Building Verified Neural Networks for Computer Systems with Ouroboros

Tianhao Wei [1] **Zhihao Jia** [1] **Changliu Liu** [1] **Cheng Tan** [2]

Abstract

Neural networks are powerful tools. Applying them in computer systems—operating systems, databases, and networked systems—attracts much attention. However, neural networks are complicated black boxes that may produce unexpected results. To train networks with well-defined behaviors, we introduce OUROBOROS, a system that constructs *verified neural networks*. Verified neural networks are those that satisfy user-defined safety properties, known as specifications. OUROBOROS builds verified networks by a training-verification loop that combines deep learning training and neural network verification. The system employs multiple techniques to fill the gap between today's verification and the properties required for systems. OUROBOROS also accelerates the training-verification loop by spec-aware learning. Our experiments show that OUROBOROS can train verified networks for five applications that we study and has a 2.8× speedup on average compared with the vanilla training-verification loop.

1 Introduction

Deep learning and neural networks are powerful. They have contributed to many fields, including computer vision, natural language processing, and speech recognition. Naturally, it attracts increasing attention to apply deep learning techniques to tasks in computer systems. For example, neural networks have been used for database indexes (Kraska et al., 2018), Internet 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), I/O latency prediction (Hao et al., 2020), and circuit design (Wang et al., 2018a).

However, it remains challenging to use neural networks in critical components of systems. This is because the correctness of neural networks is defined by statistical performance on a dataset but not by deterministic behavior on specific rules. As a result, neural networks are generally treated as black boxes that may produce arbitrary results on unseen inputs. For example, a neural network based scheduler may attempt to schedule a job that is invalid (Mao et al., 2016).

The problem cannot be addressed solely by enhancing training. On the one hand, the training dataset cannot enumerate all possible valid cases. For example, if one wants to train a neural network as a database index, the training dataset is unable to cover all possible keys (both existing and non-existing keys) because there are infinitely many of them. On the other hand, training cannot prevent neural networks from producing unexpected results: even if all training data follows a safety rule, the network may violate the rule. For example, though all training data is sampled from a monotonic function, the neural network may fail to be monotonic. Wang et al. (2020) have seen this in learned cardinality estimation.

The lesson learned is that traditional training cannot enforce a neural network to obey specific rules. In this paper, we ask: *how to train a neural network that verifiably complies with user-provided safety rules?*

Prior work has introduced *override rules* (Mao et al., 2016; Katz, 2020) that update network outputs postmortem if they are invalid. However, this approach does not work for many systems for three reasons. First, not all safety properties can be specified by override rules. Consider schedulers as an example. It is unclear how to describe the non-starvation requirement as an override rule to fix individual scheduling decisions. Second, override rules may defeat the benefit of using neural networks in the first place. Many systems use neural networks because networks have smaller memory footprints and run faster. For example, learned database indexes (Kraska et al., 2018) use ML models to replace B-trees for performance. However, with override rules, the learned index will run slower. Finally, as a separate component, override rules must be synchronous with neural networks, which is error-prone during maintenance (Katz, 2020).

Instead of fixing outputs postmortem, we directly train neural networks that strictly follow user-provided safety properties, which we call *verified neural networks*. To do so, we introduce a system, OUROBOROS. OUROBOROS combines deep learning training and neural network verification to train verified networks. OUROBOROS trains a network in a traditional approach and uses neural network verification

[1]CMU [2]Northeastern University. Correspondence to: Cheng Tan <c.tan@northeastern.edu>.

Proceedings of the 6th MLSys Conference, Miami Beach, FL, USA, 2023.

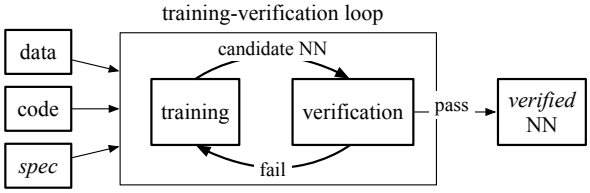

Figure 1: Training-verification loop overview. The training-verification loop requires normal training inputs ("data" and "code") and a specification ("spec"), and produces a verified neural network ("verified NN") that complies with the specification if the loop successfully finishes.

(NN-verification) to check if the candidate network satisfies user-provided properties. If yes, OUROBOROS successfully builds a verified neural network; otherwise, it iterates the training and verification. Figure 1 overviews this process.

The idea of this training-verification loop is not new. It has been extensively explored by multiple communities: in control theory, this is the classic feedback loop (Doyle et al., 2013); in programming language, this is called counterexample-guided inductive synthesis (Solar-Lezama, 2013); in deep learning, this is known as counterexample-guided learning (Dreossi et al., 2018) or certified robust training (Shi et al., 2021; Müller et al., 2022). Pulina & Tacchella (2011) and others (Gowal et al., 2018; Zhang et al., 2019a) have explore how to build robust neural networks.

Compare with prior work, OUROBOROS is the first (a) end-to-end system to train verified neural networks that (b) fully comply with (c) safety properties of computer systems. There are no prior systems that simultaneously provide (a)–(c). In particular, different from traditional robust models such as robust image classifiers (Bertsimas et al., 2011), neural networks for systems (short as NN4Sys) require *fully compliance* with the provided safety properties; otherwise, their usage in critical scenarios poses safety risks.

To build verified NN4Sys, there are two major challenges:

Challenge 1: verification of diverse properites. Ensuring correctness in computer systems such as operating systems, databases, and network systems requires sophisticated properties, which we call *specifications*. This differs from today's focus on robustness properties, which simply check for all inputs in an input domain, whether outputs fall within a given output domain. Through studying NN4Sys applications, we have observed tha developers also demand other categories of specifications, specifically, *probabilistic specifications* and *monotonicity specifications*, While techniques exist to address these specifications (Weng et al., 2019; Liu et al., 2020), no prior work has integrated all three specification verifications under a unified framework.

Challenge 2: full compliance with specifications. Another unique challenge of building verified neural networks for systems is achieving full compliance. Unlike traditional machine learning tasks, systems require specifications to be satisfied at all time, as the consequences of violating them can be severe. To achieve full compliance, the learning procedure must place emphasis on learning from the specifications; meanwhile, it is essential to balance the focus on specifications and learning from the training data, which represents average performance. It is a challenge to train networks that perform well on average cases and maintain reliability under worst cases.

In particular, this paper makes the following contributions:

- *Supporting diverse specifications (§4).* We integrate three categories of specifications—reachability specifications, probabilistic specifications and monotonicity specifications—into a unified framework to support constructing verified NN4Sys.
- *Spec-aware learning (§5).* We introduce spec-aware learning that accelerates learning from specifications by combining multiple techniques, including spec-aware data sampling and early rejection.
- *A built system and five verified NN4Sys (§6, §7).* We implement OUROBOROS as an end-to-end system and experiment with it on five existing applications (Kraska et al., 2018; Maas et al., 2020; Liu et al., 2015; Hao et al., 2020). OUROBOROS is able to construct verified versions of these applications.

Our experiments (§8) show that OUROBOROS accelerates the training-verification loop by up to 7.0× (2.8× on average), and the verified neural networks have comparable test set accuracy to the traditional networks on all five applications.

2 Motivation and Background

Neural networks for systems (NN4Sys) have recently attracted a lot of attention: people propose a broad range of applications, including databases, networked systems, and operating systems. We list a few below.

Neural networks for systems. Kraska et al. (Kraska et al., 2018) propose learned indexes for databases. They replace classic database indexes (like B-Tree) with neural networks. There is a line of work (Ding et al., 2020b;a; Tang et al., 2020; Marcus et al., 2020a) to optimize the performance of learned indexes and extend learned indexes to different environments. In addition, neural networks have also been used for database cardinality estimation (Liu et al., 2015; Wang et al., 2020) and query optimization (Krishnan et al., 2018; Marcus et al., 2019; 2020b). In networked systems, people use neural networks for 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., 2019),

optimizing video streaming (Mao et al., 2017), and packet classification (Liang et al., 2019). In operating systems, there are proposals to use neural networks for predicting I/O latency (Hao et al., 2020), page prefetching and job scheduling (Qiu et al., 2021). Also, Zhang & Huang (2019) explore the opportunities and challenges for building a "learned" operating system.

However, neural networks are black boxes and may produce unexpected results. For example, a learned scheduler may schedule an invalid job; an NN-based database index may exceed its error bounds when querying non-existing keys. One approach to verify the network behaviors is NN-verification.

Neural network verification. Neural network verification provides *formal* guarantees of neural network input-output properties: if an input satisfies a condition, then the corresponding output satisfies another condition. The promise of neural network verification is that given a trained neural network and an input-output property, a verifier either accepts (meaning the network satisfies this property) or rejects with a counterexample that violates the property.

In this paper, we call the above 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:

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

Neural network verification has been extensively studied. Existing neural network verification methods can be broadly categorized into reachability-based methods (Tran et al., 2020; Wang et al., 2018d; Gehr et al., 2018) and optimization-based methods (Lomuscio & Maganti, 2017; Tjeng et al., 2017; Raghunathan et al., 2018; Ehlers, 2017). We refer readers to Liu et al. (2019) for more verification techniques.

Ouroboros's verification. In our implementation (§7), ouroboros uses Neurify (Wang et al., 2018b). Although ouroboros taiolors algorithms to support other specifications (such as probabilistic and monotonicity specifications, §4), the core verification algorithm remains unchanged.

3 A case study: verified learned index

This section provides a case study of a verified learned index to illustrate the workflow of training a verified neural network for systems. The case study overviews (a) what specifications look like and (b) how training-verification loop works.

The problem. Database learned index (Kraska et al., 2018) is an attempt to use ML models to replace traditional index data structure such as B-tree, where the underlying data are sorted by keys. Inputs to learned indexes are database keys; outputs are predicted data positions. Because the predictions are not exact, the database needs to do a local search (e.g., binary search) to find the true position.

One challenge of using neural networks for indexing is that, for the infinitely many non-existing keys (the keys that are not in the training dataset), there is no guarantee about prediction errors because the network hasn't seen them. Therefore, instead of neural networks, recent works on learned index use piecewise linear models (Ferragina & Vinciguerra, 2020) or other monotonic functions (Marcus et al., 2020c) as the ML models. They work well for one-dimensional keys but face difficulties in handling multi-dimensional keys. Instead, we use a different approach: we use NN-verification to ensure that the prediction errors are bounded for non-existing keys.

Specifications. The correctness property of a learned index is as follows (Marcus et al., 2020a): for any given database key, including non-existing keys, the learned index's prediction should be within ϵ slots away from its true position (ϵ is a user-defined error bound). Note that because the underlying data are sorted, non-existing keys also have their true positions: between the keys immediately smaller and immediately larger than them.

To define the specifications for an entire learned index, we split the key space $\mathcal{K}$ according to existing keys, and assert that for keys within a range (e.g., $[\mathcal{K}[i], \mathcal{K}[i+1]]$), a prediction must not differ from the true position $DB(key)$ beyond an error bound ϵ. Formally, this specification reads as (f is the network; DB is the ground truth; ϵ is the error bound):

$$\begin{aligned} &\forall key \in [\mathcal{K}[i], \mathcal{K}[i+1]], \\ &\quad f(key) \in [DB(\mathcal{K}[i]) - \epsilon, DB(\mathcal{K}[i+1]) + \epsilon] \end{aligned} \tag{1}$$

Training-verification loop. As mentioned earlier (§1), the training-verification loop iterates the training and the verification procedures until the network passes the verification. When networks fail verification, the verifier returns counterexamples. Counterexamples of learned index are the keys (say $k \in [k_i, k_j]$) whose network outputs exceed the error bound (i.e., $f(k) < DB(k_i) - \epsilon$ or $f(k) > DB(k_j) + \epsilon$). We assign the correct labels to these counterexamples according to specifications, add them to the training dataset, and start the next round of training. This is an establish approach sometimes known as counterexample-guided learning (Dreossi et al., 2018). Similar approaches have also been explored elsewhere (Doyle et al., 2013; Solar-Lezama, 2013).

Data augmentation vs. regularization. In general, there are two main approaches to incorporate the feedback from verification: data augmentation (Pulina & Tacchella, 2011; Tan et al., 2021) and regularization (Zhang et al., 2019a; Fan & Li, 2021). Data augmentation guides learning by adding data (i.e., counterexamples) to training datasets; whereas,

regularization appends a regularization term to the loss function (sometimes called *robust loss*) for penalizing cases that fail verification.

OUROBOROS uses data augmentation. There are two main reasons why we choose data augmentation over regularization for OUROBOROS. First, verified NN4Sys requires *full compliance* with the specifications. For full compliance, data augmentation offers better performance as it allows us to fine-tune the percentage of training data generated from the specifications. Second, NN4Sys typically uses low-dimensional inputs (fewer than a dozen), which limits the search space of counterexamples and enables us to find high-quality spec-aware data.

To confirm our observation, we did an experiment using data augmentation (described above) and regularization (described below) to train the same monolithic network as a learned index. The network is a four-layer fully-connected network with 128-width each layer; the training data is a synthetic database with 145K integer keys sampled from a log-normal distribution. We implement our regularization similar to CROWN-IBP (Zhang et al., 2019a) with a robust loss term of $(y_{bound} - \hat{y})^2$, where y_{bound} is the upper/lower bound computed by verification and $\hat{y}$ is the true bounds (i.e., $DB(key) \pm \epsilon$). The data augmentation produces the verified learned index in 4 hours; and the regularization timed out after 24 hours of training.

Challenges and opportunities. The learned index is one example of NN4Sys. By studying multiple NN4Sys, we find two challenges to train verified networks. One is about specification capability and the second is about performance.

1. *Diverse specifications:* for many NN4Sys, the safety rules require more expressive specifications than reachability specifications. For example, cardinality estimation requires monotonicity, and bloom filter requires probabilistic specifications.
2. *Full compliance requires efficient learning from specifications:* it is inefficient to learn from few counterexamples each round, and run full verification each round. We do not have to use expensive verification if networks have superficial flaws; and we could augment the training dataset more efficiently by sampling the counterexamples according to specifications.

We propose techniques to address these two challenges in section 4 and 5, respectively.

4 VERIFYING COMMON SPECIFICATIONS FOR COMPUTER SYSTEMS

As mentioned earlier (§1), today's neural network verification techniques support a limited set of specifications. In this section, we introduce how OUROBOROS supports common

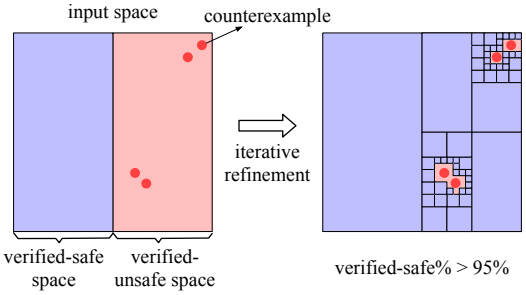

Figure 2: Probabilistic specification verification. The outermost squares represent the entire input space; each inner smaller rectangles represents an area that has been verified: blueish means verified safe space and redish means verified unsafe space. The dots are input data points that fail the specifications (i.e., counterexamples). In this example, the probabilistic specification threshold is 95%.

specifications that cover most neural networks for systems that we study (§6, Figure 5).

Common specifications. We study the existing neural networks for systems (Kraska et al., 2018; Maas et al., 2020; Dai & Shrivastava, 2019; Byun & Lim, 2021; Liu et al., 2015; Wang et al., 2020; Hao et al., 2020) and distill three categories of specifications: *reachability specifications*, *probabilistic specifications*, and *monotonicity specifications*. Admittedly, these specifications are not comprehensive. Nevertheless, they cover a wide range of applications (§6). Our future work is to discover more useful specification categories and draw a line of what can and cannot be expressed by these specification categories.

Both probabilistic and monotonicity specifications have been studied before. Weng et al. (2019) and Berrada et al. (2021) provide verifiable probabilistic guarantees for neural networks; Liu et al. (2020) and others (Sivaraman et al., 2020; Chen et al., 2021) explore monotonicity of neural networks. In OUROBOROS, we use *iterative refinements* (Wang et al., 2018c) to verify probabilistic and monotonicity specifications, which we will elaborate in section 4.1 and 4.2.

4.1 Verifying probabilistic specifications

Probabilistic specification describes a neural network obeying rules with some given probability. It is useful when either the guarantees hold probabilistically (for example, in probabilistic data structures) or users want to give some leeway to neural networks (namely, allowing false predictions). For example, bloom filters have probabilistic specifications (see details in §6). This is a category of specifications that prior neural network verification methods have little support. 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 $\langle \mathcal{X}, \mathcal{Y} \rangle$ (defined in §2) and a probability

```
1:  procedure VERIFYPROB(f, X, Y, prob):
2:      specs ← [⟨X, Y⟩]
3:      S_safe ← 0; S_unsafe ← 0
4:      // space(·) calculates the space of the given input
5:      while S_safe/space(X) < prob:
6:          X, Y ← specs.pop()
7:          if verify(f, X, Y) = ACCEPT:            // run Neurify
8:              S_safe += space(X)
9:          else:                                   // failing verification
10:             if space(X) < min_space:
11:                 S_unsafe += space(X)
12:                 if S_unsafe > (1 − prob): return REJECT
13:             else:
14:                 specs += DIVIDE(X, Y)           // line 34
15:     return ACCEPT
16:
17: procedure ISMONO(grad_lb, grad_ub, Y_dim, is_monoinc)
18:     if is_monoinc:
19:         return grad_lb[Y_dim] ≥ 0
20:     else:
21:         return grad_ub[Y_dim] ≤ 0
22: procedure VERIFYMONO(f, X, Y_dim, is_monoinc)
23:     grad_lb, grad_ub ← CalcGradientBounds(f, X)
24:     if ISMONO(grad_lb, grad_ub, Y_dim, is_monoinc): // line 17
25:         return ACCEPT
26:
27:     // if not monotonic, then refine the spec
28:     if X.ub − X.lb < min_threshold: return REJECT
29:     ⟨X1, _⟩, ⟨X2, _⟩ ← DIVIDE(X, NULL)          // line 34
30:     return VerifyMono(f, X1, Y_dim, is_monoinc) and
31:            VerifyMono(f, X2, Y_dim, is_monoinc)
32:
33:
34: procedure DIVIDE(X, Y)
35:     // find the dimension that has the largest gap
36:     dim_len, dim_id ← findmax(X.upbound − X.lowbound)
37:     // divide the dimension into two
38:     mid ← lb[dim_id] + dim_len/2
39:     X1 ← X.clone(); X1.ub[dim_id] ← mid
40:     X2 ← X.clone(); X2.lb[dim_id] ← mid
41:     return [ ⟨X1, Y⟩, ⟨X2, Y⟩]
42:
```

Figure 3: VERIFYPROB and VERIFYMONO describe how OUROBOROS verifies probabilistic specifications and monotonicity specifications, respectively. f represents the neural network; X is the input constraint (e.g., a hyperrectangle); Y is the output constraint (e.g., a hyperrectangle); *prob* represents the probability that the specification holds; *Y_dim* is the monotonic dimension to verify; *is_monoinc* represents if verifying monotonic increasing (when it is `True`) or decreasing (when `False`). We omit counterexample generating and sampling here for simplicity.

$\mathcal{P}$ indicating how likely the reachability specification holds. The probabilistic specification is written as:

$$\forall x,\ \ x \in \mathcal{X} \implies Pr(y \in \mathcal{Y} \mid y = f(x)) > \mathcal{P} \qquad (2)$$

OUROBOROS's probabilistic specification verification reduces to proving the following statement: in the network input space, the fraction of the "area" in which the specification holds over the entire space is greater than the threshold $\mathcal{P}$. By default, OUROBOROS assumes the input distribution is uniformly random, when calculating probability. Users can also specify other input distributions by providing user-defined sampling functions; then, OUROBOROS will compute the weighted volume based on the cumulated distribution.

To verify probabilistic specifications, OUROBOROS *iteratively refines* the space that hasn't been verified safe (called unverified space) as Wang et al. (2018c) does. The difference is Wang et al. (2018c) stops when finding a counterexample, while OUROBOROS continues and keeps track of the space verified safe (called verified-safe space) and its volume. OUROBOROS terminates when either the area of verified-safe space is greater than the threshold (an accept) or the area of unverified space is larger than the complementary percentage of the threshold (a reject). Figure 2 demonstrates an example of verifying a probabilistic specification with $\mathcal{P} = 95\%$. Figure 3 depicts the verification algorithm.

Our probabilistic specifications have different guarantees compared to prior work: Berrada et al. (2021) provide probabilistic guarantees under uncertainties, whereas we provide deterministic guarantees by verifying the percentage of the input set that satisfies the property. While our current method works for low-dimensional inputs, it does scale well with higher input dimensionality. To address this issue, we plan to incorporate the probabilistic guarantees in OUROBOROS in the near future.

4.2 Verifying monotonicity

Previous specifications focus on *single* neural network inference—specifying the relationship between one input and its corresponding output. Sometimes systems require properties that among *multiple* inferences, for example, monotonicity. Monotonicity is a widely used correctness property in systems. An example is database cardinality estimation (§6), a procedure to estimate 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 neural network output (that is, the number of returned rows) of the subset query should be smaller than the other's. We define monotonicity specifications as follows.

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

$$\forall x_0, x_1 \in \mathcal{X},\ \ \forall i, x_0[i] \geq x_1[i] \implies f(x_0)[j] \geq f(x_1)[j] \quad (3)$$

where j is an output dimension provided by users.

To verify the monotonicity of a neural network, OUROBOROS computes the lower and upper bounds of the network *gradients* with respect to inputs. We use the same gradient computing method as ReluVal (Wang et al., 2018c). ReluVal uses gradient bounds for heuristic iterative refinement,

while we use gradient bounds to verify monotonicity. Specifically, for a given input space, (i) if the lower bound is greater than 0, then OUROBOROS can safely conclude that the network is monotonically increasing; (ii) if the upper bound is less than 0, then the network is monotonically decreasing; (iii) if the lower bound is less than 0 while the upper bound is greater than 0, OUROBOROS uses *iterative refinement* (the same procedure in verifying probabilistic specifications, §4.1). Namely, OUROBOROS divides the input space into smaller subspaces to reduce over-approximation until the network is monotonic on the input space. Note that iterative refinement will eventually terminate because the number of activation functions in a neural network is finite, meaning the number of non-linear "parts" is finite, hence the number of refinements is finite. Figure 3 describes the algorithm to verify monotonicity specifications.

Our approach differs from prior monotonicity verifications (Liu et al., 2020; Sivaraman et al., 2020; Chen et al., 2021) in two major ways. First, OUROBOROS does not modify networks, whereas some prior work needs certain network architectures. Second, OUROBOROS uses a reachability approach to calculate gradients instead of using SMT solvers or optimization approaches for better scalability.

5 SPEC-AWARE LEARNING

The naive training-verification loop is expensive and inefficient: it trains models blindly without incorporating specifications. To accelerate the training-verification loop, we introduce *spec-aware learning* that co-designs training and verification by using two main techniques:

1. *Spec-aware data:* OUROBOROS generates spec-aware data from specifications and verification results, then adds the data to the next round of training to guide learning.
2. *Early rejection:* early rejection is a shortcut for the networks that fail known hard specifications: OUROBOROS spot-checks these specifications on the networks and (if fail) reject them before verification.

Spec-aware learning overview. Figure 4 depicts spec-aware learning and OUROBOROS's workflow.

A *trainer* trains a neural network. When the trainer finishes training, it sends the network to a *spec-aware checker*. The checker spot-checks if the network fails any hard specifications from prior rounds. If so, the checker early rejects; otherwise, the network is sent to a *verifier*.

The verifier conducts NN-verification. If the network passes the verification, the loop terminates and we get a verified network; if the network fails, the verifier produces counterexamples. Based on the counterexamples, a *spec-aware generator* develops spec-aware data that will be added to the training data of the next round.

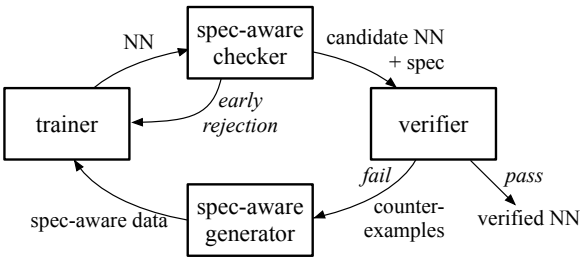

Figure 4: OUROBOROS spec-aware learning overview.

Generating spec-aware data. We observe that the vanilla training-verification loop, doesn't use specifications efficiently for two main reasons: (1) the "coverage" of counterexamples is limited. The verifier will stop when it finds one counterexample, which represents one way that the neural network violates specifications. The network however might fail specifications in many ways—think of a multiple dimensional input space; the network can violate specifications in many directions. (2) sometimes, the counterexamples are "outliers" and it requires multiple rounds to reveal the true specification boundary. In particular, it is beneficial for training to have both positive and negative cases across the specification boundary.

As a response to our two observations, OUROBOROS generates *spec-aware data* using two approaches. First, unlike traditional verifiers that find a counterexample and stop, OUROBOROS customizes the verification algorithm so that it returns multiple counterexamples sampled from different "locations" of the explored space. Second, OUROBOROS further samples around the counterexamples according to the specifications to create high-quality spec-aware data for training. OUROBOROS has a component called spec-aware data generator. It uses the counterexamples and the specifications to create new samples around the specification decision boundary. For example, a learned index may violate specifications by predicting a key (say an integer) far away from its true position. The generator will sample around the failed keys and add them to the training dataset. Similar ideas have been explored in other scenarios (Kong et al., 2018).

A challenge: counterexamples without labels. In some applications, OUROBOROS finds the counterexamples but doesn't know their corresponding true labels, so that these counterexamples cannot be directly added to the training data. As a toy example, if we require that a network f and its input x satisfy that $x < 10 \implies f(x) < 10$. We may find a counterexample: $x = 5$, $f(x) = 20$. However, OUROBOROS does not know how to correct the output 20, that is, what $f(x)$ should be. To address this challenge, we add a *specification aware loss*. Following the above example, OUROBOROS adds a loss $L = max(f(x) - 10, 0)$. With the loss, even if the true labels are missing, OUROBOROS can still optimize the network towards the specification satisfaction direction.

Application	System module and functionality	Specification category and description
Database learned index (Kraska et al., 2018)	*database index*: index is a data structure to accelerate data retrieval which maps database keys to the corresponding data positions on the underlying storage (inputs: database keys; outputs: data positions).	*reachability specification*: for all keys, the output data positions must be within ϵ slots away from their true positions where the ϵ is an allowable error bound. Notice that the database knows the true position for each key (the ground truth is known a priori).
Learning-based memory manager (Maas et al., 2020)	*memory manager*: LLAMA (Maas et al., 2020) uses a neural network to predicts lifetime classes of memory objects by their calling contexts to mitigate heap fragmentation. (inputs: the stack trace of malloc; outputs: predicted memory object lifetime).	*reachability specification*: programmers know the expected behaviors of memory objects, hence can classify objects to different lifetime classes. The specifications specify that the predicted lifetime for memory objects should fall into the expected lifetime class or adjacent classes.
Cardinality estimation (Liu et al., 2015)	*database query optimization*: cardinality estimation predicts the number of rows returned by a query, which will significantly affect the choice of query optimization policies. (inputs: SQL query; outputs: estimated number of returned rows).	*monotonicity specification*: given two SQL statements, one (say `SQL1`) queries a subset of the other (say `SQL2`). The specifications dictate that the outputs of the cardinality estimation for `SQL1` must be smaller or equal to `SQL2`'s output.
Storage latency predictor (Hao et al., 2020)	*OS I/O module*: modern flash storage has complicated internals, hence have poor predictability which harms the overall latency. LinnOS (Hao et al., 2020) uses a neural network to predict the I/O latency. (input: I/O queue status and recent history; output: binary prediction, fast I/O or slow I/O).	*monotonicity specification*: for two I/O requests (say `IOR1` and `IOR2`), if the I/O queue when issuing `IOR1` is shorter than `IOR2` and other parameters of the two requests are the same, the specifications require that `IOR1` should be predicted to be either faster than or in the same category to `IOR2`.
Learned bloom filter (Kraska et al., 2018)	*bloom filter*: a bloom filter is a probabilistic data structure that has been widely used in many systems. Bloom filters test whether an element (for example, a string) is in a pre-defined set. Bloom filters allow false positives—it may say "yes" to a not-in-the-set element. (inputs: an element; outputs: whether the element exists in the set)	*probabilistic specification*: in the case of bloom filters, the set is known ahead of time. The specifications specify that for all elements in the set, the bloom filter should return true in probability X; whereas for non-existing elements, the bloom filter should return false in probability Y. Both X and Y are parameters defined by users.

Figure 5: Applications of neural networks for systems. Note that the descriptions are high-level (see neural network details in §7). The inputs/outputs are described in their original meanings, and neural networks need first encode then use them.

Early rejection. Neural network verification is expensive. Ideally, we only use verification whenever we have to. However, we observe that, often, neural networks go to the verifier with superficial flaws. For example, neural networks haven't been sufficiently trained to learn some spec-aware data from the last round. To address this problem, we introduce *early rejection*. Early rejection allows OUROBOROS to skip expensive verifications by spot-checking several data points that are supposed to be "hard to learn": our current implementation uses counterexamples from prior rounds of verification. Similarly, prior work (Anderson et al., 2019) uses optimization approaches to find "adversarial examples", but this is too expensive in our setup. Our near-future work is to study how to efficiently construct a data set that maximizes the rejecting probability.

An optimization: fine-tuning + early rejection. When networks satisfy almost all specifications, OUROBOROS fine-tunes (Bengio, 2012; Yosinski et al., 2014) them by freezing the beginning multiple layers and only updating weights in the latter layers during training. Fine-tuning and early rejection provide a quick and efficient feedback loop for neural networks to learn those difficult specifications.

6 PUT IT ALL TOGETHER—APPLICATIONS

People have proposed many applications to use neural networks in computer systems, including databases learned index (Kraska et al., 2018), memory allocator (Maas et al., 2020), and OS I/O management (Hao et al., 2020), to name a few. All of these applications use *traditional* neural networks. To evaluate OUROBOROS in practice, we reproduce five applications, design specifications for them, and train verified neural networks to replace the original traditional networks. The five applications are: learned database index (Kraska et al., 2018), learned memory manager (Maas et al., 2020), NN-based cardinality estimation (Liu et al., 2015), LinnOS (Hao et al., 2020) (an SSD latency predictor in OS), and learned bloom filter (Kraska et al., 2018). Figure 5 describes these applications in detail.

Ouroboros component	LOC written/changed
neural network trainer	150 lines of Python
neural network verifier	400 lines of Julia
training-verification loop	800 lines of Python
five applications	700 lines of Python

Figure 6: Components of ouroboros implementation.

Of course, there are many other applications. We choose applications based on three main metrics. First, we pick applications from a diverse set of system areas. For example, learned indexes and cardinality estimation are from databases, memory managers are for programming language runtime, storage latency predictors are used in OS kernel, and bloom filters have been applied in a broad range of systems. Second, we intentionally diversify the applications' specifications to cover all three specification categories. We skip some promising applications because their specifications are similar to others. Finally, we prioritize the applications that haven't been studied by prior neural network verification work (Eliyahu et al., 2021; Dethise et al., 2021).

Note that we simplify some networks compared to the original proposal. We elaborate the modification details in the implementation section (§7). The simplification is because the current ouroboros supports only feedforward neural networks, which is a limitation. It is our near-future work to add more neural network operations and architectures (e.g., recurrent neural networks).

7 Implementation

Implementation overview. Figure 6 lists ouroboros's components. Ouroboros uses `PyTorch v1.8.1` to implement the trainer, and uses the verification toolbox `NeuralVerification.jl` (ver, 2021; Liu et al., 2019) to implement the verifier. We use Python to implement the training-verification loop module which includes the spec-aware checker (conducting early rejection) and the spec-aware generator (generating spec-aware data).

Verifier. We use Neurify (Wang et al., 2018b) as our base verification algorithm and extend it with (1) returning multiple counterexamples in each verification, (2) incremental specification, and (3) memorizing verification history for each specification. The verifier needs to keep track of the status of specifications because the order of verifying specifications matters: ouroboros's verifier prioritizes failed specifications, instead of blindly verifying all the specifications in the given order. This is because previously verified safe specifications are likely to be safe. The verifier revisits the verified safe specifications when the network passes other specifications.

Computing gradients. People have intensively studied how to calculate bounds for the gradients (Laurel et al., 2022; Jordan & Dimakis, 2020; Zhang et al., 2019b). In our implementation, we compute the symbolic gradient bound leveraging the symbolic representation of Neurify (Wang et al., 2018b). The symbolic gradient bound is an over-approximation and can be refined when the input set gets split.

Apps	Network configurations
LearnedIndex	FC: 1-1000-1 (stage 1) FC: 1-100-1 (stage 2)
MemManager	FC: 13-300-300-4
CardEsti	FC: 4-500-1
LatPredictor	FC: 9-300-300-1
BloomFilter	FC: 2-50-50-1

Figure 7: Network architectures. "FC" means fully-connected feed-forward neural networks. The number after "FC" indicates the number of neurons in each layer. Layers are separated by dashes. LearnedIndex has two stages of networks, as proposed by the original paper (Kraska et al., 2018).

Applications. We also rewrite the five applications in Figure 5 using Python and PyTorch. We have reimplemented all five applications (Figure 5) using feed-forward neural networks. The network configurations are listed in Figure 7.

Specifications. Figure 5 describes the high-level specifications. For LearnedIndex, we use an error bound $\epsilon = 1000$. MemManager produces outputs indicating objects' lifetimes, which are categorized into five different groups from shortest to longest. Our specification mandates that predictions should not differ by more than one category. With regard to CardEsti, we require SQL queries for two columns (the `page_len` and `page_latest` in table `page` of the Wikipedia; see §8) to be monotonic. As for LatPredictor, we require the predicted IO latency (the output) to be monotonic with respect to the IO queue length (one dimension of the input). Finally, for BloomFilter, we require the false positive rate to be $\leq 2.5\%$.

8 Experimental evaluation

In this section, we answer three questions:

- How long does it take ouroboros to train a verified neural network, and how does that compare to baselines?
- Compared with traditional neural networks, how do verified neural networks perform?
- How does the performance of verified neural networks compare to that of traditional data structures?

Benchmarks and datasets. We use the five applications in Figure 5 as benchmarks in our experiments. Their networks are specified in Figure 7. Here are their training datasets:

- *Learned index (LearnedIndex)*: we use the synthetic dataset used by the Kraska et al. (2018): we sample 120K integer keys from a lognormal distribution.
- *Learning-based memory manager (MemManager)*: we

Apps	normal	vanilla	OUROBOROS	speedup
LearnedIndex	34	47	47	1.0×
MemManager	5	175	25	7.0×
CardEsti	19	114	43	2.7×
LatPredictor	3	90	47	1.9×
BloomFilter	1	TO	474	>1.3×

Figure 8: Running time (in seconds) for training traditional and verified neural networks. The "speedup" column indicates OUROBOROS's speedup versus the vanilla loop. "TO" means time out (600sec).

instrument a memory allocator which records the stack trace of each `malloc` and tracks the life time of each allocated memory object. We then run a key-value store Redis (red, 2021) with this memory allocator to collect its memory traces. Our final data has 60K allocation entries.

- *Cardinality estimation (CardEsti)*: we download the Wikipedia database and rebuild one of its tables, `page`. We further generate a set of SQL statements that queries various ranges of columns `page_latest` and `page_len` and their number of returned rows. This dataset collects 100K queries and their results.
- Storage latency predictor (*LatPredictor*): we use LinnOS (lin, 2021) SSD latency collector to collect 60K SSD I/O requests on an AWS `t3.xlarge` machine.
- *Learned bloom filter (BloomFilter)*: we use a dataset, Crimes in Boston (bos, 2021), from Kaggle. We use the learned bloom filter to tell if a location (latitude and longitude) had a crime. There are 300K crime locations in this dataset.

Setup. For the experiments below, we run OUROBOROS in a machine with an AMD EPYC 7773X 64-Core Processor, 256 GB RAM, and a NVIDIA RTX A5000 GPU. The machine uses Ubuntu 20.04. The software we use is PyTorch v1.8.1 and Julia v1.8.5.

8.1 Training verified neural networks with OUROBOROS

Baselines. In this section, we consider training networks with two baselines:

- *Normal training*: running traditional training without verification. The outputs will be traditional neural networks that do not fully comply with specifications.
- *Vanilla loop*: the training-verification loop proposed by NeVer (Pulina & Tacchella, 2011) which adds counterexamples to the next round of training.

Network training time. We run OUROBOROS and two baselines on all five applications, and record their end-to-end training time from starting the training job to getting the final network. Figure 8 shows the results.

In this experiment, normal training is faster than the vanilla loop and OUROBOROS because it doesn't perform verifications at all. OUROBOROS runs faster than the vanilla loop in most cases because of spec-aware learning (§5). The only exception is LearnedIndex, for which OUROBOROS has no improvement. This is because the specifications are easy to satisfy and the networks are verified safe before spec-aware learning takes effect (see also ablation study later, Figure 10).

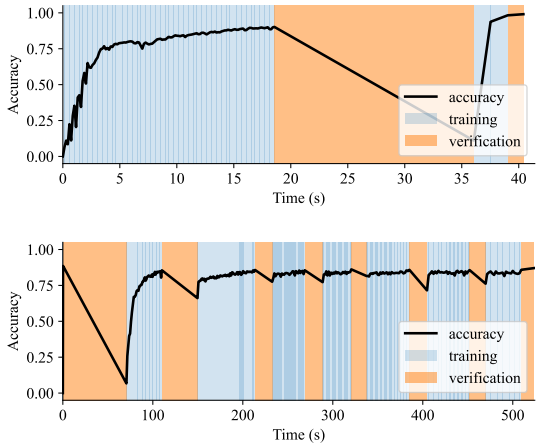

Figure 9: The training and verification switches and the model accuracy changes for CardEsti (top) and BloomFilter (bottom).

Decomposition of OUROBOROS's execution. To understand how training and verification interact, we break OUROBOROS's execution time into training and verification phases. We also record the model accuracy changes as OUROBOROS switches between training and verification. Figure 9 shows the results for CardEsti and BloomFilter.

For CardEsti, there are two training phases and two verification phases. The neural network knows nothing in the beginning, hence it requires a long time to learn from training data. The first verification is long because the neural network is not robust and therefore hard to verify. In the second phase, with the help of spec-aware data, the model quickly learns the desired specification and passed the verification.

The pattern of BloomFilter is different. The neural network quickly learns the training data. However, the initial network doesn't capture the logic of specifications. The specifications require a precies boundary produced by the neural network, which is difficult to learn but easy to verify. Therefore, BloomFilter spends most of the time on training.

Ablation study. To understand how the two techniques—spec-aware data and early rejection—- contribute to OUROBOROS's performance, we conduct an ablation study. We evaluate all five applications with four variants: (1) normal training (without verification), (2) vanilla loop, (3) vanilla loop with spec-aware data, and (4) vanilla loop with spec-aware data and early rejection (OUROBOROS). Figure 10 shows the results.

From Figure 10, we see that techniques contribute differently

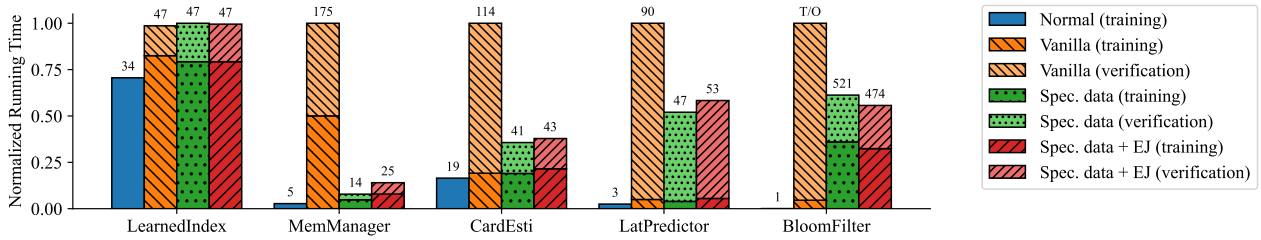

Figure 10: An ablation study of how OUROBOROS's techniques contirbute to perfromance. "Normal" represents normal training (without verification); "Vanilla" reprsents the vanilla loop baseline; "Spec. data" represents spec-aware data; "E.J." represents early rejection. All bars have two parts: the bottom part represents the training time, and the top part indicates verification time.

App	Network	Test dataset	Spec dataset
MemManager	classic net	86.0%	23.2%
	verified net	87.6%	100%
CardEsti	classic net	88.9%	1.6%
	verified net	88.0%	100%
LatPredictor	classic net	96.1%	1.9%
	verified net	97.7%	100%
BloomFilter	classic net	99.9%	4.7%
	verified net	99.5%	78.6%

Figure 11: Model accurarcy for verified and classic networks.

to different applications. For LearnedIndex, no techniques improve training because the specifications are relatively easy to satisfy. We looked into the training and found that the stage1 model (the largest network of LearnedIndex) satisfies its specifications after the first round of training, and OUROBOROS techniques are not involved at all. So, training performance is almost the same for all variants. For MemManager and CardEsti, spec-aware data plays a major role in accelerating learning. For early rejection, it doesn't help much in the first four applications because verifying reachability specifications and monotonicity specifications are relatively fast. But, early rejection contributes meaningfully for the BloomFilter given verifying probabilistic specifications is challenging and computationally expensive.

8.2 Verified neural networks vs. traditional networks

To understand if OUROBOROS sacrifices model accuracy for safety, we compare both the verified neural networks and traditional neural networks for four applications. (We exclude the LearnedIndex because Kraska et al. (2018)'s original design requires to replace the neural networks that fail specifications with B-trees, which is different from other applications in this experiment.) There are two categories of test sets. One is generated using the traditional approach that we divide the data collected from the four applications into 80:20, and use the 20% data as normal test sets. The other test set comes from specifications. We collect the counterexamples generated by the verification phases and use them as specification test sets. We evaluate the model accuracy of certified and traditional networks on the two test sets. Figure 11 shows the results.

We see that verified neural networks have comparable performance for normal test sets, and outperform traditional networks on specification test sets. Note that the spec dataset accuracy of BloomFilter is not 100% because this application uses probabilistic specifications. Moreover, the accuracy in the spec dataset is pessimistic because the spec dataset contains counterexamples that sit between decision boundaries, which are hard to predict right.

8.3 Execution performance of verified neural networks

In this section, we answer the question how classic and verified neural networks work in systems compared with existing data structures. We experiment with three applications: LearnedIndex, BloomFilter, and LatPredictor.

Baselines and setup. For LearnedIndex, we choose RMI (Kraska et al., 2018) and B-Tree as baselines. We use the original RMI implementation in C++ from RMI authors (sos, 2023), and a C++ implementation of B-Trees (btr, 2023). The dataset is from a standard learned index benchmark, SOSD (Marcus et al., 2020a); we downsampled its `face` dataset to 150K. For BloomFilter, our baseline is the scalable bloom filter (Almeida et al., 2007), the highest-stared bloom filter implementation on GitHub. The dataset is the Crimes in Boston (bos, 2021) from Kaggle. For LatPredictor, we port the original LinnOS (Hao et al., 2020) from TensorFlow to PyTorch and generate two variants: `LinnOS-binary` is the original model that does binary classification (fast IO or slow IO). We simply LinnOS' inputs to nine dimensions (was 31) to align with the training data.

Metrics. For all models, we measure the throughputs (requests per second), latency (serving one single request), and their memory consumptions (size in KB). For throughputs, all neural networks—including ours and baselines that use neural networks—use batching and run on GPUs; others run on CPUs without batching. For latencies, all models serve a single request with no batching. Neural networks run on GPUs; others run on CPUs. For LearnedIndex, we

application	model	throughput	latency (μs or ns)	size (KB)	avg/max err	FP%	accuracy
LearnedIndex	verified NN4Sys	9.8M	351 μs	47	245/906	–	–
	classic NN4Sys	9.8M	377 μs	47	239/905	–	–
	RMI (49KB)	24.5M	41 ns	49	0.7/8	–	–
	RMI (6KB)	20.4M	49 ns	6	1.2/20	–	–
	B-Tree	5.1M	195 ns	337	–	–	–
BloomFilter	verified NN4Sys	147.1K	85 μs	1965	–	2.2%	–
	classic NN4Sys	148.1K	84 μs	1965	–	56.2%	–
	Bloom Filter (1%)	333.0K	3 μs	382	–	1.0%	–
	Bloom Filter (2%)	345.5K	3 μs	325	–	2.0%	–
LatPredictor	verified NN4Sys	172.9K	77 μs	366	–	–	97.7%
	classic NN4Sys	173.2K	81 μs	366	–	–	96.3%
	LinnOS-binary	172.5K	76 μs	12	–	–	96.1%

Figure 12: Execution performance of baselines and verified neural networks. In throughputs, "M" is used to denote a million, and "K" for a thousand. "avg/max err" represent the average and max error that learned indexes predict. "FP%" means false positive rates for bloom filters. "accuracy" indicates the accuracy of IO latency predictions.

measure the average and max errors which represent the inaccuracy of model's prediction (we do not include the local search procedure for RMIs and our models). For BloomFilter, we show the false positive rates for baselines and our models. For LatPredictor, we show the accuracy results: for `LinnOS-binary`, this is the accuracy of binary classification; for our model, this is the accuracy of predicting IO latency.

Results. Figure 12 depicts the results. For LearnedIndex, there is no significant difference between verified NN4Sys and unverified NN4Sys for the same reason we mentioned in 8.1. RMIs have 2.5× higher throughput than verified NN4Sys. Latency-wise, RMIs and B-Trees are much faster; this is due to GPU-CPU data transfer and unoptimized NN4Sys: the latency of NN4Sys can be improved by model quantization, pruning, and model compilation. A deeper reason why RMIs work well is that this dataset is a one-dimension to one-dimension mapping, and linear models work really well on 1D-inputs. We expect neural networks work better in terms of model accuracy for high-dimensional inputs (like strings). For BloomFilter, verified NN4Sys has comparable performance with traditional bloom filters in throughputs, with the similar false positive rate, which is much lower compared with the unverified NN4Sys. We expect to have better performance when using inference-oriented hardware accelerators (like neural processor units). For LatPredictor, unverified NN4Sys and verified NN4Sys both have better prediction accuracy compared with `LinnOS-binary` because `LinnOS-binary` is tiny for fast inference (the original implementation complies the TensorFlow model into CPU-friendly implementation; we do not port this process.) Because the verified neural networks have been trained to be monotonic, which reflects the common sense, it performs better than unverified NN4Sys.

9 Related work

Training-verification loop. The "loop" idea is a reminiscence of the feedback loop in control theory (Doyle et al., 2013). NeVer (Pulina & Tacchella, 2011) first introduces the idea of training-verification loop in the context of neural network verification. Later, other systems (Dvijotham et al., 2018; Tan et al., 2021; Yang et al., 2021) explore similar ideas to construct neural networks with provable guarantees. Ouroboros is the first system designed for training verified neural networks for systems, with supports to probabilistic specifications and monotonicity specifications.

Neural network verification for systems. There are many attempts to replace system components with neural networks (Kraska et al., 2018; Maas et al., 2020; Dai & Shrivastava, 2019; Byun & Lim, 2021; Liu et al., 2015; Wang et al., 2020; Hao et al., 2020). One challenge they face is that neural networks are black boxes and do not strictly follow safety properties. Several tools (Kazak et al., 2019; Eliyahu et al., 2021; Dethise et al., 2021) address this challenge by using neural network verification to check whether the networks used in computer systems are safe. Compare with these tools, ouroboros uses the same techniques but for a different purpose. Ouroboros aims at constructing networks that satisfy specifications, instead of only checking if networks satisfy the specifications.

10 Conclusion

Verified neural networks are networks that are proven to comply with specifications. They are useful for scenarios that have strong safety requirements, for example, critical computer systems. Ouroboros is a system that enables users to train verified neural networks. With ouroboros, people can finally *trust* their neural networks.

Acknowledgement

We thank our shepherd Junfeng Yang and the anonymous reviewers of MLSys23 for their feedback which substantially improved this paper. This work was supported by NSF CAREER Awards #2237295, #2144489, and #2239351.

References

Crimes in Boston. https://www.kaggle.com/AnalyzeBoston/crimes-in-boston, 2021.

LinnOS Artifact. https://www.chameleoncloud.org/experiment/share/15?s=409ab137f20e4cd38ae3dd4e0d4bfa7c, 2021.

Redis: The Real-time Data Platform. https://redis.com/, 2021.

Neuralverification.jl. https://github.com/sisl/NeuralVerification.jl, 2021.

STX B+ Tree C++ Template Classes. https://panthema.net/2007/stx-btree/, 2023.

Search on Sorted Data Benchmark. https://github.com/learnedsystems/SOSD, 2023.

Almeida, P. S., Baquero, C., Preguiça, N., and Hutchison, D. Scalable bloom filters. *Information Processing Letters*, 101(6):255–261, 2007.

Anderson, G., Pailoor, S., Dillig, I., and Chaudhuri, S. Optimization and abstraction: a synergistic approach for analyzing neural network robustness. In *Proceedings of the 40th ACM SIGPLAN conference on programming language design and implementation*, pp. 731–744, 2019.

Bengio, Y. Deep learning of representations for unsupervised and transfer learning. In *Proceedings of ICML workshop on unsupervised and transfer learning*, pp. 17–36. JMLR Workshop and Conference Proceedings, 2012.

Berrada, L., Dathathri, S., Dvijotham, K., Stanforth, R., Bunel, R. R., Uesato, J., Gowal, S., and Kumar, M. P. Make sure you're unsure: A framework for verifying probabilistic specifications. *Advances in Neural Information Processing Systems*, 2021.

Bertsimas, D., Brown, D. B., and Caramanis, C. Theory and applications of robust optimization. *SIAM review*, 53(3): 464–501, 2011.

Byun, H. and Lim, H. Learned fbf: Learning-based functional bloom filter for key-value storage. *IEEE Transactions on Computers*, 2021.

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

Chen, Y., Wang, S., Qin, Y., Liao, X., Jana, S., and Wagner, D. Learning security classifiers with verified global robustness properties. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pp. 477–494, 2021.

Dai, Z. and Shrivastava, A. Adaptive learned bloom filter (ada-bf): Efficient utilization of the classifier. *arXiv preprint arXiv:1910.09131*, 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.

Doyle, J. C., Francis, B. A., and Tannenbaum, A. R. *Feedback control theory*. Courier Corporation, 2013.

Dreossi, T., Ghosh, S., Yue, X., Keutzer, K., Sangiovanni-Vincentelli, A., and Seshia, S. A. Counterexample-guided data augmentation. *arXiv preprint arXiv:1805.06962*, 2018.

Dvijotham, K., Gowal, S., Stanforth, R., Arandjelovic, R., O'Donoghue, B., Uesato, J., and Kohli, P. Training verified learners with learned verifiers. *arXiv preprint arXiv:1805.10265*, 2018.

Ehlers, R. Formal verification of piece-wise linear feed-forward neural networks. In *International Symposium on Automated Technology for Verification and Analysis*, pp. 269–286. Springer, 2017.

Eliyahu, T., Kazak, Y., Katz, G., and Schapira, M. Verifying learning-augmented systems. 2021.

Fan, J. and Li, W. Adversarial training and provable robustness: A tale of two objectives. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pp. 7367–7376, 2021.

Ferragina, P. and Vinciguerra, G. The pgm-index: a fully-dynamic compressed learned index with provable worst-case bounds. *Proceedings of the VLDB Endowment*, 13 (8):1162–1175, 2020.

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.

Gowal, S., Dvijotham, K., Stanforth, R., Bunel, R., Qin, C., Uesato, J., Arandjelovic, R., Mann, T., and Kohli,

P. On the effectiveness of interval bound propagation for training verifiably robust models. *arXiv preprint arXiv:1810.12715*, 2018.

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. 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.

Jordan, M. and Dimakis, A. G. Exactly computing the local lipschitz constant of relu networks. *Advances in Neural Information Processing Systems*, 33:7344–7353, 2020.

Katz, G. Augmenting deep neural networks with scenario-based guard rules. In *International Conference on Model-Driven Engineering and Software Development*. Springer, 2020.

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

Kong, S., Solar-Lezama, A., and Gao, S. Delta-decision procedures for exists-forall problems over the reals. In *Computer Aided Verification: 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II 30*, 2018.

Kraska, T., Beutel, A., Chi, E. H., Dean, J., and Polyzotis, N. The case for learned index structures. 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.

Laurel, J., Yang, R., Singh, G., and Misailovic, S. A dual number abstraction for static analysis of clarke jacobians. *Proceedings of the ACM on Programming Languages*, 6 (POPL):1–30, 2022.

Liang, E., Zhu, H., Jin, X., and Stoica, I. Neural packet classification. 2019.

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

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.

Liu, X., Han, X., Zhang, N., and Liu, Q. Certified monotonic neural networks. *Advances in Neural Information Processing Systems*, 33:15427–15438, 2020.

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

Maas, M., Andersen, D. G., Isard, M., Javanmard, M. M., McKinley, K. S., and Raffel, C. Learning-based memory allocation for c++ server workloads. 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. pp. 197–210, 2017.

Mao, H., Schwarzkopf, M., Venkatakrishnan, S. B., Meng, Z., and Alizadeh, M. Learning scheduling algorithms for data processing clusters. 2019.

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.

Marcus, R., Zhang, E., and Kraska, T. Cdfshop: Exploring and optimizing learned index structures. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pp. 2789–2792, 2020c.

Müller, M. N., Eckert, F., Fischer, M., and Vechev, M. Certified training: Small boxes are all you need. *arXiv preprint arXiv:2210.04871*, 2022.

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

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

Raghunathan, A., Steinhardt, J., and Liang, P. Semidefinite relaxations for certifying robustness to adversarial examples. *arXiv preprint arXiv:1811.01057*, 2018.

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.

Shi, Z., Wang, Y., Zhang, H., Yi, J., and Hsieh, C.-J. Fast certified robust training with short warmup. *Advances in Neural Information Processing Systems*, 34:18335–18349, 2021.

Sivaraman, A., Farnadi, G., Millstein, T., and Van den Broeck, G. Counterexample-guided learning of monotonic neural networks. *Advances in Neural Information Processing Systems*, 33:11936–11948, 2020.

Solar-Lezama, A. Program sketching. *International Journal on Software Tools for Technology Transfer*, 15:475–495, 2013.

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.

Tjeng, V., Xiao, K., and Tedrake, R. Evaluating robustness of neural networks with mixed integer programming. *arXiv preprint arXiv:1711.07356*, 2017.

Tran, H.-D., Yang, X., Lopez, D. M., Musau, P., Nguyen, L. V., Xiang, W., Bak, S., and Johnson, T. T. Nnv: The neural network verification tool for deep neural networks and learning-enabled cyber-physical systems. In *International Conference on Computer Aided Verification*, pp. 3–17. Springer, 2020.

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

Wang, S., Pei, K., Whitehouse, J., Yang, J., and Jana, S. Efficient formal safety analysis of neural networks. *arXiv preprint arXiv:1809.08098*, 2018b.

Wang, S., Pei, K., Whitehouse, J., Yang, J., and Jana, S. Formal security analysis of neural networks using symbolic intervals. 2018c.

Wang, S., Pei, K., Whitehouse, J., Yang, J., and Jana, S. Efficient formal safety analysis of neural networks. *arXiv preprint arXiv:1809.08098*, 2018d.

Wang, X., Qu, C., Wu, W., Wang, J., and Zhou, Q. Are we ready for learned cardinality estimation? *arXiv preprint arXiv:2012.06743*, 2020.

Weng, L., Chen, P.-Y., Nguyen, L., Squillante, M., Boopathy, A., Oseledets, I., and Daniel, L. Proven: Verifying robustness of neural networks with a probabilistic approach. In *International Conference on Machine Learning*, pp. 6727–6736. PMLR, 2019.

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.

Yang, X., Yamaguchi, T., Tran, H.-D., Hoxha, B., Johnson, T. T., and Prokhorov, D. Neural network repair with reachability analysis. *arXiv preprint arXiv:2108.04214*, 2021.

Yosinski, J., Clune, J., Bengio, Y., and Lipson, H. How transferable are features in deep neural networks? *arXiv preprint arXiv:1411.1792*, 2014.

Zhang, H., Chen, H., Xiao, C., Gowal, S., Stanforth, R., Li, B., Boning, D., and Hsieh, C.-J. Towards stable and efficient training of verifiably robust neural networks. *arXiv preprint arXiv:1906.06316*, 2019a.

Zhang, H., Zhang, P., and Hsieh, C.-J. Recurjac: An efficient recursive algorithm for bounding jacobian matrix of neural networks and its applications. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pp. 5757–5764, 2019b.

Zhang, Y. and Huang, Y. "Learned" Operating Systems. *ACM SIGOPS Operating Systems Review*, 2019.

A Artifact Appendix

A.1 Abstract

Ouroboros is a system to train verified neural networks for computer systems. Ouroboros have several parts: The training part trains models given the normal data and counterexample data. The verification part verifies models with given specifications and return counterexamples, which contains a modified version of NeuralVerification.jl package. The training verification loop part connects training and verification, and guides the selection of counterexamples and counterexample-guided training.

A.2 Artifact check-list (meta-information)

- Algorithm: Neural Network Verification
- Run-time environment: Ubuntu 20.04

- Experiment hardware: AMD EPYC 7773X 64-Core Processor, NVIDIA GA102GL [RTX A5000]
- Software dependencies: Python3.7+, Julia1.8.5, pytorch
- disk space required: 5 GB
- setup time: 30 mins
- experiment running time: 1-5 hours
- Code/Data licenses: MIT license
- DOI: 10.5281/zenodo.7788500

A.3 Description

A.3.1 How delivered

OUROBOROS's code and experiment datasets can be found at https://github.com/Khoury-srg/Ouroboros.

A.3.2 Hardware dependencies

OUROBOROS does not require specific hardware to run. But with GPUs, the performance of OUROBOROS's training phase will be accelerated.

A.3.3 Software dependencies

This program relies on the following software: julia1.8.5, python3.7+, pytorch.

A.3.4 Data sets

The datasets are contained in our repository.

A.4 Installation

Install miniconda and julia. Git clone the repository.

```
#!/bin/bash
wget https://repo.anaconda.com/miniconda/Miniconda3-
    ↪ py37_23.1.0-1-Linux-x86_64.sh
bash Miniconda3-py37_23.1.0-1-Linux-x86_64.sh
wget https://julialang-s3.julialang.org/bin/linux/
    ↪ x64/1.8/julia-1.8.5-linux-x86_64.tar.gz
tar zxvf julia-1.8.5-linux-x86_64.tar.gz
git clone git@github.com:Khoury-srg/Ouroboros.git
```

Add the following line to ~/.bashrc.

```
export PATH="$PATH:/path/to/<Julia_directory>/bin"
```

Configure julia and install NeuralVerification from our repo.

```
source ~/.bashrc
julia
```

Inside julia

```
using Pkg
Pkg.develop(path="/path/to/ouroboros/
    ↪ NeuralVerification.jl")
Pkg.add("LazySets")
# Exit by CTRL+D
```

Setup python environment

```
conda create --name ouroboros python==3.7
conda activate ouroboros
pip3 install torch torchvision torchaudio pandas
    ↪ onnx onnxruntime matplotlib annoy julia
python
```

Inside python

```
import julia
julia.install()
```

A.5 Experiment workflow

The following command runs all experiments.

```
cd /path/to/ouroboros/src
python run_exp.py
```

To draw all figures and generate the table in our paper:

```
cd /path/to/ouroboros/src
python plot_figures.py
python generate_table.py
```

A.6 Evaluation and expected result

All the models and outputs are stored in /ouroboros/results. Deliverable figures are stored in /ouroboros/imgs. Numerical results for the table are printed in the terminal.

For figure 9, we expect the reproduced accuracy (y-axis) to remain similar, but the time (x-axis) may be different.

For figure 10, we expect the reproduced results to have the same relative performance as in the figure. But the concrete running time may be different.

For table 11, we expect all the numbers to remain similar.