right. And the specifications are manually defined action properties.

Multiple approaches are available for verifying NNs, including reachability [23], optimization [8, 13], search [4], and combinations of these techniques [22]. Our design uses NN-verification as a black-box, thus we omit NN-verification's technical details in this paper. We refer readers to this survey [12] for details. But it is worth noting that different approaches have diverse goals and varied performance. Choosing a suitable verification method is a key factor of successfully training a verified NN (§4).

2.2 Learned index

Index structures are widely used in systems which enables efficient data accesses. For example, B-Tree is a type of index which allows a database to quickly pinpoint data positions on the underlying storage. As an alternative, *learned index structure* [10] uses NNs to replace B-Trees for better performance.

Next, we introduce a type of learned index, named *Recursive Model Index* (RMI), which is depicted in Figure 1. RMI has multiple stages and each stage has one or multiple models (NNs). During a key 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 key. As a best practice [10], people use two-stage RMIs.

One challenge for RMIs is to ensure that models *always* produce data positions within certain error-bounds, 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 NN models, and replace those exceeding the error-bounds with traditional B-Trees. But, this approach does not provide guarantees for *non-existing* keys—the predicted data positions can be arbitrary.[1] This affects range queries whose upper/lower bound of the range might be non-existing keys.

It will be great if we can somehow know whether NN models have bounded errors for *all* keys (including non-existing keys). NN-verification can help. Given a specification asserting that predicted positions are within an error-bound, NN-verification can comprehensively check whether models hold this property for all keys. In addition, by using the counterexamples from NN-verification, we can retrain NNs to achieve the desired error-bounds, as we will show in section 3.2.

3 System design

In section 3.1, we introduce OUROBOROS, a framework to train verified NNs. We further show a case study of training a verified RMI in section 3.2.

3.1 OUROBOROS overview

OUROBOROS trains verified NNs that satisfies a pre-defined specification. Figure 2 depicts the system's workflow.

First, users provide a set of data that the NN will be trained on and a specification that describes the required properties of the NN. After receiving the data, OUROBOROS starts a normal training process and produces a candidate model. Then, the model is sent to an NN verifier which comprehensively checks whether the NN satisfies the specification for any possible inputs. The NN verifier achieves this by running NN-verification algorithms (§2.1).

If the NN verifier accepts, then OUROBOROS builds what users want—a model with formally verified properties (the specification). If the NN verifier rejects, it generates counterexamples that violate the specifications: data satisfying the input constrains whose corresponding outputs do not meet the output constraints. Formally, counterexamples are a set of data $\{x' \mid x' \in \mathcal{X} \wedge f(x') \notin \mathcal{Y}\}$ where $\mathcal{X}$ and $\mathcal{Y}$ are input and output constraints (§2.1).

[1]As discussed in the RMI paper [10, §3.4], original RMIs can handle non-existing keys by forcing all NN models to be monotonic or using exponential search techniques, which require extra efforts.

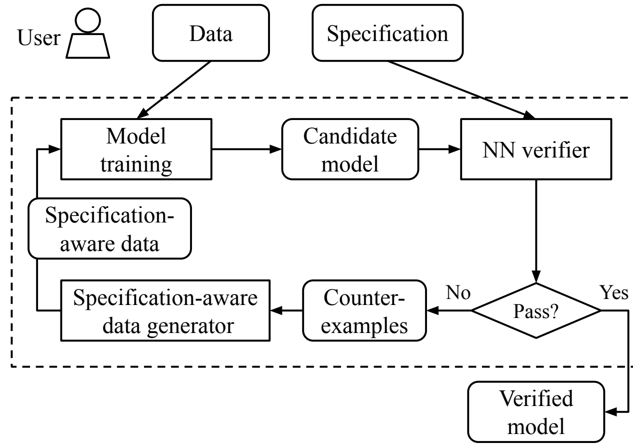

Figure 2. OUROBOROS's workflow. The dotted frame is OUROBOROS system. Users provide data and specifications and receive verified models that satisfy the specifications.

By analyzing the counterexamples, OUROBOROS generates *specification-aware data* that represents the edge cases whose outputs should have been included in $\mathcal{Y}$. With these specification-aware data, OUROBOROS retrains a new candidate model and verifies whether the new model satisfies user's specification. In our current implementation, the specification-aware data generator is a function that simply pairs counterexamples (a set of keys) with the positions that the keys should be if they were inserted into the database (see also §3.2 and §6).

OUROBOROS runs this process multiple rounds until getting a model that passes NN-verification, then we have a verified model. Or we fail to train one, if OUROBOROS times out.

3.2 Case study: training a verified RMI

In this section, we study how to build a verified RMI such that all predicted positions—including predictions for non-existing keys—are within a predefined error bound (denoted as ϵ). We first elaborate the specification for a two-stage RMI, and then introduce how to train a verified RMI using NN-verification.

Our current approach, as a starting point, is rudimentary (see our future work in §6). The main takeaway is that we are able to build a verified RMI by using NN-verification.

This case study uses the Integer Datasets [10, §3.7.1], in which keys and positions are both integers. The stored data are sorted by their keys, a common scenario in databases for supporting range queries. We follow the best practice RMI which has two stages: stage 1 has one NN model with two 16-neuron fully connected layers, and ReLU as activation functions; stage 2 has 10 linear models. The RMI's input is an integer (key) and the prediction is also an integer (position). See an RMI example in Figure 3.

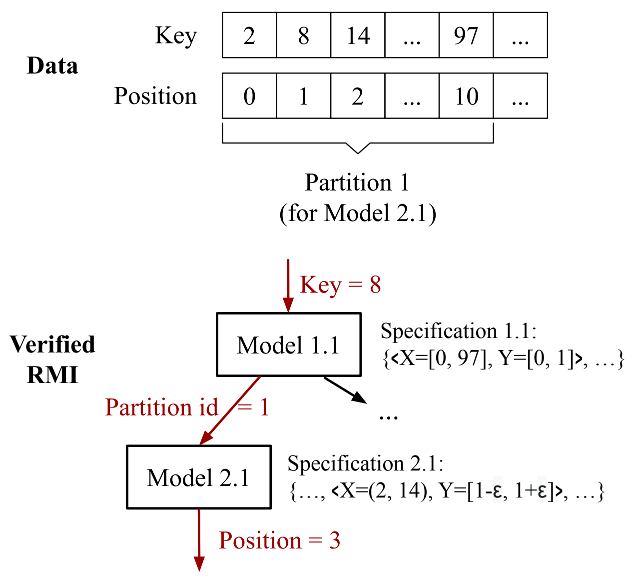

Figure 3. An example of a RMI. On the top is the (sorted) data that the RMI indexes. On the bottom is a verified two-staged RMI with specifications for each model. In this example, the RMI predicts that the data indexed by "Key=8" should locate at "Position=3", and the true position is "1" (see "Data"). If the user-defined error bound is less than 2 ($\epsilon < 2$), this RMI will be rejected by NN-verification because "Model 2.1" does not satisfy "Specification 2.1".

The required property is as follows. For all keys, the RMI's predicted positions must be at most ϵ slots away from its true position (for existing keys) or the position that the data should be (for non-existing keys).

RMI specifications. To design a specification for the required property, we partition the existing key space into 10 parts, each of which is assigned to one model in stage 2. That is, partition 1 is assigned to the first stage 2 model (Model 2.1 in Figure 3); partition 2 is assigned to the second model, and so forth.

Now, we define specifications for models of different stages separately. First, for the model of stage 1, if a key x is within the key range of partition i, then the prediction, which indicates the next model in stage 2, must be in range $[i-1, i+1]$. This means that the prediction of the next model is at most one "slot" away from the model it should be. Second, for a model in stage 2 (e.g., Model 2.1), we check whether all keys corresponding to this model satisfy the property that the predicted positions are within ϵ-slot away from their true positions. The corresponding keys include the partition assigned to this model (e.g., Partition 1 for Model 2.1) and the keys that are "accidentally" assigned by the stage 1 model (notice that the error bound for the prediction of Model 1.1 is one-slot-away).

Take specifications in Figure 3 as an example. In specification 1.1, the first constraint pair depicts input/output constraints for partition 1, which is $\langle \mathcal{X} = [0, 97], \mathcal{Y} = [0, 1] \rangle$. This constraint pair reads as, if a key $x \in \mathcal{X}$, then the prediction of Model 1.1 must be in $\mathcal{Y}$. Notice that $[0, 97]$ is the key range for partition 1, and the range $[0, 1]$ represents that either the first or second model in stage 2 will be used.

In specification 2.1, $\langle \mathcal{X} = (2, 14), \mathcal{Y} = [1-\epsilon, 1+\epsilon] \rangle$ reads as: if a key is in the range of $(2, 14)$, then the predicted position must be within $[1-\epsilon, 1+\epsilon]$, where "1" is the true position of "Key=8" and ϵ is the error bound. As we can see in Figure 3, the range $(2, 14)$ captures the existing "Key=8" and all non-existing keys that would have been placed immediately before or after this key. And, the predicted position of all these keys should be at most ϵ slots away from the true position of "Key=8", which is position "1".

Training a verified RMI. In the first round, OUROBOROS trains the RMI as described in the original RMI paper [10, Algorithm 1]. OUROBOROS uses the whole data to train the model in stage 1. For stage 2 models, OUROBOROS train each model with their corresponding partitions and the keys that are in adjacent partitions but assigned to this model by stage 1 model.

After getting a candidate RMI, instead of evaluating the existing keys and replace unfulfilled NNs with B-Trees, OUROBOROS uses NN-verification to check whether the trained RMI satisfies the specifications. OUROBOROS's verifier adopts a verification toolbox named `NeuralVerification.jl` [2], and chooses a solver ReluVal [22]. The verifier encodes the constraint ranges in specifications to `Hyperrectangle`, and invokes the ReluVal solver to verify the specifications.

If the verifier rejects, it returns a counterexample x', which often is a non-existing key. OUROBOROS calculates the true corresponding position y' for x' by pretending to insert x', and then adds the data point (x', y') into the training data. With the new data, OUROBOROS retrains the models. A future work is to design an algorithm to generate a set of new specification-aware data from few counterexamples. OUROBOROS repeats training until finding an RMI that passes the NN-verification.

4 Preliminary results

Implementation. We reimplement RMIs based on the description in the paper [10] with 500 lines of Python and Tensorflow code. We build the NN-verification in 140 lines of Julia code on top of a verification toolbox NeuralVerification.jl [2]. Finally, we use a bash script (50 lines) to coordinate training and verification. All the experiments were executed on a MacBook pro with a 2.6 GHz 6-Core Intel i7 CPU and 32GB memory.

Dataset, RMI, and specification. In this experiment, we use a synthetic dataset, Integer Datasets [10, §3.7.1], which has 190K unique integer values. These values are randomly sampled from a range of 0 to 1M, and are stored in a sorted array. Here values serve as keys (inputs to NNs), and their positions in the sorted array represent data positions in a database (outputs of NNs).

The major difference between our version and the original Integer Dataset is that we sample data uniformly random rather than in a lognormal distribution. We do this for two reasons. First, randomly sampled dataset is an easier case, and we want to start from the most basic dataset. Second, even for this easy dataset, training a verified RMI is challenging. As we will show later, a stage 2 model was retrained 13 times before it finally passes the verification, with relaxing ϵ twice (elaborate below).

As mentioned earlier, we follow the best practice RMI, which has 2 stages: stage 1 has one NN model with two fully-connected layers with 16 neurons each. The activation function is ReLU. Stage 2 has 10 linear models.

We implement specifications for models of stage 1 and stage 2 separately. For the specification of stage 1, there are 10 pairs of input/output constraints, each of which represents a partition. The specifications for stage 2 models contain 190k constraint pairs in total, and each pair represent a data point's range (see §3.2). On average, a single stage 2 specification has 19k pairs of input and output constraints.

Training a verified RMI. We follow the RMI training procedure as described in the original paper [10, Algorithm 1], except that we replace evaluating existing keys with NN-verification. During training, the stage 1 model passes verification in the first round (no retrain needed). This aligns with our expectation because it is easy to learn the general trend of a dataset without caring too much about details. The original RMI paper has similar observations.

On the contrary, stage 2 models are much harder to train (the "last mile problem"). We start stage 2 model training with $\epsilon = 100$; meanwhile, if a model is retrained five times and still fails the verification, we increase ϵ by 50 for this model's specification, which we call *ϵ-relaxation*. Also, note that $\epsilon = 100$ is a strong error bound in practice, as databases usually group data into blocks. Two positions differ by 100 are probably still in the same block, or in two consecutive blocks.

In stage 2 model training, six out of ten models pass in the first round without retraining ($\epsilon = 100$). Three models are retrained six times and end up with $\epsilon = 150$. One model is particularly hard to train and is retrained 13 times. It finally passes the verification with $\epsilon = 200$.

Verification performance. We run NN-verification with a ReluVal solver [22]. In our experiment, verifying the model of stage 1 takes 4 seconds, and verifying all stage 2 models takes 7 seconds; that is 0.7 second each.

Beyond ReluVal, we also experimented with another solver, BaB [4]. However, BaB is much slower than ReluVal in our case: it spends 18 and 155 seconds for verifying stage 1 and 2, respectively. This experiment shows that choosing a suitable solver is critical for verification performance.

5 Related work

The most related work to OUROBOROS is NeVer [19], a pioneer of NN-verification proposed a decade ago. NeVer has several innovations. The one closely related to OUROBOROS is a technique called *counterexample triggered abstraction-refinement* (*CETAR*), namely using counterexamples to repair the behavior of a NN. If we ignore the technical details (for example, NeVer uses abstract interpretation hence requires refinements, whereas OUROBOROS doesn't), OUROBOROS's retraining process is a reminiscence of CETAR, but in the context of building NNs for systems. The major difference is that OUROBOROS needs to design specifications according to system semantics, for example, RMI specifications (§3.2), which is non-trivial and requires system's domain knowledge.

Several recent works [5, 9] use NN-verification to check and/or visualize (unexpected) behaviors of NNs used in networked systems. Ouroboros uses similar techniques but studies a different problem; we focus on building verified NNs instead of testing or analyzing NNs. Consequently, OUROBOROS meets more challenges (which are also opportunities), including retraining NNs with specification-aware data, designing generalized specification primitives, partial and incremental verifications (more detailed discussion in §6).

NN-verification's applications. As mentioned in section 2.1, NN-verification has been extensively studied [4, 8, 12, 13, 22, 23], and has several main applications, including verifying the robustness of NNs against adversarial attacks [16] and ACAS Xu [8], an airborne collision avoidance system. One can find standardized benchmarks of these applications in VNN-COMP [1], a NN-verification competition.

In this paper, we observe that NNs for systems [7, 10, 11, 14, 15, 20, 21] is a new category of applications for NN-verification, which is a perfect fit to verify because NNs for systems are usually small (hence easy to verify) and have unambiguous semantics (hence their behaviors are straightforward to specify). We argue that verifying NNs for systems expands NN-verification's applications, and further NN-verification can significantly improve the safety of NNs for systems.

Testing NNs. There is a line of work [17, 18] to detect misbehavior of NNs by testing (see more machine learning testing in this survey [24]). For example, DeepXplore [17] is a whitebox testing framework that methodically construct the dataset for testing to reach high *neuron coverage*, a metric representing how much proportion of the tested NN is covered by the test. These are practical and efficient tools for discovering unexpected behaviors. However, testing cannot prove the absence of misbehavior, whereas NN-verification (hence OUROBOROS) can. Ouroboros provides a rigorous guarantee that a verified NN always follows the pre-defined specifications.

6 Discussion, limitations, and future work

Current OUROBOROS's design has multiple limitations, which inspire our future work. We discuss three of them in detail below. We also briefly mention other interesting topics at the end of this section.

First, OUROBOROS provides no guarantee on finally producing a verified NN. OUROBOROS does not explicitly manage the training process, except for adding specification-aware data (counterexamples) to the training dataset. Our future work is to design a specification-oriented training, which micro-manages the training steps with specifications in mind. Another approach is to design a good specification-aware data generating algorithm that generates representative data to "push" the model training in the right direction.

Second, OUROBOROS's current specification is ad hoc. Our future work is to design a domain specific language (DSL) for specifications. This DSL will provide pre-defined primitives for common systems (for example, OS, network, storage), which assist developers in specifying their wanted properties.

Also, it is unclear whether specification development will be a heavy burden for developers. In the case of learned index, the specification is straightforward, but this might not be always true—we could imagine a non-trivial task to design safety specifications for some complicated components, for example, database concurrency control.

Finally, efficiency training is always desirable, but OUROBOROS suffers from many rounds of retraining, which include normal NN training and NN-verification. Our future work is to accelerate this retraining process by incremental training and incremental verification. The guiding intuition is that, for each round of retraining, we do not tackle a completely new problem. Instead, we face a similar problem with updates on some parameters: updated training dataset for NN training and updated NN models for NN-verification. There ought to be a way to leverage the information from previous rounds for accelerating the retraining, which requires further research.

Beyond the above three directions, many other topics are worth exploring as well. For example, what systems can benefit from OUROBOROS, and which components can be replaced by verified NNs? Since performance of different NN-verification techniques varies, how can OUROBOROS choose the verifier's solver that fits current workload the best? Can OUROBOROS detect whether a specification is too strict to achieve? If so, OUROBOROS can directly ask developers to revise, without having to waste cycles on retraining.

Summary. To recap, OUROBOROS is the first framework that empowers us to train a verified NN with user-defined specifications. We believe this will significantly broaden the ways that people apply NNs in today's systems because now developers can have *faith* in their NNs.

References

[1] competition for neural network verification (VNN-COMP). https://sites.google.com/view/vnn20/vnncomp.

[2] Neuralverification.jl. https://github.com/sisl/NeuralVerification.jl.

[3] A. B. Arrieta, N. Díaz-Rodríguez, J. Del Ser, A. Bennetot, S. Tabik, A. Barbado, S. García, S. Gil-López, D. Molina, R. Benjamins, et al. Explainable artificial intelligence (xai): Concepts, taxonomies, opportunities and challenges toward responsible ai. *Information Fusion*, 58:82–115, 2020.

[4] R. Bunel, I. Turkaslan, P. H. Torr, P. Kohli, and M. P. Kumar. A unified view of piecewise linear neural network verification. *arXiv preprint arXiv:1711.00455*, 2017.

[5] A. Dethise, M. Canini, and N. Narodytska. Analyzing Learning-Based Networked Systems with Formal Verification. In *Proceedings of INFOCOM'21*, May 2021.

[6] F. K. Došilović, M. Brčić, and N. Hlupić. Explainable artificial intelligence: A survey. In *2018 41st International convention on information and communication technology, electronics and microelectronics (MIPRO)*, 2018.

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

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

[9] Y. Kazak, C. Barrett, G. Katz, and M. Schapira. Verifying deep-rl-driven systems. In *Proceedings of the 2019 Workshop on Network Meets AI & ML*, pages 83–89, 2019.

[10] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, 2018.

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

[12] C. Liu, T. Arnon, C. Lazarus, C. Barrett, and M. J. Kochenderfer. Algorithms for verifying deep neural networks. *arXiv:1903.06758*, 2019.

[13] A. Lomuscio and L. Maganti. An approach to reachability analysis for feed-forward relu neural networks. *arXiv preprint arXiv:1706.07351*, 2017.

[14] M. Maas, D. G. Andersen, M. Isard, M. M. Javanmard, K. S. McKinley, and C. Raffel. Learning-based memory allocation for c++ server workloads. In *Proc. ASPLOS*, 2020.

[15] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh. Learning scheduling algorithms for data processing clusters. In *Proc. SIGCOMM*. 2019.

[16] C. Müller, G. Singh, M. Püschel, and M. Vechev. Neural network robustness verification on gpus. *arXiv preprint arXiv:2007.10868*, 2020.

[17] K. Pei, Y. Cao, J. Yang, and S. Jana. Deepxplore: Automated whitebox testing of deep learning systems. In *proceedings of the 26th Symposium on Operating Systems Principles*, pages 1–18, 2017.

[18] K. Pei, Y. Cao, J. Yang, and S. Jana. Towards practical verification of machine learning: The case of computer vision systems. *arXiv preprint arXiv:1712.01785*, 2017.

[19] L. Pulina and A. Tacchella. Never: a tool for artificial neural networks verification. *Annals of Mathematics and Artificial Intelligence*, 62(3):403–425, 2011.

[20] S. Salman, C. Streiffer, H. Chen, T. Benson, and A. Kadav. Deepconf: Automating data center network topologies management with machine learning. In *Proceedings of the 2018 Workshop on Network Meets AI & ML*, 2018.

[21] H. Wang, J. Yang, H.-S. Lee, and S. Han. Learning to design circuits. *arXiv preprint arXiv:1812.02734*, 2018.

[22] S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana. Formal security analysis of neural networks using symbolic intervals. In *Proc. USENIX Security*, 2018.

[23] W. Xiang, H.-D. Tran, and T. T. Johnson. Reachable set computation and safety verification for neural networks with relu activations. *arXiv preprint arXiv:1712.08163*, 2017.

[24] J. M. Zhang, M. Harman, L. Ma, and Y. Liu. Machine learning testing: Survey, landscapes and horizons. *IEEE Transactions on Software Engineering*, 2020.