Verifying Deep-RL-Driven Systems

Yafim Kazak
yafim.kazak@mail.huji.ac.il
The Hebrew University of Jerusalem, Israel

Clark Barrett
barrett@cs.stanford.edu
Stanford University, USA

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

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

ABSTRACT

Deep *reinforcement learning* (*RL*) has recently been successfully applied to networking contexts including routing, flow scheduling, congestion control, packet classification, cloud resource management, and video streaming. Deep-RL-driven systems automate decision making, and have been shown to outperform state-of-the-art handcrafted systems in important domains. However, the (typical) non-explainability of decisions induced by the deep learning machinery employed by these systems renders reasoning about crucial system properties, including correctness and security, extremely difficult. We show that despite the obscurity of decision making in these contexts, *verifying* that deep-RL-driven systems adhere to desired, designer-specified behavior, is achievable. To this end, we initiate the study of *formal verification* of deep RL and present *Verily*, a system for verifying deep-RL-based systems that leverages recent advances in verification of deep neural networks. We employ Verily to verify recently-introduced deep-RL-driven systems for adaptive video streaming, cloud resource management, and Internet congestion control. Our results expose scenarios in which deep-RL-driven decision making yields undesirable behavior. We discuss guidelines for building deep-RL-driven systems that are both safer and easier to verify.

CCS CONCEPTS

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

KEYWORDS

reinforcement learning, neural networks, verification, video streaming, resource management, congestion control

ACM Reference Format:
Yafim Kazak, Clark Barrett, Guy Katz, and Michael Schapira. 2019. Verifying Deep-RL-Driven Systems. In *2019 Workshop on Network Meets AI & ML (NetAI'19), August 23, 2019, Beijing, China.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3341216.3342218

NetAI'19, August 23, 2019, Beijing, China

ACM ISBN 978-1-4503-6872-8/19/08…$15.00
https://doi.org/10.1145/3341216.3342218

1 INTRODUCTION

Today's network protocols and systems are handcrafted by human experts with extensive domain-specific knowledge. Recently, networking researchers have begun exploring a compelling alternative approach: employing *deep learning* [19] machinery to *automate* decision making.

This novel approach to system design builds on the classical reinforcement learning (RL) [31] framework. In RL, an *agent* automatically learns decision rules that yield high performance by continuously evaluating the impact of selected actions on experienced performance. In *deep RL*, these decision rules are learned by training a *deep neural network* (DNN), whose inputs reflect the observed environment state when making the decision, and whose outputs capture the resulting decision. DNNs have revolutionized computer vision [17], speech recognition [10], game playing [28], and other areas, because of their ability to automatically learn, from empirical observations, complex mappings from inputs to outputs.

Deep RL has recently been applied to networking contexts including cloud resource management [23], packet classification [20], routing [33], congestion control [12], and video streaming [24], and also to various additional system design contexts (such as compilers [9] and databases [16]). The performance of these deep-RL-driven systems has been shown to match or even surpass that of handcrafted solutions.

Verifying deep-RL-driven systems. While deep RL is a novel and exciting paradigm for system design, it unfortunately inherits a major drawback of DNNs: the (typical) non-explainability of decisions. Given the opaque nature of DNNs, it is difficult to determine whether a deep-RL-based system satisfies crucial correctness and security requirements. Of course, manually-created software is also prone to error, but many techniques and best practices (e.g., code reviews, refactoring, modular designs) can be applied to mitigate the risk of misbehavior. Unfortunately, these techniques are inapplicable to DNNs.

Here, we advocate employing a *formal verification* approach to address this challenge. Formal verification is an automated and rigorous approach for checking the correctness of systems. We propose formally specifying desired behaviors of a deep-RL-driven system, and then applying formal verification machinery to check whether the system always satisfies the specification.

Introducing *Verily*. We present the first (to the best of our knowledge) formal verification framework for deep RL and use this framework to design *Verily*, a verification tool for deep-RL-driven systems. Verily's design combines two elements from formal verification literature: (1) methodologies for scalable model checking; and (2) recent

developments in the formal verification of DNNs [13], a topic which has received considerable attention of late [7, 8, 11, 13, 15, 34].

Verily can be used to establish that specified requirements from a deep-RL-driven system are satisfied. This is important, e.g., for determining at what point a deep-RL-based system is "sufficiently trained" (similar to acceptance tests for traditional software), and for ensuring that a system achieves desired service-level objectives. Importantly, when Verily determines that the system does *not* satisfy a certain requirement, it provides a concrete scenario (a *counter example*) to demonstrate this. These counter examples can be used to guide changes to the DNN architecture and/or to identify circumstances in which the deep-RL-generated decisions should be overridden.

We evaluate Verily on three deep-RL-driven systems: the Pensieve adaptive video streaming scheme [24], the DeepRM scheduler for cloud resource management [23], and the Custard Internet congestion controller [12]. We formulate natural requirements for each of these systems and apply Verily to determine whether these are always satisfied and, if not, generate counter examples. Our preliminary evaluation results expose several problems in the tested systems, and suggest that the formal verification approach (and, more concretely, the Verily tool) can play an important role in the design and deployment of safer deep-RL-based systems.

2 BACKGROUND

2.1 Deep-RL-Driven Systems

In RL [31], an *agent* observes, at each discrete time step $t \in 0, 1, \ldots$, a *state* of its *environment* s_t and selects an action a_t. After selecting its action, the agent observes a *reward* r_t, representing its loss/gain from selecting a_t. The agent's goal is to choose a policy π, i.e., a mapping of states to actions, which maximizes the *expected cumulative discounted return* $R_t = \mathbb{E}\left[\sum_t \gamma^t \cdot r_t\right]$, for $\gamma \in [0, 1)$. The parameter γ is termed the *discount factor*. Recent advances in deep RL employ deep neural networks to approximate the optimal π [26, 29].

RL provides a useful abstraction for sequential decision making and, in particular, is applicable also when (i) the agent may only possess partial information about the current state; and (ii) the implications of choosing an action may become clear only in hindsight ("delayed rewards").

To illustrate the promise of utilizing deep RL for system design, we briefly discuss its recent application to HTTP-based video streaming [24]. To optimize user quality of experience (QoE), video clients employ adaptive bitrate (ABR) protocols to dynamically select the bitrates (resolutions) of requested video chunks (say, 4-second video segments). ABR protocols map local observables such as the occupancy of the client's playback buffer and the download times of prior video chunks, to choices of bitrates for upcoming video chunks. Today's protocols typically rely on "handwired" mappings from local observables to selected bitrates, devised by human experts. In contrast, the deep-RL-driven Pensieve [24] ABR protocol automatically *learns* high-performance bitrate selection policies from empirical data. Pensieve does so by testing the implications of different bitrate selections for performance, as captured by a reward function that reflects QoE goals such as sending at high bitrates and avoiding client video rebuffering and jitter in bitrates. A DNN is employed to map observables to bitrate selections, thus enabling Pensieve to efficiently learn complex ABR policies. Experimental and empirical evaluations suggest that Pensieve outperforms the widely used handcrafted algorithms, even in environments with different characteristics than those of its training environment.

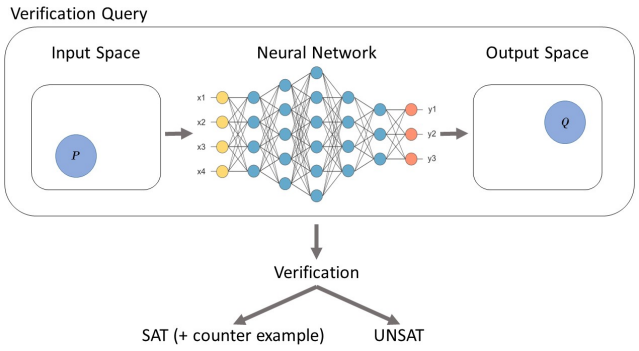

Figure 1: The neural network verification scheme.

2.2 Deep Neural Network Verification

Following the rise in popularity of DNNs, the verification community has begun addressing the need to verify neural networks [3, 7, 8, 11, 13–15, 18, 18, 34]. Because neural networks are constructed from a small set of relatively simple arithmetical operations, the DNN verification problem is decidable (which is often not the case for verification of manually crafted code). However, although decidable, DNN verification is computationally complex (NP-complete) even in simple cases [13], and scalability is a major hindrance for verification tools. Despite this, verification technology is rapidly improving and, as demonstrated by us and by others, existing tools are already sufficient for tacking real-world problems of interest.

A DNN verification query is comprised of the following: (i) a neural network N; (ii) an *input property* P; and (iii) an *output property* Q. A verification engine then tries to answer 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. In other words, the verification engine seeks a particular input x that satisfies the input property P, and is mapped by the neural network to an output that satisfies the output property Q. The verification process has two possible outcomes: (1) an *unsat* result, indicating that no such input exists; and (2) a *sat* result, accompanied by a concrete input x_0 such that $P(x_0)$ and $Q(N(x_0))$ hold. See Fig. 1 for an illustration. Q typically expresses the *negation* of the desired property, and thus an *unsat* result indicates that the property holds, whereas *sat* indicates a violation that occurs for x_0.

An important distinction between verification and other common approaches for quality assurance, such as testing and simulation, is that a single verification query can provide formal guarantees about the behavior of the system for infinitely many inputs. This can help to ensure, e.g., that a DNN operates correctly when presented with inputs that were not part of its training or validation sets. In some cases, verification queries can even be used to explain how certain decisions are reached by the DNN [4].

So far, DNN verification has focused mostly on *supervised learning*, which is prevalent, e.g., in computer vision and natural language processing. Our work is, to the best of our knowledge, the first application of formal verification to deep RL. As discussed below, deep RL poses new challenges for formal verification.

3 VERIFYING DEEP-RL-DRIVEN SYSTEMS

Verifying DNNs in the context of deep RL differs from prior approaches to DNN verification, because in deep RL, the DNN's output at a certain time step (which induces the agent's action at that time step) influences the input to the DNN at later points in time. Thus, verifying deep-RL-driven systems involves reasoning about interrelated *sequences* of DNN evaluations.

We present a scheme for deep RL verification that addresses this challenge by considering the space of all possible environment states $\mathcal{S}$, and using formal verification to identify undesirable sequences of steps through this space.

Specifically, for a particular state $x \in \mathcal{S}$ (which constitutes a possible input to the DNN-based RL agent), let $\mathcal{E}(x)$ denote the set of states reachable from x within a single time step. A state $x' \in \mathcal{E}(x)$ is thus a state that can potentially be reached after applying the DNN to x to obtain the agent's action $N(x)$, and letting the environment react to this action.

Going back to the video streaming example, a state x might encode (as in [24]) the client's current buffer occupancy, download times of the ω most recently downloaded video chunks (for some fixed $\omega > 0$), the number of remaining chunks to download, etc. $N(x)$ encodes the network's bitrate selection for input x, say, SD or HD. $\mathcal{E}(x)$ encodes the states reachable from x by updating the client's buffer occupancy after downloading the video chunk in the selected bitrate $N(x)$, updating the ω most recent download times to incorporate the last downloaded chunk, decreasing the number of chunks to download by 1, etc.

Using the aforementioned formulation, we propose an approach for verifying two kinds of specifications: *safety* properties and *liveness* properties.

Safety properties indicate that *nothing bad happens* in the system. In the video streaming context, for instance, a safety property might capture the desired behavior that if all recently downloaded video chunks were in HD and were downloaded quickly, the requested video resolution should not be changed to SD. Given a predicate $I(x)$ that returns true iff x is a possible initial state of the system, and a predicate $B(x)$ that returns true iff x is a *bad* state, i.e., a state in which the safety property is violated, we create the following query:

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

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. Clearly, such a sequence constitutes a counter example to the property being checked.

This type of query, referred to as *bounded model checking* [2], seeks safety violations of length up to k. Because of the limitation on path length (k), this approach is *incomplete*: violations it detects are correct, but it may overlook other violations (that correspond to longer paths in the state space). However, as we later demonstrate, examining paths of length $k = 2$ is often sufficient for finding counter examples.

Determining that state x_{i+1} is immediately reachable from x_i, i.e., $x_{i+1} \in \mathcal{E}(x_i)$, and that a state x_j is bad, i.e., $B(x_j)$, involves reasoning about the operation of the DNN. This can be accomplished using existing DNN verification tools.

Liveness properties indicate that "good things eventually happen." Such properties can express, e.g., that a DNN that performs cloud resource management never starves a particular job, which is equivalent to specifying that every job eventually gets scheduled. 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, we formulate this property as:

$$\exists x_1, \ldots, x_k.$$
$$I(x_1) \wedge \left(\bigwedge_{i=1}^{k-1} x_{i+1} \in \mathcal{E}(x_i)\right) \wedge \left(\bigwedge_{i=1}^{k} \neg G(x_i)\right) \wedge \left(\bigvee_{i=1}^{k-1} x_k = x_i\right)$$

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 $1 \leq j < k$ 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. DNN verification tools can be applied to validate the succession of states and non-goodness of states.

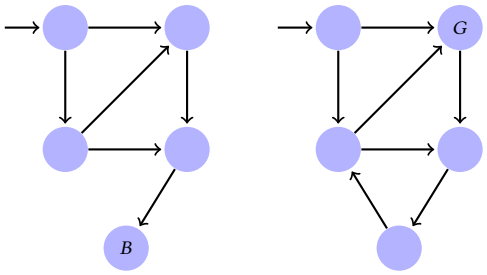

Figure 2: Examples for violated safety and liveness properties.

An illustration of the approach for safety and liveness properties appears in Fig. 2. Each node in the figure represents a system state, and edges indicate possible successor states. On the left-hand side graph, 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 graph, there exists a path from the initial state that cycles without reaching a good state (marked G) for $k = 5$, but not for $k = 1, 2, 3, 4$.

When verifying safety and liveness properties, it is better to consider larger values of k, as this covers more potential violations (all paths of length at most k). However, verification is computationally expensive and so considering larger values of k renders the query more difficult to solve. Hence, a reasonable approach is to start out with small values of k and gradually increase k as time and resources permit. This approach would be particularly appealing

if combined with incremental verification engines: engines that can re-use some of the work done for smaller values of k when solving queries with larger values of k. There exist techniques for incremental verification, although to the best of our knowledge they have not yet been applied in the context of DNNs.

Verily. We build on the aforementioned ideas to implement a verification platform called *Verily*. As its underlying engine, Verily uses the *Marabou* verification tool for deep neural networks [15]. The Marabou tool is based on a *satisfiability modulo theories* (*SMT*) verification engine that operates on feed-forward neural networks with piecewise-linear activation functions. Marabou seeks inputs to a DNN that satisfy an input property P, and for which the DNN's output satisfies an output property Q. Both input and output properties are given as conjunctions of linear constraints. Verily leverages Marabou to resolve bounded model checking queries of the aforementioned forms. Of course, Marabou could be replaced with another verification engine with similar functionality.

4 THREE CASE STUDIES

To demonstrate the usefulness of our approach, we present below preliminary evaluation results. We evaluate Verily on three recent deep-RL-driven systems: (1) the DeepRM online scheduler for cloud resource management [23], (2) the Pensieve adaptive video streamer [24], and (3) the Custard Internet congestion controller [12].

We formulate natural requirements for each of these systems, and use Verily to determine whether these are satisfied or not (and, if not, generate concrete counter examples). Our evaluation of both DeepRM and Pensieve uses the released DNN models, training data, and default configurations for these systems [22, 25]. The code and data for Custard was supplied by the authors of [12]. All of our experiments were conducted on a laptop with an Intel i7-4710MQ, 2.50GHz CPU, using 16GB of memory.

For several of the properties that were tested, Verily was able to provide counter examples. Further, these counter examples were fairly simple, highlighting the possible vulnerabilities of these systems. We regard these findings as evidence of the potential benefits of using verification as part of the design of deep-RL-driven systems.

Disclaimer: Our negative results for the evaluated systems should be taken with a grain of salt for two reasons: (1) some of these systems incorporate "sanity checks" for the purpose of overriding the DNN in scenarios in which it might yield undesirable actions, and (2) 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) Verily 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) Verily provides system *users* with the means to verify safety and liveness properties of interest.

4.1 Verifying Cloud Resource Scheduling

The DeepRM Online Scheduler. DeepRM [23] is a system for managing cloud computing resources. We present below an informal overview of DeepRM and refer the reader to [23] for a detailed exposition.

DeepRM keeps track of (i) the usage status of different resources, e.g., current usage of 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 combination of (i)-(iii) constitutes the input to DeepRM's DNN. 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 queue of pending jobs, 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.

Verifying DeepRM. We consider two types of safety properties: (1) when system resources are available, jobs in the active queue are scheduled (as opposed to waiting), and (2) when system resources are not available, DeepRM does not schedule jobs (and waits for system resources to be freed). We point out that Verily could also be used to verify more elaborate safety properties, e.g., that jobs are scheduled within T time steps.[1]

Consider a scenario where system resources that can be used to execute jobs 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, and *large jobs*, which require the entire resource pool for 20 time units. We apply Verily to DeepRM for the purpose of verifying the following properties:

- **A1:** When system resources, namely CPU and memory, are only 50% utilized (each) and there are five small jobs in the queue, DeepRM's DNN always schedules one of the five jobs.
- **A2:** When system resources are 0% utilized and there is a single large job in the queue, DeepRM's DNN always schedules that job.
- **B1:** When system resources are fully utilized and there are five small jobs waiting in the queue, DeepRM's DNN always outputs a wait action.
- **B2:** When system resources are fully utilized and there are five large jobs waiting in the queue, DeepRM's DNN always outputs a wait action.

Technically, we encoded these properties in a straightforward manner. The existence of pending jobs, the required resources for these jobs, and also the state of the backlog were encoded by fixing the corresponding DNN input values to the appropriate values as part of our input property P. For resource usage, the design of DeepRM's DNN allows for a wide range of values that describe a given situation (e.g., there are infinitely many inputs that describe the fact that a certain CPU is 50% utilized). This range of values is

[1] This would entail incorporating additional information into the states in $\mathcal{S}$ indicating the time that has passed since a job was last scheduled.

intended for visualization purposes [23]. Thus, in order to restrict the resource usage values to those described in each property, we put lower and upper bounds on the corresponding values of the DNN's inputs, to bound them within the range that matches the situation described in the property.

We verified the above properties using the safety property encoding discussed in Section 3. Our results showed that while property A2 held, properties A1, B1 and B2 did not hold for the evaluated DNN, and Verily provided counter examples for each of them. Interestingly, counter examples exist even when setting $k = 2$, i.e., for paths of length 1 (see Section 3). Further, these verification queries took only a few seconds on average to finish. This is to be expected, as many of the input variables were fixed in each property, leaving a relatively small input space for the verification tool to explore.

4.2 Verifying Adaptive Video Streaming

The Pensieve Video Streamer. We next turn our attention to the Pensieve system, discussed in Section 2.

Verifying Pensieve. We check how Pensieve behaves in situations in which the desirable action is clear: either network conditions are excellent and so the highest bitrate available (HD) should be selected, or, alternatively, the network conditions are extremely poor and so the lowest available bitrate (SD) should be selected. Again, Verily could have also been applied to verify more elaborate safety properties, e.g., that the bitrate is changed to HD/SD within T time steps for some fixed $T > 0$. We verify the following properties:

- **A:** When only a single video chunk is left to play, the client's buffer is quite full ($> X\%$), and all recently downloaded video chunks were in HD and downloaded quickly (specifically, faster than the video segment length, set to be 4s), the DNN should output HD.
- **B:** When the client's buffer is almost empty ($<$ 4s) and all recently downloaded video chunks were in SD and downloaded slowly ($>$ 4s), the DNN should output SD.

In order to encode these properties in Verily, we constructed verification queries that restricted the DNN's inputs in the following way. The inputs describing the last bit rate (e.g., HD or SD) and the number of remaining chunks were fixed according to the property at hand. Another input, which indicates the current buffer size (in seconds), was bounded from above and from below — again, according to the property being tested. The remaining inputs, i.e. those describing the throughput history and download time history of the network, proved more difficult to properly encode. In real scenarios, we expect these two inputs to be correlated: their product should equal the size of the block that was downloaded at each point in history. This kind of constraint, which is non-linear, was not directly supported by our underlying verification tool. In order to circumvent this issue, while still restricting our verification queries to realistic scenarios, we used the following strategy: we ran 160 different verification queries, each time fixing the values of the download time history and bounding the throughput history values to a range that was realistic for the particular (fixed) download time history values. While this approach did not consider all possible download time values, it covered sufficiently many options to discover many counter examples.

Each individual verification query took an average of 40 seconds to solve, leading to a total run time of 100 minutes for running all 160 queries for each of the two properties. Verily concluded, again using the safety property encoding in Section 3, that both properties were violated, and provided a counter example of length $k = 2$ for each. In fact, in 55 of the counter examples discovered for property A, not only is HD not selected, but SD is selected instead.

4.3 Verifying Internet Congestion Control

The Custard Congestion Controller. Custard [12] is a deep-RL-based congestion controller. Custard's DNN receives as input a bounded length history of (1) observations about past network conditions, including throughputs, loss rates, and changes in latency; (2) previous sending rates; and (3) previous rewards. The DNN's outputs indicate requested changes to the current sending rate.

Verifying Custard. Custard's underlying DNN was originally designed to use hyperbolic tangent activation functions. These functions, which are not piecewise-linear, are not currently supported by Verily's underlying verification engine. To resolve this, we trained a similar neural network, in which the hyperbolic tangents were replaced with (piecewise-linear) rectified linear unit functions, and used it for our verification queries. Our trained DNN achieved reward values that were similar to those obtained by the original DNN.

We consider the simple scenario in which a single Custard-controlled transport-layer connection is the only sender on a link. We ask whether, in this scenario, Custard can get stuck *indefinitely* at a rate that is significantly lower than the link's bandwidth, thus failing to efficiently utilize the link.

More specifically, we wish to refute the existence of an input state (bounded-length history) x, such that when presented with input x Custard decides not to change its rate, and then observes the exact same input state x (and so forth). Specifically, state x is such that (1) the sending rate has remained fixed throughout the observed history; (2) the received rate (the rate of packet acknowledgment receipt) has also remained fixed and matches the sending rate; (3) no loss was experienced and network latency has remained constant throughout observed history; and (4) the DNN's output implies that the sending rate should not be changed. Observe that if there are no other senders on a link, and if the (constant) sending rate in x is lower than the link's bandwidth B, (1)-(4) above imply that $x \in \mathcal{E}(x)$ since the sending rate remains unchanged. Thus, the existence of a state x as described above implies that Custard might, after deciding to not change its rate at x, observe the same state again, reach the same decision, and so on, without ever utilizing the (potentially much higher) full available bandwidth.

Encoding this property in Verily was straightforward. For each input that represents a series of observations (a history), we stipulated that all observations are equal to each other to capture the fact that the state is stable. The inputs describing the sending and received rates were set to be equal to each other, and could take on any value between 0 and 10^9 (megabits per second). The input representing past loss rate was set to 0, and the one representing past latency could take on any value between 0 and 100 (milliseconds). The last input, which represents the reward function, was calculated according to the latency and received rates values.

We utilized Verily to verify the liveness property capturing the situation described above (with $k = 2$). Verily concluded that no such state x existed, implying the system's correctness. Allowing the received rates and the past sending rates to differ slightly from each other also did not yield counter examples. Each query took approximately 1 minute to run. We leave the investigation of more complex specifications, and of larger values of k, to future work.

5 RELATED WORK

Our approach for verification of deep-RL-driven systems relies on a DNN verification engine as a black box. Verily uses the SMT-based Marabou tool [15], but other verification tools could also potentially be used in its stead, including abstract-interpretation-based engines [7], symbolic interval reasoners [34], or LP-based tools [32] (see [21] for a recent survey). As verification technology progresses, and such verification engines become more scalable, the scalability of our approach to deep-RL verification will also improve.

Approaches for verifying the correctness of *non-deep* RL-driven systems include policy extraction, where instead of the DNN component, an explainable controller such as decision trees [1, 6] is utilized.

Recent studies have addressed the topic of verifying hybrid systems with DNN controllers [5, 30]. The case of hybrid systems is similar to ours in the sense that the neural network controller is evaluated repeatedly, and past evaluations may affect future evaluations. However, much of the focus in that line of work is on handling the continuous nature of the hybrid systems.

6 DISCUSSION

Lessons for deep-RL-based system design. Our experimentation with Verily taught us important lessons regarding the design of deep-RL-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 network correctness, and also for its amenability to verification. For example, in the DeepRM case, the DNN receives as input variables that seemingly have no significance for decision making (e.g., are used solely for system visualization). However, our results reveal that certain value assignments for these input variables may trigger undesirable decision making in the DNN, even if all other input variables remain unchanged. It is also beneficial for the verification process if the input values are continuous, as opposed to discrete; and also if there are no tightly coupled (i.e., redundant) inputs to the network. This tends to render verification computationally easier, due to limitations inherent in existing verification technology.

Another design decision that affects amenability to verification is the choice of neural network architecture, and, specifically, the choice of activation functions. Many verification engines, including Verily's underlying engine Marabou, only support certain activation functions. A choice of piecewise linear activation functions, which are supported by many engines, will make the resulting system easier to verify (as discussed in the context of Custard in Section 4).

Ongoing and future research. We intend to extend Verily by incorporating *invariant inference* techniques [27] into our scheme. This will enable verifying safety and liveness properties without limiting the length of counterexamples to a small k, thus obtaining stronger guarantees. Other natural directions for further research are improving Verily by integrating additional underlying DNN verification engines, and applying Verily to verify additional properties and in additional case studies.

7 CONCLUSION

Deep RL holds great promise for system design, but also gives rise to new challenges. Due to the opacity of DNN-guided decision making, deep-RL-driven systems may exhibit undesirable behaviors that are hard for humans to detect beforehand. This risk must be mitigated before deep RL is built into real-world systems. As DNN verification techniques continue to evolve, we expect that they will play a key role in certifying deep-RL-driven systems. We view Verily as an exciting first step in this direction.

Acknowledgments. The project was partially supported by grants from the Binational Science Foundation (2017662), the Israel Science Foundation (683/18), Intel Corporation, and the National Science Foundation (1814369). The work of the 4th author was supported by an ERC starting grant and by the Israel Science Foundation.

REFERENCES

[1] O. Bastani, Y. Pu, and A. Solar-Lezama. 2018. Verifiable Reinforcement Learning via Policy Extraction. In *Proc. 32nd Conf. on Neural Information Processing Systems (NeurIPS)*. 2494–2504.
[2] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. 1999. Symbolic Model Checking without BDDs . In *Proc. 5th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 193–207.
[3] N. Carlini, G. Katz, C. Barrett, and D. Dill. 2017. Provably Minimally-Distorted Adversarial Examples. Technical Report. http://arxiv.org/abs/1709.10207.
[4] A. Choi, W. Shi, A. Shih, and A. Darwiche. 2019. Compiling Neural Networks into Tractable Boolean Circuits. In *Proc. AAAI Spring Symposium on Verification of Neural Networks (VNN)*.
[5] S. Dutta, X. Chen, and S. 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)*.
[6] D. Ernst, P. Geurts, and L. Wehenkel. 2015. Tree-Based Batch Mode Reinforcement Learning. *J. Machine Learning Research* 6 (2015), 503–556.
[7] T. Gehr, M. Mirman, D. Drachsler-Cohen, E. Tsankov, S. Chaudhuri, and M. Vechev. 2018. AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation. In *Proc. 39th IEEE Symposium on Security and Privacy (S&P)*.
[8] D. Gopinath, G. Katz, C. Păsăreanu, and C. 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.
[9] A. Haj-Ali, Q. Huang, W. Moses, J. Xiang, I. Stoica, K. Asanovic, and J. Wawrzynek. 2019. AutoPhase: Compiler Phase-Ordering for HLS with Deep Reinforcement Learning. Technical Report. http://arxiv.org/abs/1901.04615.
[10] G. Hinton, L. Deng, D. Yu, G. Dahl, A. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. Sainath, and B. Kingsbury. 2012. Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups. *IEEE Signal Processing Magazine* 29, 6 (2012), 82–97.
[11] X. Huang, M. Kwiatkowska, S. Wang, and M. Wu. 2017. Safety Verification of Deep Neural Networks. In *Proc. 29th Int. Conf. on Computer Aided Verification (CAV)*. 3–29.
[12] N. Jay, N. Rotman, P. Brighten Godfrey, M. Schapira, and A. Tamar. 2018. Internet Congestion Control via Deep Reinforcement Learning. arXiv:1810.03259
[13] G. Katz, C. Barrett, D. Dill, K. Julian, and M. Kochenderfer. 2017. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In *Proc. 29th Int. Conf. on Computer Aided Verification (CAV)*. 97–117.
[14] G. Katz, C. Barrett, D. Dill, K. Julian, and M. Kochenderfer. 2017. Towards Proving the Adversarial Robustness of Deep Neural Networks. In *Proc. 1st Workshop on Formal Verification of Autonomous Vehicles (FVAV)*. 19–26.
[15] 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. 2019. The Marabou

Framework for Verification and Analysis of Deep Neural Networks. In *Proc. 31st Int. Conf. on Computer Aided Verification (CAV)*.

[16] S. Krishnan, Z. Yang, K. Goldberg, J. Hellerstein, and I. Stoica. 2018. Learning to Optimize Join Queries With Deep Reinforcement Learning. Technical Report. http://arxiv.org/abs/1808.03196.

[17] A. Krizhevsky, I. Sutskever, and G. Hinton. 2012. Imagenet Classification with Deep Convolutional Neural Networks. *Advances in Neural Information Processing Systems* (2012), 1097–1105.

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

[19] Y. LeCun, Y. Bengio, and G. Hinton. 2015. Deep Learning. *Nature* 521, 7553 (2015), 436–445.

[20] E. Liang, H. Zhu, X. Jin, and I. Stoica. 2019. Neural Packet Classification. *CoRR* abs/1902.10319 (2019). arXiv:1902.10319 http://arxiv.org/abs/1902.10319

[21] C. Liu, T. Arnon, C. Lazarus, C. Barrett, and M. Kochenderfer. 2019. Algorithms for Verifying Deep Neural Networks. Technical Report. http://arxiv.org/abs/1903.06758.

[22] H. Mao, M. Alizadeh, I. Menache, and S. Kandula. 2016. DeepRM. Code Repository. https://github.com/hongzimao/deeprm.

[23] H. Mao, M. Alizadeh, I. Menache, and S. Kandula. 2016. Resource Management with Deep Reinforcement Learning. In *Proc. 15th ACM Workshop on Hot Topics in Networks (HotNets)*. 50–56.

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

[25] H. Mao, R. Netravali, and M Alizadeh. 2017. Pensieve. Code Repository. https://github.com/hongzimao/pensieve.

[26] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz. 2015. Trust Region Policy Optimization. In *International Conference on Machine Learning*. 1889–1897.

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

[28] D. Silver, A. Huang, C. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, and S. Dieleman. 2016. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature* 529, 7587 (2016), 484–489.

[29] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, Graepel T., and D. Hassabis. 2017. Mastering the Game of Go without Human Knowledge. *Nature* 550, 7676 (2017), 354.

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

[31] R. Sutton and A. Barto. 1998. *Introduction to Reinforcement Learning*. MIT press Cambridge.

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

[33] A. Valadarsky, M. Schapira, D. Shahaf, and A. Tamar. 2017. Learning to Route. In *Proc. 16th ACM Workshop on Hot Topics in Networks (HotNets)*. 185–191.

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