Towards Scalable Verification of Deep Reinforcement Learning

Guy Amir, Michael Schapira and Guy Katz
The Hebrew University of Jerusalem, Jerusalem, Israel
{guyam, schapiram, guykatz}@cs.huji.ac.il

***Abstract*—Deep neural networks (DNNs) have gained significant popularity in recent years, becoming the state of the art in a variety of domains. In particular, deep reinforcement learning (DRL) has recently been employed to train DNNs that realize control policies for various types of real-world systems. In this work, we present the *whiRL 2.0* tool, which implements a new approach for verifying complex properties of interest for DRL systems. To demonstrate the benefits of *whiRL 2.0*, we apply it to case studies from the communication networks domain that have recently been used to motivate formal verification of DRL systems, and which exhibit characteristics that are conducive for scalable verification. We propose techniques for performing k-induction and semi-automated invariant inference on such systems, and leverage these techniques for proving safety and liveness properties that were previously impossible to verify due to the scalability barriers of prior approaches. Furthermore, we show how our proposed techniques provide insights into the inner workings and the generalizability of DRL systems. *whiRL 2.0* is publicly available online.**

I. Introduction

In recent years, *deep neural networks* (DNNs) [23] have become highly popular due to their ability to produce state-of-the-art results in multiple fields, e.g., image recognition [34], text classification [37], game playing [45], and many others [7]. DNNs used in such contexts have been shown to successfully learn, by training on data, a model that *generalizes* to previously unseen inputs. In particular, *deep reinforcement learning* (*DRL*) [40] has been recently used to train DNNs to learn control policies for complex computer and networked systems, surpassing the state-of-the-art in a variety of application domains, including database management [60], compiler optimization [41], congestion control [27], [39] on the Internet, routing [53], compute-resource scheduling [9], [42], adaptive video streaming [38], [43], and many more.

Despite the overwhelming success of DNNs, many safety issues pertaining to them have been identified [22], [51], demonstrating that although DNN models potentially yield excellent performance, they also suffer from many weaknesses. For instance, it has been shown that DNNs can be manipulated into performing severe errors through only slight distortions to their inputs [17]. This phenomenon, called *adversarial perturbations*, plagues effectively all modern DNNs.

Adversarial perturbations, alongside other safety and security vulnerabilities, have brought about a surge of interest in formally verifying the correctness of DNNs. A plethora of approaches for DNN verification have been proposed in recent years (e.g., [19], [25], [30], [55]). Unfortunately, in general, all proposed tools face significant scalability barriers, which render them unable to verify state-of-the-art, industrial DNNs with millions of parameters. Furthermore, even when applied to small DNNs, these tools are often restricted to verifying simplistic properties. The scalability challenge is further aggravated in the DRL context, which involves *sequential* DNN-informed decision making, and so reasoning about repeated invocations of the DNN, where the outcome of one invocation can influence the input to the DNN in subsequent invocations. Consequently, the applicability of recently introduced DNN verification tools to complex properties and systems of practical interest remains extremely limited.

To begin bridging this gap, we previously introduced a tool called *whiRL 1.0* [16], which enables verifying certain safety and liveness properties, or identifying violations, for practical DRL systems. We demonstrated *whiRL 1.0*'s usefulness by verifying properties of interest for three systems from the *communication networking* domain. We identified such systems to be prime candidates for verification for two main reasons: first, state-of-the-art DNNs in this domain tend to be of moderate sizes, which are within reach of existing verification technology; and second, meaningful and complex specifications can be formulated and verified because the inputs for these systems are carefully handcrafted and reflect important semantic meaning (as opposed to raw pixel data in computer vision applications, for example). *whiRL 1.0*, which combines DNN verification techniques with bounded model checking, uses a black-box DNN verification engine as a backend, and can thus benefit from any future improvements to DNN verification technology. As exemplified by our promising initial results in [16], *whiRL 1.0* constituted a first step towards enhancing the reliability of DRL systems.

Still, *whiRL 1.0* had severe limitations: most notably, although it successfully generated violations of desired properties, it was incapable of proving that properties of practical significance held without making very strong assumptions, e.g., that runs of the considered system terminate within a very small number of steps. However, the executions of real-world systems are often infinite, or finite but consisting of many steps. In such scenarios, *whiRL 1.0* and other DRL verification tools are unable to prove that most relevant properties hold.

In this work, we present *whiRL 2.0* [1] — a verification engine for DRL systems. *whiRL 2.0* significantly extends the capabilities of the original *whiRL 1.0* tool to accommodate verifying complex properties. In particular, while *whiRL 1.0*

https://doi.org/

was limited to verifying basic safety properties, *whiRL 2.0* utilizes *k-induction* techniques for proving both safety and liveness properties of DRL systems. In addition, *whiRL 2.0* uses *invariant inference* techniques to quickly prove properties that could otherwise be quite difficult to verify. *whiRL 2.0* also incorporates *abstraction* methods for providing some visibility into the DRL system's operation. We demonstrate the effectiveness of these techniques by revisiting the three case studies involving state-of-the-art DRL systems to which *whiRL 1.0* has been applied in [16]: the *Aurora* [27] Internet congestion controller, the *Pensieve* [43] adaptive video streamer, and the *DeepRM* [42] compute resource scheduler. We are able to prove various properties of these systems that, to the best of our knowledge, were beyond the reach of prior state-of-the-art tools, including the original *whiRL 1.0* tool.

The rest of this paper is organized as follows. Section II covers basic background on DNNs, DRL systems, and DNN verification. Next, in Section III we present our *whiRL 2.0* verification tool, and describe its novelties and main components. We present *whiRL 2.0*'s semi-automated invariant inference in Section IV, and discuss the tool's implementation in Section V. Our case studies are described in Section VI, followed by related work in Section VII. We conclude in Section VIII.

II. Background

A. Deep Neural Networks and Deep Reinforcement Learning

A deep neural network (DNN) [23] is a directed graph, where the nodes (also called neurons) are organized in layers. In feed-forward DNNs, data flows from the first (*input*) layer, onto a sequence of intermediate (*hidden*) layers, and finally into a final (*output*) layer. The network is evaluated by assigning values to the input layer's neurons, and then iteratively computing the assignment of each of the hidden layers, until reaching the output layer and returning its evaluation to the user.

More specifically, the value of each neuron in the hidden and output layers is computed using the values of neurons in the preceding layer. Each such layer has a *type*, which determines the exact way in which its neuron values are computed. One common layer type is the *weighted sum* layer, in which each neuron is computed as an affine combination of the values of neurons in the preceding layer, based on edge weights and bias values determined as part of the DNN's training process. Another popular layer type is the *rectified linear unit* (*ReLU*) layer, where each node y is connected to a single node x from the preceding layer, and its value is computed by $y = \text{ReLU}(x) = \max(0, x)$. In this paper we will focus on weighted sum and ReLU layers, although there exist many additional layer types, such as *max-pooling* and *hyperbolic tangent*, to which our technique may be extended.

Fig. 1 depicts a toy DNN comprising an input layer with two neurons, followed by a weighted sum layer and a ReLU layer. For input $V_1 = [1, 3]^T$, the second layer's computed values are $V_2 = [18, -3]^T$. In the third layer, the ReLU functions are applied, resulting in $V_3 = [18, 0]^T$. Finally, the network's single output is $V_4 = [54]$.

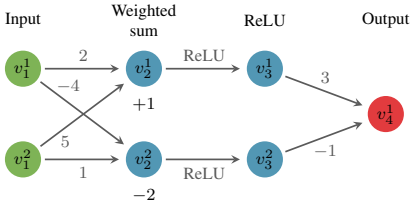

Fig. 1: A toy DNN. The values above the edges are weights, and the values below the vertices are biases.

Formally, a DNN N that receives k inputs and returns n outputs is a mapping $\mathbb{R}^k \rightarrow \mathbb{R}^n$. The DNN consists of a sequence of m layers $L_1, \ldots, L_m$, where L_1 is the input layer and L_m is the output layer. We use s_i to denote layer L_i's size, and $v_i^1, \ldots, v_i^{s_i}$ to denote L_i's individual neurons. We refer to the column vector $[v_i^1, \ldots, v_i^{s_i}]^T$ as V_i. During evaluation, the input values V_1 are fed to the network's input layer, and $V_2, \ldots, V_n$ are computed iteratively.

Each weighted sum layer L_i has a weight matrix W_i of dimensions $s_i \times s_{i-1}$ and a bias vector B_i of size s_i. These W_i and B_i are set at training time, and determine how V_i is computed: $V_i = W_i \cdot V_{i-1} + B_i$. For a ReLU layer L_i, the values of V_i are computed by applying the ReLU to each individual neuron in its preceding layer: $v_i^j = \text{ReLU}(v_{i-1}^j)$.

In *deep reinforcement learning* (*DRL*) [40], a DNN, called the *agent*, learns a *policy* π, which maps each possible observed *environment state* s to an *action* a. During training, at each discrete time-step $t \in 0, 1, 2...$, a *reward* r_t is displayed to the agent, based on the action a_t it chose to perform after observing the environment's state at that time s_t. This reward is used for tuning the agent DNN's weights. The DNNs produced using DRL fall within the same general architecture described above; the difference lies in the training process, which is aimed at generating a DNN that computes a mapping π that maximizes the *expected cumulative discounted return* $R_t = \mathbb{E}\left[\sum_t \gamma^t \cdot r_t\right]$. The *discount factor*, $\gamma \in [0, 1)$, controls the effect that past decisions have on the total expected reward.

B. Verification of Deep Neural Networks

A DNN verification query typically includes a DNN N, a pre-condition P on N's input, and a post-condition Q on N's output [28]. The verification algorithm's goal is to find a concrete input x_0 such that $P(x_0) \wedge Q(N(x_0))$ (the `SAT` case), or prove that no such x_0 exists (the `UNSAT` case). Typically, we use the pre-condition P to express some states of the environment that the network might encounter, and use the post-condition Q to encode the *negation* of the behavior we would like N to exhibit in these states. Thus, when the verification algorithm returns `UNSAT`, this implies that the desired property always holds. Conversely, a `SAT` result indicates that the desired property does not always hold, and this is demonstrated by the discovered counter-example x_0.

For example, observe the toy DNN in Fig. 1, and suppose we wish to verify that the DNN's output is strictly larger than 5, for any input, i.e., for any $x = \langle v_1^1, v_1^2 \rangle$, it holds that $N(x) =$

$v_4^1 > 5$. This is encoded as a verification query by choosing a pre-condition which does not restrict the input, i.e., $P = (true)$, and by setting $Q = (v_4^1 \leq 5)$, which is the *negation* of our desired property. For this verification query, a sound verifier will return SAT, and a feasible counter-example such as $x = \langle 0, -1 \rangle$, which produces $v_4^1 = 0 \leq 5$. Hence, the property does not hold for this DNN.

Verifying DRL Systems. Beyond the general challenges of verifying DNNs (most notably, scalability), verifying DRL systems involves additional challenges. These challenges stem from the fact that DRL agents typically run within reactive systems, and are invoked multiple times, with the inputs to each invocation usually affected by the outputs of previous invocations. This means that (i) the specifications for DRL systems need to account for multiple invocations; and (ii) the scalability issue is aggravated, because the verifier needs to consider multiple consecutive invocations of the network, which is akin to considering a significantly larger DNN.

While attempts have been made to develop tools tailored for DRL system verification (e.g., [16], [32], [44]), two important challenges have yet to be addressed. First, existing verification approaches for DRL systems have focused on refuting properties, and not on proving that they hold; and second, existing approaches were not geared towards verifying reactive systems. As part of the *whiRL* project, we make an initial attempt at addressing these two challenges.

III. *whiRL 2.0*

Our contribution in this paper is the *whiRL 2.0* verification tool, which significantly extends our existing DRL verification engine, *whiRL 1.0*. The *whiRL 2.0* tool allows to verify complex queries on DRL systems, which were previously beyond our reach. Specifically, it supports the verification of safety and liveness properties of DRL systems using a *k-induction*-based approach. Additionally, it incorporates *invariant inference* techniques, which facilitate the verification of complex safety properties. *whiRL 2.0* uses an underlying verification engine as a black-box, and is hence compatible with many existing DNN verifiers.

Formalizing DRL Agents. DRL agents typically operate within reactive systems: they process a (possibly infinite) sequence of states, each representing a current snapshot of the environment observed by the agent. Each state is obtained from its predecessor by triggering the action outputted by the DRL agent, and allowing the environment to react.

In line with the formulation proposed in [16], we formalize the DRL verification problem by encoding the DRL system, as well as its environment, into a transition system $\mathcal{T} = \langle S, I, T \rangle$. Each state $s \in S$ in this transition system is a snapshot of the current observable environment; these states correspond to the inputs of the DNN agent. We use $I \subseteq S$ to denote the set of initial states. The transition relation, $T \subseteq S \times S$, is defined such that $\langle x_i, x_j \rangle \in T$ iff the system can transition from state x_i to state x_j; i.e., when the DNN is presented with state x_i, it selects some action, to which the environment can respond in a way that leads the system to state x_j. Although the DNN is deterministic, the environment is not necessarily so, and so T need not be deterministic. An *execution* of the system is defined as a sequence of states $x_1, \ldots, x_n$, such that $x_1 \in I$, and for all $1 \leq i \leq n-1$ it holds that $T(x_i, x_{i+1})$. The process of encoding a DRL system as a transition system is supported by *whiRL 1.0*, via constructs for representing features common to DRL systems (e.g., inputs in the form of a "sliding window" over the recent history of observations) [16].

Example. As a running example, we focus on the *Aurora* DRL system [27], which implements a congestion control policy. In today's Internet, different services (e.g., video streaming like Netflix and Amazon, VoIP services such as Skype) contend over the same network bandwidth, with aggregate demand for bandwidth often exceeding the available supply. If Internet traffic sources do not pace the rates at which their data is injected into the network, the network will become congested, resulting in data being lost or delayed, and, consequently, in bad user experience and even global Internet outages. Congestion control is the task of determining, for each individual Internet traffic source, how quickly its traffic should be injected into the network at any given point in time. Congestion control is thus a both fundamental and timely networking challenge.

Recently, researchers have proposed employing DRL for this purpose, and presented the Aurora congestion controller [27]. An Aurora-controlled traffic source uses a DNN to select the next rate at which to send traffic, based on observations regarding the implications of its past choices of sending rates. Specifically, Aurora's inputs are t vectors $v_{-t}, \ldots, v_{-1}$, containing performance-related statistics pertaining to the sender's most recent t rate-change decisions. These incorporate information about what fraction of sent data packets were lost following each rate selection, how long it took the sent packets to reach the traffic's destination, etc. The DNN's output determines whether the current rate should be increased, kept steady, or decreased. Changing the sending rate can potentially affect the environment, e.g., an increase to the rate might lead to packet loss if the new rate exceeds network capacity. These changes to the environment, in turn, affect the future inputs to the DNN. See [27] for additional details.

In the formulation of Aurora as a verification challenge in [16], each state, which corresponds to a possible input to Aurora's DNN, is represented by a t-tuple of statistics vectors. The state also contains the DNN's (deterministic) output for the input it represents. This is required for defining good and bad states, as will be discussed later. Congestion controllers are expected to converge to "good" rate decisions from any starting point. Hence, we let the set of initial states be the set of all states. Recall that the input to the DNN represents a sliding window over t-long histories of statistics vectors. Thus, for each two consecutive states, $s_1 \xrightarrow{T} s_2$, it holds that s_2 is obtained from s_1 by augmenting the vectors in s_1 with a statistics vector associated with the DNN's rate change at state s_1, and discarding the vector in s_1 corresponding to the least recent of the t prior rate changes.

DRL System Specifications. Once the DRL system is formulated as a transition system, we can specify safety and liveness properties [11] that it should uphold. *Safety properties* indicate that the system never displays unwanted behavior, and these are often formulated through a predicate $P_B(s)$ that returns true iff $s \in S$ is a bad state, i.e., a state in which the property is violated. The safety verification problem then boils down to determining whether there is a reachable bad state in $\mathcal{T}$ [4]. *Liveness properties* indicate that the system eventually displays desirable behavior, and these are often formulated through a predicate $P_G(s)$ that returns true iff $s \in S$ is a good state, i.e., a state in which the property is fulfilled. Verifying a liveness property is performed by checking that there are no infinite sequences of consecutive states in which only finitely many of the states are good [4]. For instance, a natural safety property with respect to Aurora is that when Aurora observes excellent network conditions (no packet loss, close-to-minimum packet delays), as reflected by the statistics vectors fed to the DNN, the DRL agent does not advise to decrease the sending rate in the *next time-step*. An example of a liveness property in this setting is that if excellent network conditions persist, Aurora should always *eventually* increase the sending rate.

K-Induction. Proving that safety or liveness properties hold (or finding counter-examples) involves traversing large transition system graphs. For modern DRL systems, this is often infeasible, in particular because the rich environments in which these systems operate can react in many ways after each action taken by the agent, resulting in high (or even infinite) out degrees for many states. In *whiRL 1.0*, this issue was addressed through the application of *bounded model checking* (BMC), an approach that explores only a small fraction of the transition system graph, namely, states within a k-step distance from an initial state. BMC can find safety and liveness violations (if they are reachable within k steps) as depicted in Fig. 2, but cannot prove the absence of such violations.

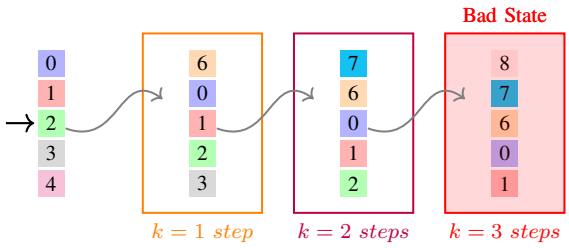

Fig. 2: BMC searches for violations of a safety property. Each vector represents a state, and encodes the statistics that Aurora observed in the past $t = 5$ time-steps. The unwanted state is surrounded by a red rectangle, and is reachable only after $k = 3$ steps from the initial state. Note that consecutive states have shared inputs shifted, and each time-step sample is depicted in a different color.

In *whiRL 2.0*, we address this important gap by adding the means for proving that safety and liveness properties hold. To this end, we employ the method of *k-induction* [11].

Intuitively, the idea in k-induction is to look for state sequences of length k, which can start from arbitrary states in $\mathcal{T}$ (not necessarily from initial states), and for which the property is violated. If a violating execution exists, it must contain an indicative k-long sequence of steps — a suffix of the execution that ends in the bad state for safety properties, or a sequence of non-good states for liveness properties. Thus, if a verifier finds that a k-induction query is UNSAT, we know that the corresponding property holds. If, however, it returns SAT with a counter-example that does not start at an initial state, we cannot conclude whether the property holds, and must increase k in search of a conclusive answer. Fig. 3 depicts a snapshot of the k-induction process used for proving a safety property.

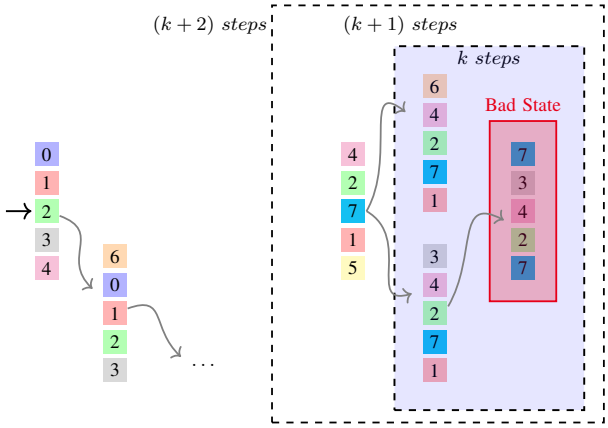

Fig. 3: Using k-induction to prove a safety property, i.e., that the system never reaches the bad state (surrounded by a red rectangle). Although there are k-long and $(k+1)$-long execution sequences that end in the bad state, there is no such sequence of length $(k+2)$; and due to this and to BMC on the base cases, the property holds.

More formally, following the terminology in [4], verifying ω-regular liveness properties is reducible to checking persistence properties of the form *"eventually forever B"*, where B represents a "bad" state ($\exists s \; s.t. \; B = \neg P_G(s)$). Using k-induction in the spirit of [6], [54], we can rule out the existence of k-long sequences of bad states for a given k (even ones not starting at an initial state). This is performed by formulating the following query:

$$\exists x_1, x_2, \ldots, x_k . \Big(\bigwedge_{i=1}^{k-1} T(x_i, x_{i+1}) \Big) \wedge \Big(\bigwedge_{i=1}^{k} \neg P_G(x_i) \Big)$$

for increasingly large values of k. As soon as one such query returns UNSAT, we are guaranteed that the liveness property holds. A similar encoding can be used for proving safety properties.

We note that realizing k-induction in our case-studies entailed contending with challenges such as the need to encode verification queries that capture the system-environment interaction from *any* (possibly non-initial) state. An additional challenge was scalability; duplicating the network to encode k steps can induce an exponential blowup in running time. *whiRL 2.0* curtails the search space by using bound tightening mechanisms, and by enforcing certain dependencies between the inputs to the k duplicate networks encoded as part of a k-

induction query. Specifically, these k inputs typically represent the k recent observations of the agent's environment, and can be restricted by requiring them to constitute a "sliding window": each pair of consecutive inputs must agree on the $k-1$ previous observations that appear in both inputs.

BMC and k-induction are related techniques; the former is geared towards refuting a property, and the latter is geared towards proving it. In *whiRL 2.0*, we take a portfolio approach, as depicted in Fig. 4: we alternate between BMC and k-induction queries, until we: (i) refute the property (BMC returns `SAT`); or (ii) prove the property (k-induction returns `UNSAT`); or (iii) hit a timeout threshold. When steps 1 and 2 both fail, we increment k by 1 and repeat the process. Thus, although we do not know in advance whether the property in question holds, we hope that one of the two techniques will either find a counter-example or prove the property.

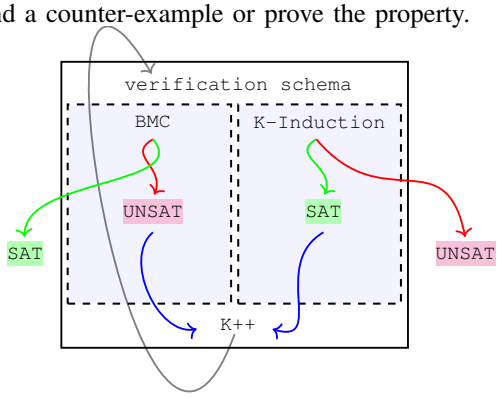

Fig. 4: *whiRL 2.0*'s verification schema.

Abstraction. In computer networking systems, such as the Aurora congestion controller, the system's state is often a set of observations about the environment. Through close inspection of our considered case-studies, we observe that occasionally some of the input fields are irrelevant to the property being checked, in the sense that the property can be proved even when disregarding them. We thus integrate into *whiRL 2.0* *abstraction* capabilities [10] — the ability to strip off irrelevant input fields, as indicated by the user, when dispatching a verification query. The original transition system $\mathcal{T}$ is thus changed into an abstract transition system, $\mathcal{T}'$, which over-approximates the original one. Specifically, the states of $\mathcal{T}'$ are symbolic, each corresponding to multiple states of $\mathcal{T}$; and $s'_1 \xrightarrow{T'} s'_2$ if and only if some states s_1 and s_2, to which s'_1 and s'_2 correspond, satisfy $s_1 \xrightarrow{T} s_2$. If the verification engine concludes that the property holds for $\mathcal{T}'$ (i.e., the negation of the property is `UNSAT`), it follows that it also holds for the original $\mathcal{T}$. However, a counter-example for $\mathcal{T}'$ may be spurious, as it may not be valid for $\mathcal{T}$, in which case the original query may need to be solved to obtain a definite result.

For example, in Aurora, the DNN input represents performance-related statistics pertaining to the t most recent rate adjustments made by the sender. In Aurora's implementation used for our evaluation, we chose $t = 10$ (as in [27]). In this context, abstraction might expose, for instance, that a certain property holds regardless of what values are assigned to the fields not relating to the 5 most recent rate changes, indicating that the policy is, in essence, dependent only on the 5 most recently observed statistics vectors.

We leverage the fact that inputs to recently-proposed computer networked systems consist of fairly few fields with natural semantic meaning, thus leading to a limited number of actual combinations of input fields that are abstracted.

In Section VI we demonstrate how *whiRL 2.0*'s abstraction capabilities can shed light on the inner workings of the verified system, rendering the "black-box" policy learned by the DRL system somewhat more translucent.

IV. Invariant Inference

Verifying DRL systems is difficult, as one must often reason about transitions across many states to establish that a property holds. BMC and k-induction can mitigate this issue to some extent, but sometimes this is not enough. To further boost the scalability of *whiRL 2.0*, we enhanced it with semi-automated *invariant inference* capabilities.

In the context of safety verification of a transition system graph, an *invariant* can be regarded as a partition of the state space S into two disjoint sets, S_1 and S_2, such that no transition leads from one set to the other: $s_1 \in S_1 \wedge s_2 \in S_2 \Rightarrow \langle s_1, s_2 \rangle \notin T$. Invariants are useful if we know that $I \subseteq S_1$ (all initial states are in S_1) and $P_B(s) \Rightarrow s \in S_2$ (all bad states are in S_2). In this case, the existence of the invariant immediately guarantees that no bad states are reachable. Unfortunately, discovering such useful invariants is known to be undecidable in general, and very difficult to accomplish in practice [46].

As part of *whiRL 2.0*, we propose a heuristic for semi-automated invariant inference, which leverages common traits of communication networking systems. More precisely, we observe that many relevant properties in these systems can be regarded as *Boolean monotonic functions*; they tend to be satisfiable when the DNN's input vectors are allowed to fluctuate extensively, but quickly become unsatisfiable when these input vectors are restricted. Often, finding the tipping point, i.e., the minimal input restrictions that cause the property to shift from `SAT` to `UNSAT`, constitutes an invariant that is useful for proving other properties, and which can also render the policy learned by the DNN more translucent to humans.

We demonstrate these notions on the Aurora congestion controller. Recall that Aurora's output indicates whether the sending rate should be increased, maintained, or decreased. *whiRL 2.0* can search for an invariant that translates to the range of inputs for which the DNN outputs that the sending rate should be decreased. Such an invariant can assist in the verification of complex properties, and provide human engineers with comprehensible insights into the DRL system.

Technically, *whiRL 2.0* allows the user to specify the output property and mark the relevant input fields. For example, in Aurora's case, "the sending rate should be decreased" as the output property, and a subset of the input statistics as the relevant fields. Then begins a binary search on the range of the inputs in order to find the minimal restrictions that render

the verification query UNSAT. At each step of the binary search, we invoke a black-box verification procedure to solve the resulting query. This allows us to locate the tipping point up to a prescribed precision. *whiRL 2.0* has built-in *templates* for input and output restrictions, which can be regarded as different strategies for conducting the aforementioned binary search. Each template takes into account either the DRL system's input variables or output variables, and controls them by adjusting their bounds; tightening them to "push" the query towards the UNSAT region. Currently, these templates include (i) for a fixed output, tightening or loosening the bounds of the specified input variables, executing binary search until the point in which the query switches from SAT to UNSAT is discovered; and (ii) performing a similar operation, but this time on the bounds of the specified output variables, while fixing the inputs according to user-specified constants.

Fig. 5 illustrates an invariant search procedure. In this procedure, we have a candidate invariant (the middle blue line) that splits the search space into two parts. Ideally, the reachable states should all be on one side of the partition, and the bad states on the other side. Our binary search automatically adjusts the invariant candidate. In case an initial invariant candidate is too strong (there are reachable states on both sides), it is weakened, and the line is moved towards B. If, however, the initial invariant candidate is too weak (there are bad states on both sides), it is strengthened, and the line is moved towards I. Both kinds of adjustments are performed by tightening or loosening the bounds on the input or output variables.

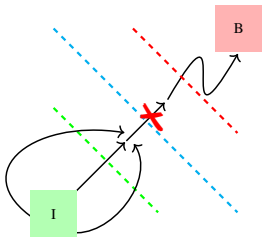

Fig. 5: Invariant search procedure. The initial states are the green square labeled I, and the bad states are the red square labeled B.

V. Implementation

We implemented *whiRL 2.0* as a Python framework that provides general functionality for verifying DRL systems. *whiRL 2.0* uses Marabou [31], a state-of-the-art SMT-based [5], [12], [14] DNN verifier, as a backend (although other verifiers could also be used). *whiRL 2.0* includes the following key modules, which did not exist in *whiRL 1.0*:

1) **K-Induction Query Verifier.** A module that allows the user to generate k-induction queries. The module can encode either a safety property or a liveness property, specified by their $P_B(s)$ and $P_G(s)$ predicates, respectively.
2) **Invariant Finder.** A module through which a user can instruct *whiRL 2.0* to search for an invariant. The user needs to provide the post-condition Q, and mark the variables to focus on. *whiRL 2.0* then performs the previously described semi-automated search procedure, and returns within the specified parameters a range for which the invariant holds, if such a range is found.
3) **Input Abstraction.** A module that allows the user to specify, for a given verification query, which input fields should be abstracted. When abstraction is applied, *whiRL 2.0* will either return UNSAT (if the abstract query returns UNSAT), or default to the original query if the abstract query returns a spurious counter-example.

TABLE I: *whiRL 2.0* features used in each case study.

	Aurora	Pensieve	DeepRM
K-Induction	✓	✓	✗
Bounded Model Checking	✓	✓	✓
Invariant	✓	✗	✓
Abstraction	✗	✓	✓

Additionally, *whiRL 2.0* retains some of *whiRL 1.0*'s functionality, most notably its DNN loading interfaces and bounded model checking capabilities. The code for *whiRL 2.0*, alongside documentation and the experiments described in the paper, are all available online under a permissive license [1]. An appendix with the formulation of the verified properties is also available online [2].

VI. Case Studies

We evaluate *whiRL 2.0* on three case studies of DRL systems: the *Aurora* [27] congestion controller, the *Pensieve* [43] adaptive video streamer, and the *DeepRM* [42] compute resource scheduler. All three case studies, which were used to illustrate the power of *whiRL 1.0* in [16], are from the domain of communication networks. We have identified such DRL systems as highly suitable candidates for evaluating DRL system verification techniques as they achieve state-of-the-art results despite being of moderate sizes, rendering verification tractable. Table I summarizes the *whiRL 2.0* capabilities applied in each case study. All experiments were conducted on an HP EliteDesk machine with six Intel $i5-8500$ cores running at 3.00 GHz, and with a 32 GB memory.

A. The Aurora Congestion Controller

Aurora [27] is a state-of-the-art DRL system that acts as a congestion controller for data transmission [27]. Aurora receives an input vector of size $3t$, which consists of observations from the previous t time-steps. Specifically, the input consists of 3 distinct values representing performance-related statistics for each of the previous t rate changes outputted by the DNN: (i) *latency gradient*: the derivative of latency (packet delays) across time, as measured by the sender, following a change to the rate; (ii) *latency ratio*: the ratio of the average latency experienced by the sender, following a change to the rate, to the minimum past latency experienced. This value is never smaller than 1; and (iii) *sending ratio*: the ratio of the rate at which packets are injected into the network by the sender (i.e., the sending rate), to the rate at which the sent packets arrive at the receiver. We note that the latter rate can be strictly lower than the former rate if the network is congested, which can lead to sent packets being forced to wait in in-network buffers, or being dropped along the way. The sending ratio is never smaller than 1. Intuitively, simultaneous low latency gradient, latency ratio, and sending ratio are indicative

of excellent network conditions. Aurora has a single output value, which indicates whether the sending rate should be increased (positive output), decreased (negative output), or maintained (output is zero). When network conditions are good (low latency, no packet loss), this in indicative of the current rate not overshooting the network bandwidth. Hence, we expect the sending rate to increase so as to take over available bandwidth. In contrast, when network conditions are poor (high latency, high packet loss), this is indicative of network congestion, and so we expect Aurora to decrease the rate. See [16], [27] for additional details.

In line with previous work [16], [27], we set $t = 10$, i.e., the input size to Aurora's DNN is of size $3t = 30$. Aurora's DNN has a single hidden ReLU layer with 48 neurons, and a single neuron in its output layer.

Proving Liveness. In our previous work [16], two liveness properties of Aurora were formulated, but could not be verified using *whiRL 1.0*. Using *whiRL 2.0*, we successfully proved that both properties from [16] always hold. Details follow.

- **Property 1: excellent network conditions eventually imply rate increase.** When Aurora observes a history of excellent network conditions (low latency, no packet loss), the DRL system should *eventually increase* the sending rate, i.e., eventually output positive values. Using *whiRL 2.0*'s k-induction capabilities, we successfully proved that this property, as formulated in [16], indeed holds for any infinite run. The property was successfully proved, within a few seconds, for $k = 2$.
- **Property 2: poor network conditions eventually imply rate decrease.** Symmetrically to property 1, when Aurora observes a history of poor network conditions, the DRL system should *eventually decrease* the sending rate by outputting negative values. By performing k-induction with $k = 5$, we proved that this property, as formulated in [16], indeed holds for all infinite executions. This query took approximately 4.5 hours to solve.

Semi-Automatic Invariance Inference. Next, we used *whiRL 2.0*'s invariant inference capabilities to find invariants for proving safety properties of Aurora.

- **Invariant A: bounding the next-step decrease in sending rate for excellent network conditions.** When Aurora observes a history of excellent network conditions (low latency, no packet loss), the DRL agent's output should be non-negative, i.e., should not imply a decrease to the sending rate. This safety property was shown to be violated in previous work [16]. Here, we utilize *whiRL 2.0*'s invariance inference techniques to prove a bound on this (undesirable) next-step decrease in sending rate, to provide visibility into the performance of the DRL system.

  whiRL 2.0's method for producing the desired invariant appears in Alg. 1. The algorithm takes two user inputs: the *latency slack* ϵ, and the *precision* η. The ϵ input captures the notion of "excellent network conditions" encoded as inputs to the DNN: the observed latency gradient is restricted to the range $[-\epsilon, \epsilon]$; and the observed latency ratio is restricted to the range $[1, 1 + \epsilon]$. Additionally, the sending ratio is set to 1 (indicating that sent traffic arrives at the receiver without being delayed or dropped within the network). The algorithm now performs a binary search over the DNN's output space (leaving the prescribed input ranges for the DNN fixed). Specifically, the η input specifies the desired precision: the output of the algorithm will be an upper bound b on the DNN's output, such that the output b is impossible, but $b + \eta$ is possible, given the aforementioned input restrictions. Recall that the upper bound b relates to the *negation* of the desired property, and so an upper bound of b implies that Aurora's DNN will never decrease the sending rate by b *or more* when network conditions are excellent.
  This procedure terminates within a few seconds, returning an upper bound on the input for which the DNN verifier returns `UNSAT`. The algorithm's correctness immediately follows from the underlying verifier's soundness.

Algorithm 1 *Finding Invariant A*

Input: ϵ, η *// latency slack, precision*
Output: UB_{UNSAT} *// worst-case output decrease bound*

```
UB_UNSAT ← −∞ // −M, for some large constant M
UB_SAT ← 0
QUERY ← DNN VERIFY ( ϵ, output ≤ 0 )
while ( |UB_SAT − UB_UNSAT| ≥ η ) do
    OUT_UPPER ← ½ ( UB_UNSAT + UB_SAT )
    QUERY ← DNN VERIFY (ϵ, output≤ OUT_UPPER )
    if QUERY is SAT then UB_SAT ← OUT_UPPER
    if QUERY is UNSAT then UB_UNSAT ← OUT_UPPER
return UB_UNSAT
```

- **Invariant B: inferring when Aurora fails to decrease the next-step sending rate even though network conditions are poor.** We now wish to characterize poor network conditions in which Aurora does not decrease its sending rate, as expected of it. The procedure is described in Alg. 2. Now, the sending ratio is not fixed to 1, but is rather within the range [1, *P*], for a user-specified *P* value. *P* represents a user-provided upper bound on ratio of the rate at which packets leave the sender (i.e., the sending rate) to the rate which these packets arrive at the receiver. For a slack ϵ, the procedure again restricts the latency gradient to the range $[-\epsilon, \epsilon]$ and the latency ratio to the range $[1, 1 + \epsilon]$. Intuitively, setting low values for ϵ while allowing sending ratios to be high corresponds to sending traffic across communication networks in which in-network buffers are very shallow. In such networks, packets cannot accumulate within the network, resulting in low latencies for packet delivery. However, since in-network buffers are shallow, packets are dropped once network bandwidth is even slightly exceeded, resulting in high sending ratios when the sending rate significantly overshoots the network's capacity (and many packets are lost).
  The algorithm fixes the output's lower bound to be non-negative, and executes a binary search on the input sending

ratio. Specifically, the algorithm returns, for any user-chosen value P, a lower bound (LB_{UNSAT}) such that Aurora always decreases the sending rate when its observations regarding past sending ratios all lie within the range $[LB_{UNSAT}, P]$. *whiRL 2.0* finds the invariant within a few seconds.

Algorithm 2 *Finding Invariant* B

Input: $P \geq 2$ *// upper bound on the sending ratio*
Output: LB_{UNSAT} *// worst-case sending ratio bound*

```
LB_SAT, SR_LOWER ← 1
LB_UNSAT, SR_UPPER ← P
QUERY ← DNN VERIFY ( ϵ, output ≥ 0, SR_LOWER, SR_UPPER )
while ( LB_SAT + 1 < LB_UNSAT ) do
    SR_LOWER ← 1/2 ( LB_SAT + LB_UNSAT )
    QUERY ← DNN VERIFY ( ϵ, output ≥ 0, SR_LOWER, SR_UPPER )
    if QUERY is SAT then LB_SAT ← SR_LOWER
    if QUERY is UNSAT then LB_UNSAT ← SR_LOWER
return LB_UNSAT
```

Observing the bounds produced by Alg. 2 yielded surprising insights regarding the decision-making policy learned by Aurora. Specifically, to gain insight into what our discovered invariants reveal regarding the policies, we created multiple instances of Aurora agents, and trained them all on the same training data until achieving an averaged reward value similar to that of the original Aurora controller [27]. We then observed that for some of the Aurora instances, the discovered invariants depended only on the *proportion* between the sending ratio's lower bound (SR_{LOWER}) and upper bound (SR_{UPPER}), as opposed to their *absolute* values. Specifically, for violating counter-examples (inputs to Aurora's DNN) produced for these instances, the ratio between the highest and lowest past sending ratios was at least 2, with lower ratios giving rise to desirable behavior by Aurora. For other trained instances of Aurora, violating counter-examples only depended on the absolute values of the bounds; e.g., Aurora always decreases the rate for inputs to the DNN where all sending ratios lie in the range $[1, M]$ for some value M, but not when these lie in the range $[1, M + \delta]$ for some small δ. Our findings show that policies that yield the same expected reward on the training set might *generalize* very differently to inputs that lie outside this training set, and that our discovered invariants can shed light on the generalization strategies of different policies learned.

B. The Pensieve Video Streamer

Pensieve is a DRL system [43] for *adaptive bitrate* (ABR) selection. To provide high quality of experience for video clients, Pensieve continuously collects statistics about the client's experience when downloading video chunks (e.g., was the video rebuffered? how long did it take to download the chunk?) to dynamically adapt the resolution at which the next video chunk is downloaded from the video server. Each video chunk represents a fixed-duration video segment (e.g., 4-second-long chunks in our experiments) encoded in one of several possible resolutions (SD, HD, etc.), with higher resolutions corresponding to larger chunks, in terms of number of bits. When client-sensed network conditions are good, we expect the ABR algorithm to decide that the next video chunk will be downloaded in high resolution (HD); and when they are poor, we expect a low resolution (SD) to be selected, to avoid having the client not finish the download in time, which leads to video rebuffering. The input to Pensieve's DNN consists of $(2t + M + 3)$ fields, where $t > 0$ represents the number of recent video chunk downloads considered, and $M > 0$ represents the number of available video resolutions. The input comprises: (i) the *bitrate* (1 field) in which the last video chunk was downloaded; (ii) the current *video buffer size* (1 field) of the client, reflecting the number of seconds of unwatched video stored at the client; (iii) network *throughput measurements* for video chunks downloaded in the past t time-steps (t fields); (iv) *download times* for the video chunks downloaded in the past t time-steps (t fields); (v) *resolution options* (M fields) to download the next chunk; and (vi) the number of *remaining chunks* to be downloaded (1 field). See [43] for a thorough exposition of Pensieve, and [16] for a formalism of the Pensieve verification challenge.

To maintain consistency with Pensieve's original hyper-parameters, in our experiments $t = 8$ and $M = 6$. Due to the nature of an ABR algorithm, all executions are finite (downloads finish in finite time), and so all relevant properties are safety properties. In previous work [16], *whiRL 1.0* was applied to check two safety properties of Pensieve:

- **Property 1.** When the chunk download history represents *excellent conditions* (short download times, large client buffer size), the DRL system should *increase* the resolution at which chunks are requested before the download finishes.
- **Property 2.** When the download history represents *poor network conditions* (long download times, small client buffer size), the DRL system should *decrease* the resolution at which chunks are requested before the download finishes.

While Property 1 was shown not to hold [16], no counter-examples could previously be found for Property 2, and so it could neither be proved nor disproved using existing tools.

Using *whiRL 2.0*, we were able to prove that Property 2 indeed holds under certain, realistic, assumptions.[1] To achieve this, we applied k-induction, with $k = 1$. The result returned by the verifier indicated that the bad states are unreachable, and, hence, that the undesirable behavior cannot occur. These verification queries took approximately 20 minutes to solve.

C. The DeepRM Resource Manager

DeepRM [42] is a DRL-based resource manager, responsible for allocating various cluster compute resources (e.g., CPU, memory) to queued jobs, in order to optimize the cluster's throughput. DeepRM receives the following as input: (i) the *current resource usage* in the system; (ii) a *queue* with up to

[1] We assumed that chunks represent 4-second-long video segments. Considered chunk download times are between 4 to 15 seconds per chunk, which implies that downloading each chunk takes longer than consuming it.

Q pending jobs waiting to be scheduled; and (iii) a *backlog*, indicating the number of jobs waiting to be scheduled that are not yet in the queue. For a fixed Q-sized job queue, the DeepRM controller may output one of $(Q+1)$ possible actions: a *wait* action (i.e., no resources will be allocated at this time-step), or a $schedule_q$ action for $1 \leq q \leq Q$, indicating that job q should be scheduled next. DeepRM's output is interpreted as a probability distribution, assigning a certain probability to each of the $(Q+1)$ possible actions. We refer the reader to [42] for a thorough exposition of DeepRM, and to [16] for a formalism of the DeepRM verification challenge.

In our case study, as in [16], we used a DeepRM system trained with $R = 2$ resources: *CPU* and *memory units*, and a job queue of size $Q = 5$. Overall system resources consist of 10 CPUs and 10 memory units. We considered two kinds of jobs: *small* jobs, which require 1 CPU and 1 memory unit for a single time-step, and *large* jobs, which require 10 CPUs and 10 memory units, for $t = 20$ time-steps.

Previous work [16] considered the following safety properties for DeepRM:

- **Property 1.** When all resources are fully available, and the queue is filled with *small* jobs, DeepRM should never assign the highest probability to the *wait* action.
- **Property 2.** When no resources are available, and the queue is filled with *small* jobs, DeepRM should assign the highest probability to the *wait* action.
- **Property 3.** When no resources are available, and the queue is filled with *large* jobs, DeepRM should assign the highest probability to the *wait* action.

Using *whiRL 1.0*, it was shown [16] that Property 1 holds, and that there exist counter-examples for Properties 2 and 3. However, by using *whiRL 2.0* we were able to prove (within a few seconds) a stronger property that, in fact, generalizes properties 1, 2 and 3. By applying *whiRL 2.0*'s abstraction capabilities to both the inputs indicating resource utilization and the output indicating the recommended action, we proved that for *any* resource utilization level, when the queue is filled with identical jobs, the DRL system's output assigns a higher probability to $schedule_2$ than to *wait*. This immediately proves Property 1, and implies that Properties 2 and 3 cannot hold.

This finding sheds new light on previous results, and enhances our understanding of DeepRM: (i) the three original properties do not depend on the current resource utilization. Rather, due to the DRL system learning a suboptimal policy, it is biased towards scheduling a specific job (job #2), and may fail to select *wait* when appropriate; and (ii) the counter-examples found for Properties 2 and 3 are not outliers, but rather the general case. Indeed, we were able to use *whiRL 2.0* to prove that the inverses of both these properties always hold. These results demonstrate that, beyond proving or disproving specific properties, *whiRL 2.0* can shed light on the policy learned by the DRL system, and expose problematic issues.

VII. Related Work

Due to the increasing use of DNNs, many DNN verification tools have been proposed in recent years; some are SMT-based (e.g., [28], [31], [35], [47]), whereas others use different verification strategies, such as *abstract interpretation* [48], [56], [59], *mixed integer linear programming* (MILP) [52], and many others. Recently, these approaches were extended to verify systems with multi-step executions, such as Recurrent Neural Networks (RNNs) [26], [58] or hybrid systems [50].

In our evaluation of *whiRL 2.0*, we used *Marabou* [31], [57] as a black-box DNN verifier. To date, Marabou has mostly been applied for solving adversarial robustness queries [3], [8], [24], [29], and our work demonstrates that it is also applicable in the field of computer and networked systems. Marabou affords additional features, such as built-in abstraction [15], simplification [20], [36], repair [21] and optimization [49] techniques, which could also be applied to our case studies.

In addition to general DNN verification engines, methods have been devised to formally verify safety properties of DRL systems, which are the subject matter of this work. Such approaches include *shield synthesis* [33], and combining the verification process with *verified runtime monitoring* [18]. Other methods focus on finding adversarial attacks that pertain specifically to DRL agents, e.g., by using MILP [13].

In addition to the *whiRL* project, other approaches have been proposed for verifying DRL systems in the domain of communication networks. These include, e.g., *Verily* [32] and *Metis* [44]. Importantly, however, our focus is on verifying (as opposed to only refuting) various safety and liveness properties of these systems. To the best of our knowledge, this lies beyond the grasp of other existing tools.

VIII. Conclusion

DRL systems provide excellent performance in multiple settings, but suffer from severe vulnerabilities. Several verification tools have been developed to mitigate this concern, but these mostly refute, as opposed to prove, safety and liveness properties of interest. In this work, we presented *whiRL 2.0* — a novel verification engine that supports proving both safety and liveness properties of DRL systems. *whiRL 2.0* accomplishes this through semi-automatic invariance inference, alongside techniques such as k-induction and query abstraction. We demonstrated our tool's capabilities through three case studies from the communication networks domain. In addition, we demonstrated how *whiRL 2.0* can provide insights into the inner workings of these systems, uncovering weaknesses that would otherwise remain unnoticed.

In the future, we plan to enhance our tool's scalability by using improved search heuristics. Also, we intend to enrich the semi-automatic invariant inference templates to support searching for more complex invariants.

Acknowledgements. We thank Nathan Jay, Tomer Eliyahu and the anonymous reviewers for their contributions to this project. The project was partially supported by the Israel Science Foundation (grant number 683/18), the Binational Science Foundation (grant numbers 2017662 and 2019798), and the Center for Interdisciplinary Data Science Research at The Hebrew University of Jerusalem.

REFERENCES

[1] G. Amir, M. Schapira, and G. Katz. Artifact Repository, 2021. https://doi.org/10.5281/zenodo.4769612.

[2] G. Amir, M. Schapira, and G. Katz. Towards Scalable Verification of Deep Reinforcement Learning, 2021. Technical Report. https://arxiv.org/abs/2105.11931.

[3] G. Amir, H. Wu, C. Barrett, and G. Katz. 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)*, pages 203–222, 2021.

[4] C. Baier and J. Katoen. *Principles of Model Checking*. MIT press, 2008.

[5] C. Barrett and C. Tinelli. Satisfiability modulo theories. In *Handbook of Model Checking*, pages 305–343. Springer, 2018.

[6] A. Biere, C. Artho, and V. Schuppan. Liveness checking as safety checking. *Electronic Notes in Theoretical Computer Science*, 66(2):160–177, 2002.

[7] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba. End to End Learning for Self-Driving Cars, 2016. Technical Report. http://arxiv.org/abs/1604.07316.

[8] N. Carlini, G. Katz, C. Barrett, and D. Dill. Provably Minimally-Distorted Adversarial Examples, 2017. Technical Report. https://arxiv.org/abs/1709.10207.

[9] W. Chen, Y. Xu, and X. Wu. Deep reinforcement learning for multi-resource multi-machine job scheduling. *arXiv preprint arXiv:1711.07440*, 2017.

[10] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In *Proc. 12th Int. Conf. on Computer Aided Verification (CAV)*, pages 154–169, 2000.

[11] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem. *Handbook of Model Checking*, volume 10. Springer, 2018.

[12] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[13] A. Dethise, M. Canini, and N. Narodytska. Analyzing learning-based networked systems with formal verification. *IEEE International Conference on Computer Communications (IEEE InfoCom*, 2021.

[14] B. Dutertre and L. De Moura. The yices smt solver. *Tool paper at http://yices.csl.sri.com/tool-paper.pdf*, 2(2):1–2, 2006.

[15] Y. Elboher, J. Gottschlich, and G. Katz. An Abstraction-Based Framework for Neural Network Verification. In *Proc. 32nd Int. Conf. on Computer Aided Verification (CAV)*, pages 43–65, 2020.

[16] T. Eliyahu, Y. Kazak, G. Katz, and M. Schapira. Verifying Learning-Augmented Systems. In *Proc. Annual Conf. of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, 2021.

[17] H. F. Eniser, M. Christakis, and V. Wüstholz. Raid: Randomized adversarial-input detection for neural networks. *arXiv preprint arXiv:2002.02776*, 2020.

[18] N. Fulton and A. Platzer. Safe reinforcement learning via formal methods: Toward safe control through proof and learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2018.

[19] T. Gehr, M. Mirman, D. Drachsler-Cohen, E. Tsankov, S. Chaudhuri, and M. Vechev. AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation. In *Proc. 39th IEEE Symposium on Security and Privacy (S&P)*, 2018.

[20] S. Gokulanathan, A. Feldsher, A. Malca, C. Barrett, and G. Katz. Simplifying Neural Networks using Formal Verification. In *Proc. 12th NASA Formal Methods Symposium (NFM)*, pages 85–93, 2020.

[21] B. Goldberger, Y. Adi, J. Keshet, and G. Katz. Minimal Modifications of Deep Neural Networks using Verification. In *Proc. 23rd Int. Conf. on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, pages 260–278, 2020.

[22] C. Gongye, H. Li, X. Zhang, M. Sabbagh, G. Yuan, X. Lin, T. Wahl, and Y. Fei. New passive and active attacks on deep neural networks in medical applications. In *Proceedings of the 39th International Conference on Computer-Aided Design*, pages 1–9, 2020.

[23] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016.

[24] D. Gopinath, G. Katz, C. Păsăreanu, and C. Barrett. DeepSafe: A Data-driven Approach for Assessing Robustness of Neural Networks. In *Proc. 16th. Int. Symposium on on Automated Technology for Verification and Analysis (ATVA)*, pages 3–19, 2018.

[25] X. Huang, M. Kwiatkowska, S. Wang, and M. Wu. Safety Verification of Deep Neural Networks. In *Proc. 29th Int. Conf. on Computer Aided Verification (CAV)*, pages 3–29, 2017.

[26] Y. Jacoby, C. Barrett, and G. Katz. Verifying Recurrent Neural Networks using Invariant Inference. In *Proc. 18th Int. Symposium on Automated Technology for Verification and Analysis (ATVA)*, 2020.

[27] N. Jay, N. Rotman, B. Godfrey, M. Schapira, and A. Tamar. A deep reinforcement learning perspective on internet congestion control. In *International Conference on Machine Learning*, pages 3050–3059. PMLR, 2019.

[28] G. Katz, C. Barrett, D. Dill, K. Julian, and M. Kochenderfer. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In *Proc. 29th Int. Conf. on Computer Aided Verification (CAV)*, pages 97–117, 2017.

[29] G. Katz, C. Barrett, D. Dill, K. Julian, and M. Kochenderfer. Towards Proving the Adversarial Robustness of Deep Neural Networks. In *Proc. 1st Workshop on Formal Verification of Autonomous Vehicles (FVAV)*, pages 19–26, 2017.

[30] G. Katz, C. Barrett, D. Dill, K. Julian, and M. Kochenderfer. Reluplex: a Calculus for Reasoning about Deep Neural Networks. *Formal Methods in System Design (FMSD)*, 2021. To appear.

[31] G. Katz, D. Huang, D. Ibeling, K. Julian, C. Lazarus, R. Lim, P. Shah, S. Thakoor, H. Wu, A. Zeljić, D. Dill, M. Kochenderfer, and C. Barrett. The Marabou Framework for Verification and Analysis of Deep Neural Networks. In *Proc. 31st Int. Conf. on Computer Aided Verification (CAV)*, pages 443–452, 2019.

[32] Y. Kazak, C. Barrett, G. Katz, and M. Schapira. Verifying Deep-RL-Driven Systems. In *Proc. 1st ACM SIGCOMM Workshop on Network Meets AI & ML (NetAI)*, pages 83–89, 2019.

[33] B. Könighofer, F. Lorber, N. Jansen, and R. Bloem. Shield Synthesis for Reinforcement Learning. In *Proc. Int. Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, pages 290–306, 2020.

[34] A. Krizhevsky, I. Sutskever, and G. Hinton. Imagenet Classification with Deep Convolutional Neural Networks. In *Proc. 26th Conf. on Neural Information Processing Systems (NIPS)*, pages 1097–1105, 2012.

[35] L. Kuper, G. Katz, J. Gottschlich, K. Julian, C. Barrett, and M. Kochenderfer. Toward Scalable Verification for Safety-Critical Deep Networks, 2018. Technical Report. https://arxiv.org/abs/1801.05950.

[36] O. Lahav and G. Katz. Pruning and Slicing Neural Networks using Formal Verification , 2021. Technical Report. https://arxiv.org/abs/2105.13649.

[37] S. Lai, L. Xu, K. Liu, and J. Zhao. Recurrent Convolutional Neural Networks for Text Classification. In *Proc. 29th AAAI Conf. on Artificial Intelligence*, 2015.

[38] A. Lekharu, K. Moulii, A. Sur, and A. Sarkar. Deep learning based prediction model for adaptive video streaming. In *2020 International Conference on COMmunication Systems & NETworkS (COMSNETS)*, pages 152–159. IEEE, 2020.

[39] W. Li, F. Zhou, K. R. Chowdhury, and W. Meleis. Qtcp: Adaptive congestion control with reinforcement learning. *IEEE Transactions on Network Science and Engineering*, 6(3):445–458, 2018.

[40] Y. Li. Deep reinforcement learning: An overview. *arXiv preprint arXiv:1701.07274*, 2017.

[41] R. Mammadli, A. Jannesari, and F. Wolf. Static neural compiler optimization via deep reinforcement learning. In *2020 IEEE/ACM 6th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC) and Workshop on Hierarchical Parallelism for Exascale Computing (HiPar)*, pages 1–11. IEEE, 2020.

[42] H. Mao, M. Alizadeh, I. Menache, and S. Kandula. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM workshop on hot topics in networks*, pages 50–56, 2016.

[43] H. Mao, R. Netravali, and M. Alizadeh. Neural adaptive video streaming with pensieve. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 197–210, 2017.

[44] Z. Meng, M. Wang, J. Bai, M. Xu, H. Mao, and H. Hu. Interpreting deep learning-based networking systems. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 154–171, 2020.

[45] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[46] O. Padon, N. Immerman, S. Shoham, A. Karbyshev, and M. Sagiv. Decidability of Inferring Inductive Invariants. In *Proc. 43th Symposium on Principles of Programming Languages (POPL)*, pages 217–231, 2016.

[47] L. Pulina and A. Tacchella. Challenging smt solvers to verify neural networks. *Ai Communications*, 25(2):117–135, 2012.

[48] G. Singh, T. Gehr, M. Mirman, M. Püschel, and M. T. Vechev. Fast and effective robustness certification. *NeurIPS*, 1(4):6, 2018.

[49] C. Strong, H. Wu, A. Zeljić, K. Julian, G. Katz, C. Barrett, and M. Kochenderfer. Global Optimization of Objective Functions Represented by ReLU networks, 2020. Technical Report. http://arxiv.org/abs/2010.03258.

[50] X. Sun, K. H., and Y. Shoukry. Formal Verification of Neural Network Controlled Autonomous Systems. In *Proc. 22nd ACM Int. Conf. on Hybrid Systems: Computation and Control (HSCC)*, 2019.

[51] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus. Intriguing Properties of Neural Networks, 2013. Technical Report. http://arxiv.org/abs/1312.6199.

[52] V. Tjeng, K. Xiao, and R. Tedrake. Evaluating Robustness of Neural Networks with Mixed Integer Programming, 2017. Technical Report. http://arxiv.org/abs/1711.07356.

[53] A. Valadarsky, M. Schapira, D. Shahaf, and A. Tamar. Learning to route with deep rl. In *NIPS Deep Reinforcement Learning Symposium*, 2017.

[54] T. Wahl. The k-induction principle. *Northeastern University, College of Computer and Information Science*, pages 1–2, 2013.

[55] S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana. Formal Security Analysis of Neural Networks using Symbolic Intervals. In *Proc. 27th USENIX Security Symposium*, pages 1599–1614, 2018.

[56] L. Weng, H. Zhang, H. Chen, Z. Song, C. Hsieh, L. Daniel, D. Boning, and I. Dhillon. Towards fast computation of certified robustness for relu networks. In *International Conference on Machine Learning*, pages 5276–5285. PMLR, 2018.

[57] H. Wu, A. Ozdemir, A. Zeljić, A. Irfan, K. Julian, D. Gopinath, S. Fouladi, G. Katz, C. Păsăreanu, and C. Barrett. Parallelization Techniques for Verifying Neural Networks. In *Proc. 20th Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, pages 128–137, 2020.

[58] H. Zhang, M. Shinn, A. Gupta, A. Gurfinkel, N. Le, and N. Narodytska. Verification of Recurrent Neural Networks for Cognitive Tasks via Reachability Analysis. In *Proc. 24th Conf. of European Conference on Artificial Intelligence (ECAI)*, pages 1690–1697, 2020.

[59] H. Zhang, T. Weng, P. Chen, C. Hsieh, and L. Daniel. Efficient neural network robustness certification with general activation functions, 2018.

[60] J. Zhang, Y. Liu, K. Zhou, G. Li, Z. Xiao, B. Cheng, J. Xing, Y. Wang, T. Cheng, L. Liu, et al. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *Proceedings of the 2019 International Conference on Management of Data*, pages 415–432, 2019.