Verifying Learning-Augmented Systems

Tomer Eliyahu
tomer.eliyahu2@mail.huji.ac.il
The Hebrew University of Jerusalem
Jerusalem, Israel

Yafim Kazak
Yafim.Kazak@gmail.com
The Hebrew University of Jerusalem
Jerusalem, Israel

Guy Katz
guykatz@cs.huji.ac.il
The Hebrew University of Jerusalem
Jerusalem, Israel

Michael Schapira
schapiram@cs.huji.ac.il
The Hebrew University of Jerusalem
Jerusalem, Israel

ABSTRACT

The application of deep reinforcement learning (DRL) to computer and networked systems has recently gained significant popularity. However, the obscurity of decisions by DRL policies renders it hard to ascertain that learning-augmented systems are safe to deploy, posing a significant obstacle to their real-world adoption. We observe that specific characteristics of recent applications of DRL to systems contexts give rise to an exciting opportunity: applying formal verification to establish that a given system *provably* satisfies designer/user-specified requirements, or to expose concrete counter-examples. We present *whiRL*, a platform for verifying DRL policies for systems, which combines recent advances in the verification of deep neural networks with scalable model checking techniques. To exemplify its usefulness, we employ whiRL to verify natural requirements from recently introduced learning-augmented systems for three real-world environments: Internet congestion control, adaptive video streaming, and job scheduling in compute clusters. Our evaluation shows that whiRL is capable of guaranteeing that natural requirements from these systems are satisfied, and of exposing specific scenarios in which other basic requirements are not.

CCS CONCEPTS

• **Networks → Protocol testing and verification**; • **Software and its engineering → Formal software verification**.

KEYWORDS

deep reinforcement learning, deep learning, neural networks, formal verification, networked systems, congestion control, adaptive bitrate algorithms, resource scheduling

ACM Reference Format:
Tomer Eliyahu, Yafim Kazak, Guy Katz, and Michael Schapira. 2021. Verifying Learning-Augmented Systems. In *ACM SIGCOMM 2021 Conference (SIGCOMM '21), August 23–27, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3452296.3472936

1 INTRODUCTION

Recent advances in reinforcement learning (RL) [74], namely deep RL (DRL) [68, 70], have given rise to a surge of interest in applying DRL to sequential decision making in many areas, including a broad variety of practical computer and networked systems contexts. Examples include compilers [30], databases [42], compute resource scheduling [55], congestion control on the Internet [1, 35], video streaming [57], and many more. See [31, 56] and references therein for an overview and more examples. DRL suggests a compelling alternative for the traditional handcrafting of such systems by domain-specific experts.

However, the (typical) opacity of decision making by deep neural networks (DNNs) renders it difficult to determine whether a DRL policy satisfies even basic correctness and/or security requirements. This constitutes a major obstacle to the actual deployment of DRL policies for real-world systems. While testing policies in simulated, emulated, or live environments can expose performance/security flaws, it cannot establish their absence (even in scenarios that only slightly differ from the training/validation sets [75]).

We aim to facilitate the safe deployment of DRL-based systems by providing the means to *guarantee* that a DRL policy meets a designer/user-specified requirement, or, alternatively, for identifying a concrete scenario in which the requirement is violated. Counter-examples for satisfying a certain requirement can be utilized in various manners: (1) they can serve as an indication that the DRL policy is not "sufficiently trained" (similarly to acceptance tests for traditional software), suggesting that the training process should be prolonged or that more data should be added to the training set; (2) in some cases, the generated counter-examples themselves can be injected into the training data to enhance the system's robustness, i.e., be used for adversarial training [52, 84]; and (3) the existence of counter-examples can aid in identifying circumstances in which the DRL policy should be manually overridden. To accomplish our goal of provably determining whether a specified requirement from a DRL policy holds or not, we build on recent advancements in *formal verification of DNNs*.

DNN verification enables establishing that the mapping from inputs to outputs by a DNN provably satisfies certain properties. Verification provides far stronger guarantees than testing, because they potentially extend to *any* input that the DNN might encounter—even if there are infinitely many possible inputs. Yet, despite great strides in recent years, state-of-the-art verification approaches (e.g., [23, 33, 37, 71, 80, 82]) face severe scalability issues. Specifically, such approaches typically (1) scale only to moderate-size neural networks (tens of thousands of neurons, at most); (2) consider a single invocation of the DNN (whereas verification of DRL policies involves reasoning about *sequences* of DNN invocations); and (3) focus on proving *local* properties (e.g., adversarial robustness properties [75]), in which the considered space of possible inputs is highly restricted [23, 33, 37, 80, 82]. This renders formal DNN verification inapplicable to many domains and properties of interest.

Recently, DRL has been increasingly applied to computer and networked systems. We observe that the specific characteristics of these systems give rise to exciting opportunities for formally verifying *non-local* properties of *sequential* decision making. Specifically, the input to the DNN in many recently proposed DRL-based systems consists of a fairly few features, which were manually handcrafted by domain experts (as opposed, to, e.g., computer vision applications, where the extraction of meaningful features from vast amounts of raw data is left to the DNN). As we later explain, the low dimensionality and inherent structure of the DNN input space have important implications for formal verification; specifically, these traits facilitate the use of DNNs whose sizes are within reach of existing DNN verification technologies, accommodate the formulation of expressive and complex desired properties, and simplify reasoning about how an invocation of the DNN affects the DNN's future inputs.

We present a framework for verifying DRL policies for systems. Our framework combines formal verification of DNNs [37] with methodologies for bounded model checking, where the latter ingredient enables scalable verification. We implement our approach in a new tool suite, called whiRL, and demonstrate its usefulness by evaluating it on three learning-augmented computer systems—for Internet congestion control [35], for adaptive video streaming [57], and for scheduling jobs in compute clusters [55]. We formulate natural requirements from each of these systems, and use whiRL to determine whether these are *always* satisfied or not. Our evaluation results expose several problems in the target systems, suggesting that formal verification can play an important role in the design and deployment of safe DRL policies for computer systems.

To facilitate the reproducibility of our results and to support follow-up research, our code and experiments are available online [20].

This paper is organized as follows. We provide necessary background on DRL and verification in Section 2. We discuss the challenges and opportunities in DRL verification in Section 3, and present our verification technique in Section 4. Our evaluation results for whiRL on the three considered case studies appear in Section 5. We discuss lessons for DRL system design and directions for future research in Section 6, present related work in Section 7, and conclude in Section 8.

This work does not raise any ethical issues.

2 BACKGROUND

We provide below necessary background on learning with deep neural networks, DRL and its applications to systems, and DNN verification.

Deep Neural Networks (DNNs). A DNN is a weighted directed graph, where the nodes (also referred to as neurons) are organized in layers. The first layer is called the *input layer*, the final layer is called the *output layer*, and the remaining, intermediate layers are called *hidden layers*. In feedforward neural networks, which are prevalent in DRL policies for computer and networked systems, outgoing edges from neurons in each layer i only lead to neurons in layers higher than i. We henceforth restrict our attention to feedforward networks. We briefly discuss the possible extension of our technique to recurrent neural networks (RNNs) in Section 4.4.

A DNN represents a function mapping its inputs to its outputs; the output for initial values assigned to its input neurons is determined by propagating these values through the DNN, layer by layer, until the values of the output layer are computed. When hidden layer i is traversed, each neuron performs the following two steps. First, it computes a *weighted sum* of the values of neurons connected to it from previous layers, according to the edge weight associated with each neuron. Next, an *activation function* is applied to the computed weighted sum to determine the value of the neuron. A very popular activation function is the piecewise linear ReLU function [27], defined as $\text{ReLU}(x) = \max(0, x)$. The output layer is typically simply a weighted sum of its preceding layer, without an activation function.

An illustration appears in Fig. 1. This network has an input layer, two hidden (ReLU) layers, and a final output layer. For input values $\langle v_1^1, v_1^2 \rangle = \langle 1, 1 \rangle$, the values of the first layer's neurons are given as:

$$v_2^1 = \text{ReLU}(v_1^1 + 2v_1^2 + 1) = \text{ReLU}(1 + 2 + 1) = 4$$
$$v_2^2 = \text{ReLU}(-5v_1^1 + v_1^2 + 2) = \text{ReLU}(-5 + 1 + 2) = 0$$

Similarly, we get that $\langle v_3^1, v_3^2 \rangle = \langle 0, 9 \rangle$, and so $v_4^1 = -18$.

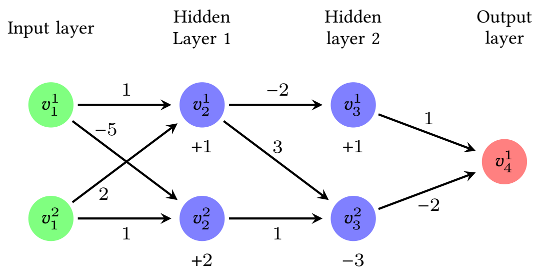

Figure 1: A toy DNN.

The edge weights of a DNN are determined in the process of training the DNN, which involves continuously updating these weights to reduce the error of the DNN's output on the training dataset. The training process goes on until either a "sufficiently good" mapping from inputs to outputs is learned, or until the learning process converges and stops. See [27] for additional details.

DRL and learning-augmented systems. In RL [74], an *agent* observes, at each discrete time step $t \in 0, 1, ...,$ a *state* of its *environment* s_t, and selects an action a_t. After selecting action a_t, the agent observes a *reward* r_t. The agent's goal is to learn a policy π, i.e., a mapping from states to actions, which maximizes the *expected cumulative discounted return* $R_t = \mathbb{E}\big[\sum_t \gamma^t \cdot r_t\big]$, for $\gamma \in \big[0, 1\big)$. The parameter γ is termed the *discount factor.* In DRL, DNNs are employed to approximate the optimal π [68, 70]. Recently, DRL has been applied to many computer and networked systems, ranging from compilers and databases to Internet routing and congestion control [31, 56].

To motivate our approach and methodology, we will use DRL-based Internet congestion control [1, 35] as a running example. Recently, DRL policies have been proposed as an alternative for today's handcrafted congestion control algorithms [1, 35]. Such policies map local observables about the history of traffic conditions (e.g., achieved throughput, packet loss rates, experienced latency) to the next choice of sending rate. See [1, 35] for a thorough exposition.

DNN verification and its limitations. Following the rise in popularity of DNNs, the verification community has begun addressing the need for verifying and reasoning about them (e.g., [12, 23, 26, 28, 33, 37–40, 45, 45, 47, 71, 80, 82]). A DNN verification query comprises (i) a neural network N, (ii) an *input property* P, and (iii) an *output property* Q. A verification algorithm answers the question "*does there exist an input vector x, such that $P(x)$ holds and $Q(N(x))$ also holds?*", where $N(x)$ is the output vector that the neural network produces for input x, and Q typically expresses the *negation* of the desired property. Verification has two possible outcomes: (1) UNSAT—no such input exists, indicating that the property holds; and (2) SAT, accompanied by a concrete input x_0 such that $P(x_0)$ and $Q(N(x_0))$ hold, evidencing a violation of the property.

Let us revisit the toy DNN in Fig. 1, and suppose we wish to prove that for every input $x = \langle v_1^1, v_1^2 \rangle$, it holds that the output value $N(x) = v_4^1$ is positive. This is encoded as a verification query by setting $P = (true)$, i.e. the inputs are not restricted, and $Q = (v_4^1 \leq 0)$, which is the opposite of our desired property, and by setting N to be the DNN from Fig. 1. Here, a sound verification engine will return SAT, providing a counter example—e.g., $x = \langle 1, 1 \rangle$, which we have already seen to produce a negative output, $v_4^1 = -18$. Hence, the property does not hold in this case.

Scalability is a major hindrance for existing DNN verification tools. When reasoning about repeated DNN invocations, where the output of the DNN influences its future inputs, as in the context of DRL policies, verification becomes *significantly* more difficult. Specifically, reasoning about t consecutive invocations of a DNN with n nodes requires encoding t copies of the DNN into the underlying verification engine, and so effectively considering a DNN with $t \cdot n$ nodes. Because the verification problem is NP complete [37], the worst-case complexity goes up from 2^n to $2^{t \cdot n}$. Consequently, work on DNN verification is typically limited in three important respects: (1) to fairly small DNNs; (2) to a *single* invocation of the DNN; and (3) to simple, *local* properties, where only a very small range of possible inputs is considered. For instance, many efforts have been devoted to verifying adversarial robustness properties [75] with the objective of establishing, for some *fixed* input α to the DNN, that minor perturbations to α do not result in major changes to the DNN's output [23, 28, 80].

3 VERIFYING DRL SYSTEMS: MOTIVATION AND OPPORTUNITIES

Applying DRL in computer and networked systems contexts entails unique challenges and unique opportunities. We next discuss these challenges and opportunities, and provide a high-level overview of our approach, which combines recent advances in DNN verification with model checking.

Motivation: avoiding bad generalization across MDPs. A key challenge in machine learning and, more specifically, in RL, is *generalization*—the ability of the learned model to adapt to previously unseen data. A standard reason for bad generalization in RL is variability in environmental conditions that is not adequately covered by the finite training data (this was recently highlighted in the context of congestion control in [86]). Hence, even if a DRL-based system fares marvelously during training, formally verifying desired correctness and safety properties for the system is important to eliminate the possibility that these properties are violated when the policy is put into practice (thus mitigating the risk of bad generalization).

Recent efforts to apply DRL in systems contexts, however, also give rise to a largely unexplored challenge, which renders formal verification even more important, as discussed next. The standard notion of generalization in machine learning, and established generalization guarantees, are with respect to the scenario that the training environment (the "training distribution" over data) and the operational environment (the "test distribution") are the same, and bad generalization arises from scenarios encountered during test not being adequately represented in the finite training data. When the two distributions differ significantly, there are no generalization guarantees [6, 10, 64].[1] DRL-based systems, however, are often expected to operate in environments that can significantly differ from their training environments.

This challenge can easily be exemplified through our running example of Internet congestion control. Recent proposals for DRL congestion control (see [1, 35]) train a DRL policy for congestion control on synthetic network environments using a network emulator. This training environment naturally fails to capture many of the intricacies of real-world networks. The goal of these research efforts is to show that congestion control policies learned by training on rich synthetic network environments can provide good performance even when applied to real-world network conditions. Phrased using RL terminology, this translates to expecting a policy learned under a certain Markov decision process (MDP) [11] to perform well under a *different* MDP. Importantly, even if a DRL policy is trained within the very same network environment in which it is intended to operate (e.g., using empirical datasets, or by training the policy *in situ* [86], i.e., in live deployment), the network conditions the trained policy will encounter can easily deviate from its training environment due to unexpected changes in network

[1]While the mismatch between training and test distributions is the main challenge in *off-policy RL* [48] (see theoretical guarantees in [24, 32, 76–78] and recent work on offline deep RL [22, 44, 67]), this mismatch is due to a change in *policy*, not in *state transitions*, as in our context.

conditions (routing changes, the appearance of new traffic sources, etc.). This, again, gives rise to the challenge of *generalizing* from one MDP to another.

Thus, DRL-based computer and networked systems are more prone to encountering environmental conditions that trigger bad behavior than the standard examples considered in RL literature, further motivating the formal verification of the correctness and security of such systems.

Opportunity: coupling DNN verification with (bounded) model checking. As discussed in Section 2, the scalability limitations of today's techniques for verifying DNNs render them inapplicable to many domains and properties of interest. Our framework for verifying DRL-based systems hinges on the following crucial observation: the input to the DNN in many recently proposed DRL-based systems consists of rich domain-specific *features*, which were manually handcrafted by human experts, e.g., the recent history of carefully chosen aggregate network statistics for congestion control [1, 35]. This is in contrast to applications of deep learning in domains such as computer vision, where manually generating useful features is extremely challenging, and so the DNN is typically given large volumes of raw visual data and must extract meaningful features from this data.

This rich and meaningful nature of features passed into the DNN has important implications for formal verification: (1) the DNNs used in recent DRL systems tend to be quite small, presumably because extracting useful features from the meaningful, hardcrafted input features requires less manipulation to the DNN's input. This alleviates some of the scalability concerns discussed above. See Table 1 for an illustration; (2) the semantic richness of inputs to the DNN can be leveraged for formulating complex requirements from the system, moving beyond simple, local properties like adversarial robustness properties [75]; and (3) recall that in DRL systems, each output of the DNN translates into an action that can affect its environment, which, in turn, can affect the DNN's next action, and so on. The low dimensionality and inherent structure of the input space simplifies reasoning about how an invocation of the DNN affects its future inputs. This enables reasoning about system executions that include *sequences* of DNN invocations. These characteristics of many recently proposed DRL-based systems make them prime candidates for formal verification and, more specifically, motivate coupling DNN verification with another formal verification technique, called *model checking* [8], which facilitates reasoning about the ongoing behavior of a system.

Model checking involves formalizing, for a considered system, a state-transition graph capturing the possible evolution of system-environment interaction over time, as well as the property to be verified, which pertains to sequences of states in the state-transition graph. Applying model checking to our environments of interest thus requires grappling with questions regarding how desired properties should be formulated and how state-transitions should be captured. Through the examination of our three case studies, we identify useful abstractions. In particular, we observe that, in many cases, a natural and important requirement from a system is to not exhibit long-term, yet *time-bounded* bad behavior (for instance, a congestion controller should not keep the sending rate "too low" for "too long" when network conditions are excellent). We explain how such properties can be formalized as "bounded liveness" queries to whiRL. In addition, we observe that, for many DRL-based systems, the states observed by a system reflect a sliding window over its recent history of observations regarding its environment. We hence incorporate into whiRL the notion of "history buffers" to accommodate the scalable verification of such systems, by curtailing the search space that the model checking procedure explores. To enhance the scalability of our approach, we adopt the notion of *bounded model checking* for contending with the infinite, or prohibitively large, size of the state-transition graph induced by many DRL-based systems.

We show that the above combination of techniques enables the verification of natural, non-local properties of sequential decision making for DRL-based systems of interest.

4 VERIFYING DRL POLICIES WITH WHIRL

We next present a detailed exposition of our verification approach and the whiRL verification platform.

4.1 Applying Model-Checking to DNN Policies

In order to perform model checking, one must specify: (i) the state space for the system, $\mathcal{S}$; (ii) a set of initial states within $\mathcal{S}$, given by a predicate $I(x)$ that returns true iff $x \in \mathcal{S}$ is an initial state; (iii) a transition relation, $T(x, x')$ which specifies to which states x' the system can transition in a single step from state x; and (iv) some property to be verified; for example, that the system never transitions into a particular bad state. We next explain on how we specify each of these components, and how we perform the actual check that the property in question holds.

Defining the state space. In our verification framework, the states correspond to the possible inputs to the DRL agent's DNN. Thus, state transitions enable reasoning about how the inputs to the DRL-based system evolve as a consequence of its actions and the environment. For instance, if the DNN in question has s_0 continuous inputs, each with a (possibly infinite) lower bound l_i and an upper bound u_i, then the state space $\mathcal{S}$ is simply the hyper-rectangle $[l_1, u_1] \times ... \times [l_{s_0}, u_{s_0}]$. Observe that in this case, the set $\mathcal{S}$ is infinite; indeed, this is supported by our technique, and is one of the advantages that model checking and verification afford in this context.

Defining the initial state. The initial state is property-specific, that is, it depends greatly on what we are trying to prove. As an example, if we wish to show that some bad state is never reached, regardless of the initial state, we could set $I = true$. We also give examples of more restricted I's when we discuss our case studies.

Defining the transition relation. Of the four components required for model checking, this is the most complex to define in the context of a DRL agent. Suppose we are currently in some system state, $x \in \mathcal{S}$. By definition, this state is an input to the DNN policy, for which it produces some output, $N(x)$, which induces an action that the system takes. Next, the environment may react to this action, after which a fresh input x' will be passed to the DNN, representing the next state. Assuming the environment's response is not uniquely determined by $N(x)$, which is the typical case (e.g.,

Table 1: DNN sizes for learning-augmented computer and networked systems.

System	Application Domain	# Neurons
Aurora [35]	congestion control	48
NeuroCuts [49]	packet classification	1024
[62]	SQL optimization	50
NEO [60]	SQL optimization	~1500
DeepRM [55]	resource allocation	20
[85]	resource allocation	96
[51]	resource & power management	30
[43]	compiler phase ordering	68
[63]	device placement	320
Placeto [2]	device placement	2× input size
Decima [59]	spark cluster job scheduling	48
Pensieve [57]	adaptive video streaming	384
AuTO [13]	traffic optimizations	1200

is true for all case studies we examined), there can be multiple states x' that satisfy this criterion.

The precise definition of the transition relation T is property-specific. When precisely capturing the definition of T is too complex, our strategy is making T overly *permissive*, i.e., we define T so that if $T(x, x')$ is false then x' indeed cannot be reached from x, but if $T(x, x')$ is true, x' might also actually not reachable from x. This is known as *over-approximation* in verification terminology [14] and guarantees the soundness of our approach; if our verification procedure outputs that the desired property holds, then indeed no violating runs exist. However, if the verification procedure returns a counter-example, it may be spurious, i.e., it may contain a transition $x \longrightarrow x'$ that cannot occur in practice. Of course, in cases where T can be precisely characterized (and so $T(x, x')$ is true if and only if x' is reachable within a single step from x), any counter-example discovered is guaranteed to be a true counter-example.

In our case studies, we observed that the presence of sliding window "history buffers" as the possible inputs to the DRL-system poses the restriction on T that x and x' must agree on the shared history and on the action that was selected in x and led to x'. This also contributes to the scalability of the verification process by reducing the overall number of states that need to be explored.

Going back to the congestion control example from Section 2 and the implementation described in [35], a state x, which is an input to the DNN controller, encodes the recent history of locally-perceived packet loss rates and delays. When the DNN chooses a sending rate, the environment responds by producing a new packet loss rate and packet-latency-related statistics, which constitute the newest entry in the history included in the successor state x'. Thus, for this example, when defining T we specify that the history buffers stored in x' are precisely those in x, but shifted by one, and appended with a new entry. Any state x' for which this does not hold is not a successor state of x.

Defining the property to be verified. We consider two classes of properties: *safety* and *liveness* [8]. These two classes are known to be sufficiently expressive to encode properties of interest in many settings [8] (including, as we show below, in our three case studies).

Safety properties encode that *nothing bad happens* in the system. In the congestion control context, a safety property might state that if the input to the DNN of the traffic sender reflects very high packet loss, then the sender must decrease its sending rate. A safety property is described by a predicate $B(x)$ that returns true iff $x \in \mathcal{S}$ is a *bad* state, i.e., a state in which the safety property is violated. Given a predicate $B(x)$, we say that a *violating run* of the system is a finite sequence of states, $x_1, \ldots, x_n$, such that $I(x_1)$, $T(x_i, x_{i+1})$ for all $i = 1, \ldots, n-1$, and $B(x_i)$ holds for some $1 \le i \le n$; in other words, x_1 is an initial state, each x_i transitions into state x_{i+1}, and at least one of these states is bad. We later discuss how to find such a sequence of states.

Liveness properties encode that *good things eventually happen*. Such properties can express, e.g., that a DRL policy for Internet congestion control must *eventually* increase the sending rate if network conditions are consistently excellent (no packet loss, no self-induced packet delays). A liveness property is violated if the system can enter an infinite loop in which good states are never reached. Given a predicate $G(x)$ that returns true iff x is a good state, a violating run of the system is again a finite sequence of states, $x_1, \ldots, x_n$ such that $I(x_1)$, $T(x_i, x_{i+1})$ for all $i = 1, \ldots, n-1$, there exists a $1 \le j \le n-1$ such that $x_n = x_j$, and for all $1 \le i \le n$ it holds that $\neg G(x_i)$; in other words, x_1 is an initial state, each x_i transitions into state x_{i+1}, the execution loops back to a previously-visited state, and none of the visited states is good.[2] We later discuss how to find such a sequence of states.

Illustrations of safety and liveness violations appear in Fig. 2. Each node in the figure represents a system state, and edges indicate possible successor states. In the left hand side transition system, there is a single state marked as bad. Because that state is reachable from the initial state, there exists a violating run for this safety property. In the right hand side transition system, there is a single state marked as good. Because there is a cycle of states that are

[2]We point out that when the system can run ad infinitum, a liveness violation need not correspond to a loop in the state-transition graph, but can also be in the form of a loop-free infinite sequence of states that contains no good states. Such violations are infeasible to detect using bounded model checking, the verification procedure we describe next. We leave the verification of liveness properties for infinite system runs that do not manifest in cycles of visited states to the future.

not good, and because that cycle is reachable from the initial state, there exists a violating run for this liveness property.

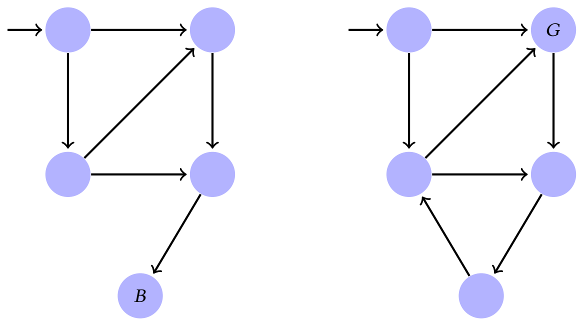

Figure 2: Examples for violated safety and liveness properties.

4.2 Solving the Model-Checking Problem for DRL Policies in Practice

Once the four components described in the previous section (state space, initial states, transition relation and property) are formulated, solving the model checking problem can be performed using standard algorithms [8]. Specifically, for safety properties we can run a search algorithm on the transition system to look for bad states that are reachable from the initial state; and for liveness properties, we can run a nested DFS algorithm that searches for reachable "non-good" cycles [8]. In both cases, for a finite transition system, the problem can be solved in time that is polynomial in the size of the transition graph.

The obstacle, however, is that the transition graphs may be infinite, or prohibitively large. Thus, to render our approach sufficiently scalable for systems of interest, we apply the notion of *bounded model checking* (*BMC*) from the field of formal verification.

In BMC, instead of searching for violating runs *of any length* (for either a safety or a liveness property), we focus instead on runs that have a finite length $k > 0$. This renders the problem much more tractable, while still guaranteeing that any counter-example that we discover is correct. For example, a BMC query for a safety property is formulated as follows:

$$\exists x_1, \ldots, x_k.\ I(x_1) \wedge (\bigwedge_{i=1}^{k-1} T(x_i, x_{i+1})) \wedge (\bigvee_{i=1}^{k} B(x_i))$$

Intuitively, this query is satisfiable iff there is a sequence of consecutive states $x_1, \ldots, x_k$, such that x_1 is an initial state, and there is a bad state reachable from x_1 within k or fewer steps. Because of the limitation on path length, this approach is *incomplete*; violations it detects are correct, but it may overlook other violations (that correspond to longer paths in the state space).

In the case of liveness properties, a BMC query is formulated as follows: $\exists x_1, \ldots, x_k$.

$$I(x_1) \wedge (\bigwedge_{i=1}^{k-1} T(x_i, x_{i+1})) \wedge (\bigwedge_{i=1}^{k} \neg G(x_i)) \wedge (\bigvee_{i=1}^{k-1} x_k = x_i)$$

This formula captures paths of the form $x_1, \ldots, x_k$ such that x_1 is an initial state, each state is a successor of the preceding state, none of the states are good, and for some $1 \le j < k$ the sequence of states $x_j, \ldots, x_k$ forms a cycle (x_j is also a successor of x_k). As before, this formulation only considers paths of length up to k, and so is incomplete.

Observing again the left hand side of Fig. 2, we see that there exists a path from the initial state to the bad state (marked B) for $k = 4$, but not for $k = 1, 2, 3$. On the right-hand side, there exists a path from the initial state that cycles without reaching a good state (marked G) for $k = 5$, but not for $k = 1, 2, 3, 4$.

Bounded model checking can already greatly simplify the complexity of verification queries on DNN policies for DRL systems. Still, for some systems, liveness properties may be difficult to formulate. For instance, if the system state includes a running counter, this will prevent the same state from being revisited even if, apart from the counter, the system is cycling and will never reach a good state. The definition of liveness can be relaxed slightly to address this, into what we refer to as "*bounded liveness*":

$$\exists x_1, \ldots, x_k . I(x_1) \wedge (\bigwedge_{i=1}^{k-1} T(x_i, x_{i+1})) \wedge (\bigwedge_{i=l}^{k} \neg G(x_i))$$

This definition again looks for a path of length k, but this time does not require that the path form a cycle; instead, it only requires that the last $k - l$ steps of the path not be good states for some predetermined k. Note that although this definition involves finite sequences of states, and hence is not a liveness property, it also does not follow our definition of a safety property, that only requires that a *single* bad state be visited.

In bounded model checking, the larger the value of k, the more violations can be discovered. Unfortunately, large k values also lead to computationally expensive verification queries. Hence, we propose to start out with small values of k, and then to gradually increase k as time and resources permit.

4.3 Introducing whiRL

We present a verification platform called *whiRL*, which allows users to verify the correctness of DRL systems with DNN-based policies. Internally, whiRL realizes the aforementioned techniques for bounded model checking, and uses the Marabou [40] off-the-shelf DNN verification engine as a backend. Our code, together with the benchmarks described in Section 5, is available online [20].

A whiRL user is required to provide: (i) the DRL agent's DNN, given in TensorFlow format; (ii) the state space $\mathcal{S}$ of the system, defined by providing lower and upper bounds for each of the DNN's inputs; (iii) a definition for the initial state set, I; (iv) the transition relation $T(x, x')$, given as a set of constraints the connect the values of x' to those of x, and which also takes into account the reactions of the environment; (v) a predicate B defining the bad states (for safety), or a predicate G defining the good states (for liveness); and (vi) the parameter k for BMC queries (see Section 4.2).

Once this information is provided, whiRL generates a BMC query, to be dispatched by the DNN verification engine. Because DNN verification engines take as input $\langle N, P, Q\rangle$ triples as described in Section 2, we must formulate our BMC query accordingly. We do this by creating k copies of the input DNN and combining them into a single, larger network N'. The original input layer is also duplicated, so that it is k times larger and effectively encodes k successive states of the system, which the original network encounters one by one. We use P to express the fact that the first set of inputs adheres to I, and that each intermediate input is obtained from its predecessor according to T; and use Q to specify that at least one state is bad (for safety), or that no states are good (for liveness). An illustration of this encoding appears in Fig. 3. We then pass $\langle N', P, Q\rangle$ to the underlying verifier, and either answer that the system is safe if it returns UNSAT, or provide the counter-example to the user if SAT is returned.

In the definition above, we did not fully specify what kinds of constraints can be included in I, T, B and G. Although our definitions are general, it is useful to restrict these components to contain only *linear or piecewise-linear constraints*, combined using Boolean connectives (e.g., $\vee, \wedge, \neg, \rightarrow$). The reason for this is that almost all available DNN verification engines (e.g., [18, 23, 37, 40, 82]) are restricted to these kinds of constraints. This restriction could be relaxed as DNN tools become more mature and support additional constraints.

We illustrate the entire process with an example. Consider again our toy DNN from Fig. 1. Recall that this DNN receives two inputs, v_1^1, v_1^2, and produce a single output v_4^1. Suppose now that this DNN realizes a policy within a more complex system; when the DNN's output is positive, the environment reacts by increasing the values of the two inputs, and when its output is non-positive, the environment reacts by decreasing the values of the two inputs. Further, suppose that the two input values are always within the range $[-1, 1]$, and that at each step the system can increase or decrease them by at most $1/2$. Finally, suppose that we wish to prove that v_4^1 is always less than 10; and that we formulate this requirement as BMC query of length $k = 3$.

The BMC query is visualized in Fig. 4. Because we set $k = 3$, we encode a larger network that is actually 3 copies of the original network, side by side. This new network has 6 inputs neurons and 3 output neurons (the original 2 inputs and 1 output, times 3). The property P encodes the constraints over the input neurons: that they are always between -1 and 1 (first conjunct); that they must increase by at most $1/2$ if the previous output was positive (second conjunct); and that they must decrease by at most $1/2$ if the previous output was non-positive (third conjunct).

$$\begin{aligned} P = &\bigwedge_{i=1}^{6}(-1 \leq x_i \leq i) \\ &\wedge \bigwedge_{i=1}^{2}\left(y_i > 0 \rightarrow x_i + \frac{1}{2} \geq x_{i+1} \geq x_i\right) \\ &\wedge \bigwedge_{i=1}^{2}\left(y_i \leq 0 \rightarrow x_i - \frac{1}{2} \leq x_{i+1} \leq x_i\right) \end{aligned}$$

Observe that the input property P also refers to output variables. This is supported by many off-the-shelf verification engines, including the Marabou framework that we used. Next, we use the property Q to indicate that a bad state has been reached, i.e. that our desired property is violated: $Q = \bigvee_{i=1}^{3}(y_i \geq 10)$.

The resulting query, i.e. P, Q, and the network from Fig. 4, can be dispatched by the backend verification engine. A SAT answer will constitute a counter-example (in this case, the example is guaranteed to be true, i.e., to not contain any spurious transitions, because we perfectly captured the transition relation T in our verification query); whereas an UNSAT answer will indicate that no counter-example exists for the chosen $k = 3$. In the latter case, we might choose to increase k in search of more complex counter examples.

4.4 Limitations

We next discuss two important limitations of whiRL.

Deterministic DRL policies. Our framework is focused on verifying deterministic DRL policies. Deterministic policies are of great interest in RL due to their high expressiveness. In particular, in sequential decision making under an MDP, there always exists a deterministic policy that is optimal with respect to the objective of maximizing the expected cumulative discounted return. Indeed, various recently proposed DRL-based computer and networked systems use deterministic DRL policies (see, e.g., [1, 35]). That said, many recently proposed DRL-systems do incorporate randomness. One reason for this is that some common (specifically, policy-gradient-based) reinforcement learning algorithms learn stochastic policies to enable the agent to better explore the state space while training (by not re-choosing the same action when the same state is revisited). In addition, when the learned policy is suboptimal, non-determinism adds "noise" to the agent's actions that can potentially prevent the agent from getting stuck in undesirable local optima. We leave the exploration of formal verification of stochastic policies to future research (see discussion in Section 6).

Restrictions on DNNs: feedforward neural networks, piecewise-linear activation functions. Our discussion of DRL verification is focused on feedforward neural networks, as this is, to date, the dominant type of DNNs employed in proposed DRL-based computer and networked systems. We point out, however, that other types of DNNs, namely recurrent neural networks (RNNs), have proved immensely useful in other domains, such as speech recognition. RNNs possess the ability to store information from previous invocations of the DNN in constructs called memory units. We leave the extension of our DRL verification framework to RNNs, e.g., by leveraging ideas and techniques from [3, 34], to the future.

In addition, as discussed above, today's DNN verification engines typically support only piecewise-linear functions (e.g., ReLUs). Thus, DRL-based systems that incorporate other types of activation functions pose difficulties for formal verification. As formal DNN verification techniques evolve to encompass more activation functions, whiRL could also be extended accordingly.

5 EVALUATION: THREE CASE STUDIES

For evaluation purposes, we implemented whiRL as a collection of Python scripts, and integrated it with the recently proposed *Marabou* DNN verification framework [40]. Our code is available online [20].

We next present evaluation results for three learning-augmented computer systems. Our results demonstrate that whiRL can indeed be applied to verify natural requirements for systems of interest (or provide concrete counterexamples for such requirements). We also

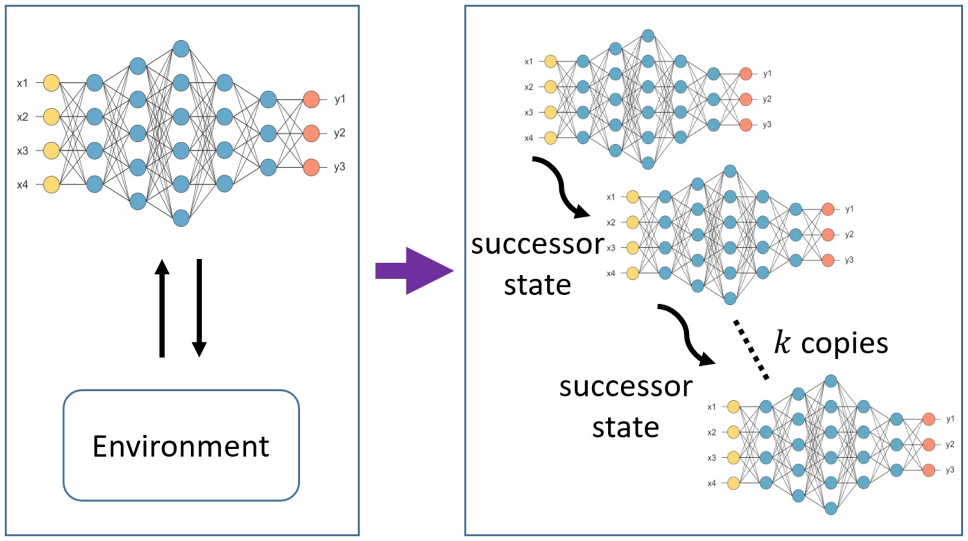

Figure 3: The bounded model checking queries generated by whiRL. The DRL agent and its interactions with the environment (left) are encoded as a DNN that is k times larger than the original (right). As part of the verification query, whiRL enforces that the first set of inputs to the larger neural network is indeed an initial state, and that each consecutive set of inputs is a successor state, based on the previous state and the DNN's selected action.

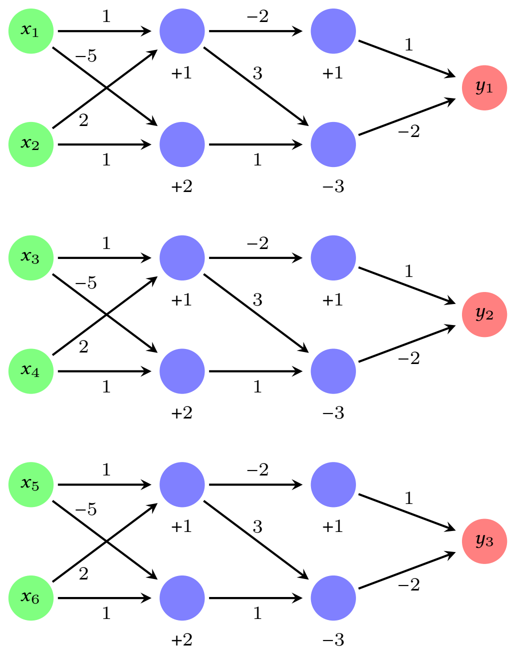

Figure 4: The toy DNN from Fig. 1, duplicated three times in order to be used in a BMC query. The larger network has 6 input nodes and 3 output nodes.

discuss how whiRL can be employed to check whether a learning-augmented system is sufficiently trained (which can be regarded as the ML analogue of acceptance tests for traditional software).

To improve readability and provide intuition, we sometimes settle for an informal description of some the technical details of the systems and refer the reader to the relevant references. All experiments reported below were run on a Intel Xeon E5-2670 v2 machine with 2.50 GHz CPU and 16GB of memory, using a single thread.

Disclaimer: Our negative results for the evaluated systems should be taken with a grain of salt for several reasons: (1) to accommodate the limitations of our framework discussed in Section 4.4, our results pertain to variants of the three systems that are amenable to verification, as will be discussed below; (2) some of the considered systems already incorporate "sanity checks" for the purpose of overriding the DNN in scenarios in which it might yield undesirable actions; and (3) the results are naturally highly dependent on the data on which the DNN is trained and the training duration.

Importantly, our aim is not to suggest that the evaluated systems cannot, with sufficient training on sufficient data, or by incorporating manual DNN-overriding rules, satisfy the considered requirements. The goal of our evaluation is to present evidence that (1) whiRL provides system *designers* with the means to determine whether the training data and training duration are sufficient to guarantee desired system properties, and whether manual rules for overriding the DNN's output might be needed; and (2) whiRL provides system *users* with the means to verify safety and (bounded or unbounded) liveness properties of interest.

As discussed in Section 4.4, as DNN verification engines evolve to support additional (non-piecewise linear) activation functions and additional DNN types (namely, RNNs), whiRL, which treats the DNN verification engine as a black box, could immediately benefit.

We also point out that whiRL could conceivably be leveraged to verify properties that pertain to the system in its entirety, as opposed to the DRL controller alone, so long as manual DNN-overriding rules are properly formulated and incorporated into whiRL's input (more specifically, into the transition relation). We discuss possible extensions of whiRL, left for future work, in Section 6.

5.1 The Aurora Congestion Controller [35]

Aurora [35] is a DRL-based Internet congestion control algorithm. It employs a DNN to map the recent history of observed statistics regarding sent traffic (e.g., experienced latency, throughput) into a change to the sending rate.

Aurora's DNN takes as input a vector with $3t$ entries (for a fixed $t > 0$): (i) t entries represent the observed *latency ratios* for the most recently chosen t sending rates; (ii) t entries represent the observed *sending ratios* for the most recently chosen t sending rates; and (iii) t entries represent the observed *latency gradients* for the most recently chosen t sending rates. Intuitively, the latency ratio is the ratio between experienced latency and the minimum latency observed thus far, the sending ratio is the ratio between the number of packets sent and the number of packets arriving at the destination, and the latency gradient is an indication of whether the observed latency is increasing or decreasing. See [35] for a detailed exposition.

The DNN's single output indicates whether the sending rate should be increased (positive output), decreased (negative output), or maintained (output is zero).

. Encoding Aurora in whiRL. Aurora's DNN controller, trained according to the default settings in [36], is provided as a TensorFlow protobuf. Aurora uses a small, fully connected neural network architecture with hyperbolic tangent activation functions. These non-piecewise-linear activation functions are not supported by many DNN verification tools (e.g., [18, 37, 82]), including Marabou [40], which we use as a backend. To resolve this, we replaced the hyperbolic tangents with rectified linear unit (ReLU) functions, and retrained the DNN. Our DNN demonstrated similar performance (i.e., obtained similar reward values) to that of the original DNN.

The state space of the system is uniquely determined by the DNN's input vectors, as described above. In our evaluation, we set $t = 10$ and so the input vector to our DNN contains $10 \cdot 3 = 30$ entries. We define the transition relation T as follows: for states x_1 and x_2, $T(x_1, x_2)$ is true if and only if the history vectors in x_2 are precisely those in x_1, shifted by one value. In addition, the DNN's output in x_1, denoted $N(x_1)$, determines the sending rate stored in the newest entry in the bounded-length history of x_2.

We define the set of initial states, I, to contain the entire input space, i.e., $I = true$. This reflects the fact that congestion controllers are expected to operate correctly from any starting point (e.g., after changes in underlying routes, the entry or departure of other flows, etc.).

. Safety and Liveness Properties for Aurora. We specify several properties of interest for Aurora:

Property 1. *Intuition:* when the history of local observations reflects excellent network conditions (close-to-minimum latency, no packet loss), the DNN should not get stuck at its current rate. *Type:* liveness (equivalently: the DNN should eventually change the sending rate). The *negation* of a good state: all past latency gradient entries in the DNN's input are in the range $[-0.01, 0.01]$; all past latency ratio entries in the DNN's input are in the range $[1.00, 1.01]$; all past sending ratio entries in the DNN's input are 1; and the DNN's output is 0.

Property 2. *Intuition:* when the history of local observations reflects excellent network conditions (close-to-minimum latency, no packet loss), the DNN should not constantly maintain or decrease its sending rate. *Type:* liveness (equivalently: the DNN should eventually increase the sending rate). The *negation* of a good state: all past latency gradient entries are in the range $[-0.01, 0.01]$; all past latency ratio entries are in the range $[1.00, 1.01]$; all past sending ratio entries are equal to 1; and the DNN's output is not positive.

Property 3. *Intuition*: when the congestion controller is sending on a link with a shallow buffer (and so experienced latency is always close to the minimum latency) and experiences high packet loss, it should decrease its sending rate. *Type:* safety. A *bad* state: all past latency gradient entries are in the range $[-0.01, 0.01]$; all past latency ratio entries are in the range $[1.00, 1.01]$; all past sending ratio entries are at least 2; and the DNN's output is not negative.

Property 4. *Intuition:* when the congestion controller is sending on a link with a shallow buffer (and so experienced latency is always close to the minimum latency) and consistently experiences high packet loss, it should not indefinitely maintain or increase its sending rate. *Type:* liveness (equivalently: the DNN should eventually decrease the sending rate). The *negation* of a good state: all past latency gradient entries are in the range $[-0.01, 0.01]$; all past latency ratio entries are in the range $[1.00, 1.01]$; all past sending ratio entries are at least 2; and the DNN's output is not negative.

When checking for liveness violations, our queries search for a cycle of non-good states. Such cycles have a very specific structure. For example, when using bounded model checking to search for cycles of length 2, we seek inputs to the DNN where the history vectors have the form $\langle x, y, x, y, \ldots, x, y\rangle$, and for which the successor states' inputs have the form $\langle y, x, y, x \ldots, y, x\rangle$. Cycles of length 3 will start from a state $\langle x, y, z, x, y, z, \ldots, x, y, z\rangle$, etc.

Evaluation Results. We used whiRL to verify each of the aforementioned properties for varying values of k. For property 1, whiRL could not find a counter-example for any $k \leq 10$. For property 2, whiRL concluded that the property does not hold. Specifically, it provided a counter-example, even for $k = 2$, in which despite experiencing excellent network conditions, the DNN would repeatedly decrease the sending rate until eventually reaching the minimal possible sending rate and staying there. We note that this immediately implies the existence of counter-examples for every $k > 2$ (because cycles need not contain distinct states). For property 3, whiRL again concluded that the property does not hold. In the counter-example (produced for $k = 1$) the DNN maintains the current sending rate, i.e., outputs 0, even in the presence of high and fluctuating rates of packet loss that clearly call for a reduction in sending rate. Finally, for property 4, whiRL could not find a counter-example for $k \leq 8$, implying that the desired property holds for these values of k. For $k = 9, 10$, the tool timed out.

Verifying properties 1–3 took seconds to finish. For property 4, the queries took seconds to solve for $k \leq 3$; minutes for $4 \leq k \leq 6$; hours for $7 \leq k \leq 8$; and timed out after 24 hours for $k \geq 9$. Query solving can be expedited by parallelizing the underlying verification jobs [83].

5.2 The Pensieve Video Streamer [57]

To optimize user quality of experience (QoE) in video streaming, video clients employ adaptive bitrate (ABR) algorithms. An ABR algorithm dynamically selects the bitrate (resolution) at which the next video chunk (say, 4-second video segment) should be downloaded from the server. ABR protocols map local observables, such as the occupancy of the client's playback buffer and the download times of prior chunks, to choices of bitrates for upcoming video chunks. Pensieve [57] is a recently proposed DRL-based ABR algorithm.

Pensieve's DNN takes as input the following vector: (i) one entry represents the bitrate at which the last chunk was downloaded; (ii) one entry represents the current size (in seconds) of the client's playback buffer; (iii) h entries represent the download times for the h previously downloaded video chunks (for some fixed $h > 0$); (iv) h entries represent the network throughput measurements for the h previously downloaded video chunks; (v) m entries represent the available sizes for the next video chunk, where m is the number of bitrates supported by the video streamer; and (vi) one entry represents the number of unwatched chunks left in the video. The DNN's output indicates at which of the m possible bitrates the next chunk should be downloaded.

. Encoding Pensieve in whiRL. Pensieve's DNN, trained as prescribed in [58], is provided as a TensorFlow protobuf. Pensieve's original output is a probability distribution over the possible bitrates. Since our framework is focused on the verification of deterministic policies, in our training of Pensieve, the output is determined to be the bitrate associated with the neuron with the highest value in the last layer of the DNN. The state space of the system consists of all possible input vectors to the DNN (see above). The transition relation T is defined as follows: $T(x_1, x_2)$ is true if and only if (i) the history vectors in x_2 are precisely those in x_1, shifted by one value; (ii) the number of chunks left until the end of the video in x_2 is that of x_1, decreased by one; (iii) the last chosen bitrate in x_2 matches the output of the DNN in x_1; and (iv) the last chunk size (as reflected by the last throughput) in x_2 matches the output of the DNN in x_1.

The initial states I are those states in which (1) only a single chunk was downloaded, and so all history values in the state that do not represent the most recent time step are set to 0; and (2) the first chunk was downloaded at the default bitrate (which is the second lowest bitrate).

. Liveness Properties for Pensieve. We use whiRL to verify that Pensieve behaves reasonably when streaming a full (short) video, in situations where the desirable actions are clear: (1) when network conditions are excellent, the whole video should not be streamed at the lowest bitrate available (SD); and (2) when network conditions are extremely poor, the video should not be streamed at high bitrates (to avoid video rebuffering). Note that both of the aforementioned properties deal with the *eventual* behavior of the system over sequences of inputs. For example, if the DNN controller was to request only one chunk in low resolution although network conditions were good, this might not be a serious issue; however, if it failed to *eventually* increase the resolution, this would be more severe. This fact suggests that both properties should be encoded as liveness properties. However, due to the presence of the "number of remaining chunks" field in the DNN's input, and since this field is decreased with each transition, it is impossible for the system to revisit a previously visited state. We thus formalize the properties as *bounded liveness* properties, instead:

Property 1. *Intuition:* when starting from an initial state, if in each of k successive states chunks were downloaded quickly (more quickly than it takes to play a chunk), the DNN should eventually not choose the worst resolution (SD). *Type:* bounded liveness. The *negation* of a good state: current buffer occupancy is at least a chunk's duration (4 seconds); past chunks' download times are shorter than a chunk's duration; and the DNN outputs the lowest resolution (SD).

Property 2. *Intuition:* when starting at an initial state, if in each of k successive states the buffer is almost empty (at most one chunk is available for playing), and past chunks were downloaded slowly (more slowly than it takes to play a chunk), the resolution picked by the DNN should eventually not be high. *Type:* bounded liveness. The *negation* of a good state: the current buffer occupancy is the duration of a single chunk (4 seconds); past chunk download times are longer than a chunk's duration; and the DNN's output is a higher resolution than SD.

Evaluation Results. We used whiRL to verify properties (1) and (2) above. For each of these properties, we ran queries for varying values of k.

For property 1, whiRL discovered multiple violations. For each k in the range $2 \leq k \leq 8$, the produced counter-example represents a video of duration $4(k+1)$ seconds. Each of the $k+1$ consecutive 4-second-long video chunks comprising the video, with the exception of the first chunk (assumed to be already downloaded), was downloaded at the lowest available resolution, though the DNN's inputs indicated that higher resolutions could and should have been selected. For property 2, we ran similar experiments, again checking all values of k in the range $2 \leq k \leq 8$. Here, whiRL was unable to find any violations, meaning that the property holds for these values of k. The running time of our tool ranged from a few seconds for $k = 2$ to roughly an hour for $k = 8$.

We speculate that property 2 holds while property 1 does not because of the high penalty for video rebuffering events in Pensieve's reward function. Incorporating stronger incentives for better utilizing available bandwidth into the reward function might produce better results with respect to property 1.

5.3 The DeepRM Resource Manager [55]

DeepRM [55] is a DRL-based, multi-resource cluster scheduler for managing cloud computing resources. DeepRM repeatedly assigns available resources to incoming jobs, with the goal of maximizing job throughput.

DeepRM's DNN receives as input (i) the usage status of d different resources, e.g., system CPU and memory; (ii) a queue of M jobs to be scheduled, for a fixed $M > 0$, with the duration and resource requirements for each of these jobs; and (iii) the number of jobs awaiting to be scheduled (the *backlog*) beyond the jobs in the queue. The DNN's output determines the choice of next action: either *schedule* a specific job from the queue of active jobs, or *wait*, i.e., do not schedule any job at this time. When a job is scheduled, the status of available system resources is updated accordingly, the job is removed from the active jobs queue, and a new job from the backlog might take its place. As time progresses, the execution of scheduled jobs progresses, resources are freed, and the system may schedule new jobs.

. Encoding DeepRM in whiRL. DeepRM's DNN is provided as a Theano model, trained according to the default settings in [54], which we converted to a TensorFlow protobuf. Similarly to Pensieve, DeepRM's original output is a probability distribution over possible decisions and so we trained a variant DeepRM that yields a deterministic decision. Each state in the system's state space is an image of size $(d \times (M + 1) \times 10 + \text{backlog size}) \times 20$, which represents the allocation of cluster resources and the resource profiles of jobs waiting to be scheduled. The transition relation T is defined as follows: for two states x_1 and x_2, it holds that $T(x_1, x_2)$ is true if and only if: (i) if a job from x_1's queue was scheduled, resources are updated accordingly in x_2 and the job no longer appears in x_2's queue; and (ii) if a *wait* action was selected, x_2's queue is the same as x_1's, but the resource allocation is updated accordingly.

. Safety Properties for DeepRM. We specify four safety properties for DeepRM. Consider a scenario where system resources consist of 10 units of CPU and 10 units of memory (for an appropriate definition of units). We consider two types of jobs: *small jobs*, which require 1 unit of each resource for a single time unit (a single time step), and *large jobs*, which require the entire resource pool for 20 time units. We apply whiRL to DeepRM to verify the following properties:

Property 1. *Intuition:* when many system resources are available, and there are small jobs waiting to be scheduled, at least one of them should be scheduled. *Type:* safety. A *bad* state: CPU and memory are only 50% utilized; there are five small jobs in the queue; and the DNN's output is "wait".

Property 2. *Intuition:* when all system resources are free and there is a single large job in the queue, it should be scheduled. *Type:* safety. A *bad* state: CPU and memory are 0% utilized; there is one large job in the queue; and the DNN's output is "wait".

Property 3. *Intuition:* when system resources are not available and there are five small jobs waiting in the queue, no job should be scheduled. *Type:* safety. A *bad* state: CPU and memory are 100% utilized; there are five small jobs in the queue; and the DNN's output is not "wait".

Property 4. *Intuition:* when system resources are not available and there are five large jobs waiting in the queue, no job should be scheduled. *Type:* safety. A *bad* state: CPU and memory are 100% utilized; there are five large jobs in the queue; and the DNN's output is not "wait".

Evaluation Results. For properties 2, 3, and 4, whiRL found counter-examples evidencing that the properties do not hold already for $k = 1$. In contrast, whiRL was able to verify property 1, proving that if all system resources are free and there is a single large job waiting in the queue, that job will be scheduled by the DNN. whiRL's running time for each property was several seconds.

5.4 Verifying Sufficient Training

Our case studies involved verifying properties for already-trained DNNs. However, an important challenge in DRL is determining when training can safely be concluded. We argue that here, too, formal verification could prove useful. We propose to establish a list of properties that a DNN *must* uphold as an acceptance test, and then use whiRL during training to determine when the DNN passes this test. We applied this methodology to Aurora and Pensieve, using the aforementioned properties as our acceptance tests.

For Aurora, we trained the DNN over 7 training episodes. Using whiRL, we observed that the properties that the fully trained network upheld (properties 1, 4) were learned already after the first training episode. The remaining properties, i.e. those that did not hold also for the final system, did not hold at any point in the training process. For Pensieve, we trained the DNN over 10 training episodes. Applying the same methodology, we again observed that the safety properties that held for the final system were learned at a very early stage.

We hypothesize that these results reflect the fact that the reward functions used in both cases assign very high penalties to specific unwanted scenarios (e.g., video rebuffering in the context of Pensieve), leading the DNN to quickly learn some desired properties pertaining to these scenarios, while failing to learn others even over prolonged periods of training. Examining which properties the DRL policies satisfies during training can aid the system designer in reasoning about the effects the different choices of reward functions, and in identifying which manual safeguards should be introduced for overriding the DNN's decisions.

6 DISCUSSION AND FUTURE RESEARCH

Lessons for DRL-based system design: towards verification-aware design. Our experimentation with whiRL taught us important lessons regarding the design of DRL-based systems. For example, a crucial design element is the choice of inputs to feed into the DNN. Naturally, the DNN's inputs should contain sufficient information to facilitate good decisions. However, the choice of inputs also has important implications for the system's amenability to verification. For example, it is beneficial for the verification process if there are no tightly coupled (i.e., redundant) inputs to the network. This is not the case, e.g., for Pensieve's DNN, where the past throughputs and download times are dependent on each other. Such dependencies tend to make it more difficult to encode DNN verification queries, as some dependencies, e.g., non-linear ones, are not widely supported by existing verification technology. In our case-study we bypassed this issue by focusing on queries in which one of the dependent parameters was fixed.

Another design decision that affects amenability to verification is the choice of neural network architecture, and, specifically, the choice of activation functions. Choosing activation functions supported by many engines (specifically, piecewise-linear functions) will make the resulting system easier to verify.

Future research: improved scalability and understanding through invariant inference. A technique that will allow for more tractably solving the full verification problem for DRL-based agents without resorting to bounded model checking is integrating *invariant inference* techniques [69] into our scheme. An invariant is a logical condition φ that holds for all initial states of the system, i.e., $I(x) \Rightarrow \varphi(x)$, and also continues to hold after each transition of the system, i.e., $\varphi(x) \wedge T(x, x') \Rightarrow \varphi(x')$. Thus, an invariant can be regarded as an over-approximation of all reachable systems states, and so can be used for proving that the system satisfies desired safety and liveness properties. Initial work on invariant inference for DNNs [34] shows promise, although significant work will be required to extend these techniques to the DRL setting, which incorporates non-deterministic reactions from the environment. See some initial results in [4].

Beyond improving the scalability of DRL verification, employing invariant inference for verifying DRL-based systems can potentially yield another significant benefit. Specifically, in our evaluation of whiRL in Section 5, the properties considered contained somewhat arbitrary constants. For instance, to capture low network latency, the latency gradient entries and latency ratio entries in the input to Aurora's DNN were constrained to the ranges $[-0.01, 0.01]$ and $[1.00, 1.01]$, respectively. This choice of constants to plug into the verification query is fairly ad hoc. While system designers/users can gain insights into the choices of parameters for which a certain property holds by formulating queries with varying choices of constants and observing whiRL's outcome, we posit that a more elegant solution is feasible. To wit, we believe that, in use-cases like the ones considered here, using invariant inference techniques could be leveraged to characterize the restrictions on the DNN's input for which the desired property is satisfied.

Future research: verifying stochastic policies. An important remaining direction for future research is verifying stochastic policies. A stochastic policy maps each observed state to a *probability distribution* over actions (in contrast to a single action for deterministic polices). This renders the verification of safety and liveness properties for such policies more challenging, as the transition relations and the verification properties become more complex [8]. We believe that by leveraging probabilistic model checking techniques, the verification of stochastic policies could be successfully tackled as well [8, 15, 46].

Additional research directions. Other natural directions for future research include applying whiRL to verify additional properties in additional case studies, and exploring to what extent the counter-examples produced by whiRL could be leveraged to improve the training of DRL-based systems (via adversarial training [52, 84]; see [25] for a recent application of adversarial training to networking domains).

7 RELATED WORK

Deep RL has recently been applied to address many challenges pertaining to computer and networked systems. The reader is referred to [31, 56] for recent surveys.

The state of the art in increasing the reliability of systems that use DNN components relies mostly on testing and simulation (e.g., [65, 79]). There are also efforts being directed at increasing the explainability of and interpretability of DNN models through testing, observations, visualization, and additional means (e.g., [29, 53, 87]), recently focusing also on DRL-based networked systems [16].

Our approach for verifying DRL-driven systems uses a DNN verification engine as a black box. whiRL uses the SMT-based Marabou tool [40, 72, 83], but other verification tools could be used in its stead, including abstract-interpretation-based engines [23, 81], symbolic interval reasoners [82], LP-based tools [80], abstraction-refinement based tools [7, 19], or others. Some of these techniques have also been extended to support quantized neural networks [5, 61]. See [50] for a recent survey.

Approaches for verifying the correctness of *non-deep* RL policies include policy extraction, where instead of the DNN, an explainable agent, such as a decision tree [9, 21], is utilized. Recent studies have also tackled the verification of hybrid systems with DNN controllers [17, 73]. This is similar to our context in the sense that it pertains to sequential DNN evaluations, yet the focus of that line of work is on handling the continuous nature of hybrid systems.

Our preliminary results on verifying DRL-based systems appeared in a workshop paper [41]. We note that the evaluation results in [41] are restricted to verifying safety and liveness properties that translate to paths/cycles of the minimal length, and so boil down to traditional one-shot DNN verification.

Recently, researchers have proposed contending with bad generalization of DRL-based systems by monitoring the system *in real time* in an attempt to identify when the decisions reached by the system are no longer reliable, and defaulting to some safe policy when this occurs [66]. This methodology is termed online safety assurance in [66]. Our approach here, which is also motivated by the same concerns, is complementary to online safety assurance; while online safety assurance detects scenarios that trigger bad behavior *after the fact*, our approach can eliminate the possibility of certain system user-/designer-specified scenarios occurring at all.

8 CONCLUSION

DRL holds great promise for computer system design, and is rapidly becoming pervasive. However, despite its excellent performance in many cases, it remains highly opaque. Consequently, DNN-guided decision making comes at the risk of failing to meet even basic performance requirements, and this risk could significantly hinder the deployment of such systems. Common testing-based and simulation-based approaches can mitigate this risk, but cannot eliminate it entirely. We argue that this concern must be addressed before DRL is built into real-world systems. We believe that our proposed framework is a meaningful step towards increasing the reliability of DRL-based systems.

ACKNOWLEDGMENTS

We thank our shepherd, Prof. Mohammad Alizadeh, and the anonymous SIGCOMM reviewers, for their valuable comments and suggestions, which have significantly improved this work. We also thank Prof. Clark Barrett for his insightful comments on an earlier manuscript of this work. This research was partially supported by grants from the Binational Science Foundation (BSF grants 2017662 and 2019798), the Israel Science Foundation (ISF grant 683/18), and Facebook.

REFERENCES

[1] Soheil Abbasloo, Chen-Yu Yen, and H. Jonathan Chao. 2020. Classic Meets Modern: A Pragmatic Learning-Based Congestion Control for the Internet. In *Proc. Annual Conf. of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*. 632–647.

[2] Ravichandra Addanki, Shaileshh Bojja Venkatakrishnan, Shreyan Gupta, Hongzi Mao, and Mohammad Alizadeh. 2019. Learning Generalizable Device Placement Algorithms for Distributed Machine Learning. In *Proc. 33rd Conf. on Advances in Neural Information Processing Systems (NeurIPS)*. 3981–3991.

[3] Michael E. Akintunde, Andreea Kevorchian, Alessio Lomuscio, and Edoardo Pirovano. 2019. Verification of RNN-Based Neural Agent-Environment Systems. In *Proc. 33rd Conf. on Artificial Intelligence (AAAI)*. 6006–6013.

[4] Guy Amir, Michael Schapira, and Guy Katz. 2021. Towards Scalable Verification of RL-Driven Systems. *arXiv preprint arXiv:2105.11931* (2021).

[5] Guy Amir, Haoze Wu, Clark Barrett, and Guy Katz. 2021. An SMT-Based Approach for Verifying Binarized Neural Networks. In *Proc. 27th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 203–222.

[6] Martin Arjovsky, Léon Bottou, Ishaan Gulrajani, and David Lopez-Paz. 2019. Invariant Risk Minimization. *arXiv preprint:1907.02893* (2019).

[7] Pranav Ashok, Vahid Hashemi, Jan Kretinsky, and Stefanie Mohr. 2020. DeepAbstract: Neural Network Abstraction for Accelerating Verification. In *Proc. 18th Int. Symposium on Automated Technology for Verification and Analysis (ATVA)*.

[8] Christel Baier and Joost-Pieter Katoen. 2008. *Principles of Model Checking*. MIT Press.

[9] Osbert Bastani, Yewen Pu, and Armando Solar-Lezama. 2018. Verifiable Reinforcement Learning via Policy Extraction. In *Proc. 32nd Conf. on Neural Information Processing Systems (NeurIPS)*. 2494–2504.

[10] Shai Ben-David, John Blitzer, Koby Crammer, and Fernando Pereira. 2007. Analysis of Representations for Domain Adaptation. In *Proc. 21st Conf. on Advances in Neural Information Processing Systems (NeurIPS)*. 137–144.

[11] Dimitri P Bertsekas. 1995. *Dynamic Programming and Optimal Control*. Vol. 1. Athena scientific Belmont, MA.

[12] Nicholas Carlini, Guy Katz, Clark Barrett, and David Dill. 2017. Provably Minimally-Distorted Adversarial Examples. *arXiv preprint arXiv:1709.10207* (2017).

[13] Li Chen, Justinas Lingys, Kai Chen, and Feng Liu. 2018. AuTO: Scaling Deep Reinforcement Learning for Datacenter-Scale Automatic Traffic Optimization. In *Proc. Conf. of the ACM Special Interest Group on Data Communication (SIGCOMM)*. 191–205.

[14] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2000. Counterexample-Guided Abstraction Refinement. In *Proc. 12th Int. Conf. on Computer Aided Verification (CAV)*. 154–169.

[15] Christian Dehnert, Sebastian Junges, Joost-Pieter Katoen, and Matthias Volk. 2017. A Storm is Coming: A Modern Probabilistic Model Checker. In *Proc. 29th Int. Conf. on Computer Aided Verification (CAV)*. 592–600.

[16] Arnaud Dethise, Marco Canini, and Srikanth Kandula. 2019. Cracking Open the Black Box: What Observations Can Tell Us About Reinforcement Learning Agents. In *Proc. 2019 Workshop on Network Meets AI & ML (NetAI)*. 29–36.

[17] Souradeep Dutta, Xin Chen, and Sriram Sankaranarayanan. 2019. Reachability Analysis for Neural Feedback Systems using Regressive Polynomial Rule Inference. In *Proc. 22nd ACM Int. Conf. on Hybrid Systems: Computation and Control (HSCC)*. 157–168.

[18] Rüdiger Ehlers. 2017. Formal Verification of Piece-Wise Linear Feed-Forward Neural Networks. In *Proc. 15th Int. Symp. on Automated Technology for Verification and Analysis (ATVA)*. 269–286.

[19] Yizhak Elboher, Justin Gottschlich, and Guy Katz. 2020. An Abstraction-Based Framework for Neural Network Verification. In *Proc. 32nd Int. Conf. on Computer Aided Verification (CAV)*. 43–65.

[20] Tomer Eliyahu, Yafim Kazak, Guy Katz, and Michael Schapira. 2021. The whiRL Repository. https://github.com/WhiRLVerification/WhiRL.

[21] Damien Ernst, Pierre Geurts, and Louis Wehenkel. 2015. Tree-Based Batch Mode Reinforcement Learning. *J. Machine Learning Research* 6 (2015), 503–556.

[22] Scott Fujimoto, David Meger, and Doina Precup. 2018. Off-Policy Deep Reinforcement Learning without Exploration. *arXiv preprint arXiv:1812.02900* (2018).

[23] Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin Vechev. 2018. AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation. In *Proc. 39th IEEE Symposium on Security and Privacy (S&P)*. 3–18.

[24] Mohammad Ghavamzadeh, Marek Petrik, and Yinlam Chow. 2016. Safe Policy Improvement by Minimizing Robust Baseline Regret. In *Proc. 30th Conf. on Neural Information Processing Systems (NeurIPS)*. 2298–2306.

[25] Tomer Gilad, Nathan H. Jay, Michael Shnaiderman, Brighten Godfrey, and Michael Schapira. 2019. Robustifying Network Protocols with Adversarial Examples. In *Proc. 18th ACM Workshop on Hot Topics in Networks (HotNets)*. 85–92.

[26] Sumathi Gokulanathan, Alexander Feldsher, Adi Malca, Clark Barrett, and Guy Katz. 2020. Simplifying Neural Networks using Formal Verification. In *Proc. 12th NASA Formal Methods Symposium (NFM)*. 85–93.

[27] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press.

[28] Divya Gopinath, Guy Katz, Corina Păsăreanu, and Clark Barrett. 2018. DeepSafe: A Data-driven Approach for Checking Adversarial Robustness in Neural Networks. In *Proc. 16th. Int. Symp. on on Automated Technology for Verification and Analysis (ATVA)*. 3–19.

[29] David Gunning. 2017. Explainable Artificial Intelligence (XAI). Defense Advanced Research Projects Agency (DARPA) Project.

[30] Amir Haj-Ali, , Qijing Huang, William Moses, John Xiang, Ion Stoica, Krste Asanovic, and John Wawrzynek. 2019. AutoPhase: Compiler Phase-Ordering for HLS with Deep Reinforcement Learning. *arXiv preprint arXiv:1901.04615* (2019).

[31] Ameer Haj-Ali, Nesreen K. Ahmed, Theodore L. Willke, Joseph Gonzalez, Krste Asanovic, and Ion Stoica. 2019. Deep Reinforcement Learning in System Optimization. *arXiv preprint arXiv:1908.01275* (2019).

[32] Josiah P. Hanna, Peter Stone, and Scott Niekum. 2017. Bootstrapping with Models: Confidence Intervals for Off-Policy Evaluation. In *Proc. 31st Conf. on Artificial Intelligence (AAAI)*.

[33] Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. 2017. Safety Verification of Deep Neural Networks. In *Proc. 29th Int. Conf. on Computer Aided Verification (CAV)*. 3–29.

[34] Yuval Jacoby, Clark Barrett, and Guy Katz. 2020. Verifying Recurrent Neural Networks using Invariant Inference. In *Proc. 18th Int. Symposium on Automated Technology for Verification and Analysis (ATVA)*. 57–74.

[35] Nathan Jay, Noga Rotman, Brighten Godfrey, Michael Schapira, and Aviv Tamar. 2019. A Deep Reinforcement Learning Perspective on Internet Congestion Control. In *Proc. 36th Int. Conf. on Machine Learning (ICML)*. 3050–3059.

[36] Nathan Jay, Noga Rotman, Brighten Godfrey, Michael Schapira, and Aviv Tamar. 2019. PCC, Code Repository. https://github.com/PCCproject/PCC-RL.

[37] Guy Katz, Clark Barrett, David Dill, Kyle Julian, and Mykel Kochenderfer. 2017. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In *Proc. 29th Int. Conf. on Computer Aided Verification (CAV)*. 97–117.

[38] Guy Katz, Clark Barrett, David Dill, Kyle Julian, and Mykel Kochenderfer. 2017. Towards Proving the Adversarial Robustness of Deep Neural Networks. In *Proc. 1st Workshop on Formal Verification of Autonomous Vehicles (FVAV)*. 19–26.

[39] Guy Katz, Clark Barrett, David Dill, Kyle Julian, and Mykel Kochenderfer. 2021. Reluplex: a Calculus for Reasoning about Deep Neural Networks. *Formal Methods in System Design (FMSD)* (2021). To appear.

[40] Guy Katz, Derek Huang, Duligur Ibeling, Kyle Julian, Christopher Lazarus, Rachel Lim, Parth Shah, Shantanu Thakoor, Haoze Wu, Aleksandar Zeljić, David Dill, Mykel Kochenderfer, and Clark Barrett. 2019. The Marabou Framework for Verification and Analysis of Deep Neural Networks. In *Proc. 31st Int. Conf. on Computer Aided Verification (CAV)*. 443–452.

[41] Yafim Kazak, Clark W. Barrett, Guy Katz, and Michael Schapira. 2019. Verifying Deep-RL-Driven Systems. In *Proceedings of the 2019 Workshop on Network Meets AI & ML, NetAI@SIGCOMM 2019, Beijing, China, August 23, 2019*. ACM, 83–89. https://doi.org/10.1145/3341216.3342218

[42] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph Hellerstein, and Ion Stoica. 2018. Learning to Optimize Join Queries With Deep Reinforcement Learning. *arXiv preprint arXiv:1808.03196* (2018).

[43] Sameer Kulkarni and John Cavazos. 2012. Mitigating the Compiler Optimization Phase-Ordering Problem using Machine Learning. In *Proc. ACM Int. Conf. on Object Oriented Programming Systems Languages and Applications (OOPSLA)*. 147–162.

[44] Aviral Kumar, Justin Fu, Matthew Soh, George Tucker, and Sergey Levine. 2019. Stabilizing Off-Policy Q-Learning via Bootstrapping Error Reduction. In *Proc. 33rd Conf. on Neural Information Processing Systems (NeurIPS)*. 11761–11771.

[45] Lindsey Kuper, Guy Katz, Justin Gottschlich, Kyle Julian, Clark Barrett, and Mykel Kochenderfer. 2018. Toward Scalable Verification for Safety-Critical Deep Networks. *arXiv preprint arXiv:1801.05950* (2018).

[46] Marta Kwiatkowska, Gethin Norman, and David Parker. 2011. PRISM 4.0: Verification of Probabilistic Real-Time Systems. In *Proc. 23th Int. Conf. on Computer Aided Verification (CAV)*. 581–591.

[47] Ori Lahav and Guy Katz. 2021. Pruning and Slicing Neural Networks using Formal Verification. *arXiv preprint arXiv:?* (2021).

[48] Sergey Levine, Aviral Kumar, George Tucker, and Justin Fu. 2020. Offline Reinforcement Learning: Tutorial, Review, and Perspectives on Open Problems. *arXiv preprint:2005.01643* (2020).

[49] Eric Liang, Hang Zhu, Xin Jin, and Ion Stoica. 2019. Neural Packet Classification. In *Proc. of the ACM Special Interest Group on Data Communication (SIGCOMM)*. 256–269.

[50] Changliu Liu, Tomer Arnon, Christopher Lazarus, Clark Barrett, and Mykel Kochenderfer. 2019. Algorithms for Verifying Deep Neural Networks. *arXiv preprint arXiv:1903.06758* (2019).

[51] Ning Liu, Zhe Li, Jielong Xu, Zhiyuan Xu, Sheng Lin, Qinru Qui, Jian Tang, and Yanzhi Wang. 2017. A Hierarchical Framework of Cloud Resource Allocation and Power Management using Deep Reinforcement Learning. In *Proc. IEEE 37th Int. Conf. on Distributed Computing Systems (ICDCS)*. 372–382.

[52] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. 2017. Towards Deep Learning Models Resistant to Adversarial Attacks. *arXiv preprint arXiv:1706.06083* (2017).

[53] Aravindh Mahendran and Andrea Vedaldi. 2015. Understanding Deep Image Representations by Inverting Them. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*. 5188–5196.

[54] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. 2016. DeepRM, Code Repository. https://github.com/hongzimao/deeprm.

[55] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. 2016. Resource Management with Deep Reinforcement Learning. In *Proc. 15th ACM Workshop on Hot Topics in Networks (HotNets)*. 50–56.

[56] Hongzi Mao, Parimarjan Negi, Akshay Narayan, Hanrui Wang, Jiacheng Yang, Haonan Wang, Ryan Marcus, Ravichandra Addanki, Mehrdad Khani Shirkoohi, Songtao He, Vikram Nathan, Frank Cangialosi, Shaileshh Bojja Venkatakrishnan, Wei-Hung Weng, Song Han, Tim Kraska, and Mohammad Alizadeh. 2019. Park: An Open Platform for Learning-Augmented Computer Systems. In *Proc. 33rd Conf. on Neural Information Processing Systems (NeurIPS)*. 2490–2502.

[57] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. 2017. Neural Adaptive Video Streaming with Pensieve. In *Proc. Conf. of the ACM Special Interest Group on Data Communication (SIGCOMM)*. 197–210.

[58] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. 2017. Pensieve, Code Repository. https://github.com/hongzimao/pensieve.

[59] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. 2019. Learning Scheduling Algorithms for Data Processing Clusters. In *Proc. Conf. of the ACM Special Interest Group on Data Communication (SIGCOMM)*. 270–288.

[60] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A Learned Query Optimizer. *arXiv preprint arXiv:1904.03711* (2019).

[61] Nina Narodytska, Shiva Kasiviswanathan, Leonid Ryzhyk, Mooly Sagiv, and Toby Walsh. 2017. Verifying Properties of Binarized Deep Neural Networks. *arXiv preprint arXiv:1709.06662* (2017).

[62] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S. Sathiya Keerthi. 2018. Learning State Representations for Query Optimization with Deep Reinforcement Learning. In *Proc. 2nd Workshop on Data Management for End-To-End Machine Learning (DEEM)*. 1–4.

[63] Aditya Paliwal, Felix Gimeno, Vinod Nair, Yujia Li, Miles Lubin, Pushmeet Kohli, and Oriol Vinyals. 2019. Regal: Transfer Learning for Fast Optimization of Computation Graphs. *arXiv preprint arXiv:1905.02494* (2019).

[64] Sinno Jialin Pan and Qiang Yang. 2009. A Survey on Transfer Learning. *IEEE Transactions on knowledge and data engineering* 22, 10 (2009), 1345–1359.

[65] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. Deepxplore: Automated Whitebox Testing of Deep Learning Systems. In *Proc. 26th Symposium on Operating Systems Principles (SOSP)*. 1–18.

[66] Noga H. Rotman, Michael Schapira, and Aviv Tamar. 2020. Online Safety Assurance for Deep Reinforcement Learning. *arXiv preprint arXiv:2010.03625* (2020).

[67] Elad Sarafian, Aviv Tamar, and Sarit Kraus. 2018. Safe Policy Learning from Observations. *arXiv preprint arXiv:1805.07805* (2018).

[68] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. 2015. Trust Region Policy Optimization. In *Proc. 32nd Int. Conf. on Machine Learning (ICML)*. 1889–1897.

[69] Rahul Sharma and Alex Aiken. 2016. From Invariant Checking to Invariant Inference using Randomized Search. *Formal Methods in System Design* 48, 3 (2016), 235–256.

[70] David Silver, Aja Huang, Chris Maddison, Arthus Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. 2017. Mastering the Game of Go without Human Knowledge. *Nature* 550, 7676 (2017), 354.

[71] Gagandeep Singh, Rupanshu Ganvir, Markus Püschel, and Martin Vechev. 2019. Beyond the Single Neuron Convex Barrier for Neural Network Certification. In *Proc. 33rd Conf. on Advances in Neural Information Processing Systems (NeurIPS)*. 15098–15109.

[72] Christopher Strong, Haoze Wu, Aleksandar Zeljić, Kyle Julian, Guy Katz, Clark Barrett, and Mykel Kochenderfer. 2020. Global Optimization of Objective Functions Represented by ReLU networks. *arXiv preprint arXiv:2010.03258* (2020).

[73] Xiaowu Sun, Haitham Khedr, and Yasser Shoukry. 2019. Formal Verification of Neural Network Controlled Autonomous Systems. In *Proc. 22nd ACM Int. Conf. on Hybrid Systems: Computation and Control (HSCC)*. 147–156.

[74] Richard Sutton and Andrew Barto. 1998. *Introduction to Reinforcement Learning*. MIT press Cambridge.

[75] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. 2013. Intriguing Properties of Neural Networks. *arXiv preprint arXiv:1312.6199* (2013).

[76] Philip Thomas and Emma Brunskill. 2016. Data-Efficient Off-Policy Policy Evaluation for Reinforcement Learning. In *Proc. 33rd Int. Conf. on Machine Learning (ICML)*. 2139–2148.

[77] Philip Thomas, Georgios Theocharous, and Mohammad Ghavamzadeh. 2015. High-Confidence Off-Policy Evaluation. In *Proc. 29th Conf. on Artificial Intelligence (AAAI)*.

[78] Philip Thomas, Georgios Theocharous, and Mohammad Ghavamzadeh. 2015. High Confidence Policy Improvement. In *Proc. 32nd Int. Conf. on Machine Learning (ICML)*. 2380–2388.

[79] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. Deeptest: Automated Testing of Deep-Neural-Network-Driven Autonomous Cars. In *Proc. 40th Int. Conf. on Software Engineering (ICSE)*. 303–314.

[80] Vincent Tjeng, Kai Xiao, and Russ Tedrake. 2019. Evaluating Robustness of Neural Networks with Mixed Integer Programming. Proc. 7th Int. Conf. on Learning Representations (ICLR).

[81] Hoang-Dung Tran, Stanley Bak, and Taylor Johnson. 2020. Verification of Deep Convolutional Neural Networks Using ImageStars. In *Proc. 32nd Int. Conf. on Computer Aided Verification (CAV)*. 18–42.

[82] Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. 2018. Formal Security Analysis of Neural Networks using Symbolic Intervals. In *Proc. 27th USENIX Security Symposium*. 1599–1614.

[83] Haoze Wu, Alex Ozdemir, Aleksandar Zeljić, Ahmen Irfan, Kyle Julian, Divya Gopinath, Sadjad Fouladi, Guy Katz, Corina Păsăreanu, and Clark Barrett. 2020. Parallelization Techniques for Verifying Neural Networks. In *Proc. 20th Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*. 128–137.

[84] Cihang Xie, Yuxin Wu, Laurens van der Maaten, Alan L. Yuille, and Kaiming He. 2019. Feature Denoising for Improving Adversarial Robustness. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*. 501–509.

[85] Zhiyuan Xu, Yanzhi Wang, Jian Tang, Jing Wang, and Mustafa Gursoy. 2017. A Deep Reinforcement Learning Based Framework for Power-Efficient Resource Allocation in Cloud RANs. In *Proc. IEEE Int. Conf. on Communications (ICC)*. 1–6.

[86] Francis Y Yan, Hudson Ayers, Chenzhi Zhu, Sadjad Fouladi, James Hong, Keyi Zhang, Philip Levis, and Keith Winstein. 2020. Learning in Situ: A Randomized Experiment in Video Streaming. In *Proc. 17th Symposium on Networked Systems Design and Implementation (NSDI)*. 495–511.

[87] Jason Yosinski, Jeff Clune, Anh Nguyen, Thomas Fuchs, and Hod Lipson. 2015. Understanding Neural Networks through Deep Visualization. *arXiv preprint arXiv:1506.06579* (2015).