Analyzing Learning-Based Networked Systems with Formal Verification

Arnaud Dethise
KAUST

Marco Canini
KAUST

Nina Narodytska
VMware

***Abstract*—As more applications of (deep) neural networks emerge in the computer networking domain, the correctness and predictability of a neural agent's behavior for corner case inputs are becoming crucial. Enabling the formal analysis of agents with nontrivial properties, we bridge between specifying intended high-level behavior and expressing low-level statements directly encoded into an efficient verification framework. Our results support that within minutes, one can establish the resilience of a neural network to adversarial attacks on its inputs, as well as formally prove properties that were previously relying on educated guesses. Finally, we also show how formal verification can help create an accurate visual representation of an agent behavior to perform visual inspection and improve its trustworthiness.**

***Index Terms*—Machine Learning, Formal Verification**

I. Introduction

Neural control agents – a class of decision-making systems enabled by learned policies in the form of (deep) neural networks (NNs) – can achieve impressive performance results in solving difficult problems such as congestion control [15], cache management [19], video streaming [23], and datacenter flow scheduling [2]. Many of these networking applications do not have tractable optimal solutions, and the application of machine learning (ML) techniques to optimize performance based on learning can significantly outperform traditional approaches.

As applications of neural control agents continue to emerge in networking problems, the correctness and predictability of learning-based networked systems for corner case inputs are of crucial importance; in particular, when systems taking incorrect or unexpected decisions can hamper performance or worse, cause failures (e.g., incorrect routing, throughput collapse).

However, a challenge in applying ML in networking applications is that a NN is like a black box: it takes an input and produces an output, but does not offer any insight into why that output was chosen over other possible choices. This causes a lack of assurances and trust about system behavior, which could respond in unexpected or incorrect ways [9], [34], and overall hinders the adoption of learning-based solutions.

Despite longstanding interest in the rigorous verification of NNs, only the recent developments of automated reasoning tools demonstrated practical verification of certain properties,[1] such as robustness (to input perturbations), safety, and equivalence [20], [21]. These advances are significant not only because automated reasoning techniques offer formal guarantees about NNs' behavior that enhance the overall trustworthiness of learning-based systems but especially because the problem of scaling verification to NNs of relevant sizes has been a significant challenge. Although this challenge is not entirely overcome, recent results have made NN verification practical for the kind of problems that we consider.

We focus on the problem of analyzing neural control agents of learning-based networked systems through the muscles of NN verification. As we elaborate later, formally verifying control agents is a larger problem than NN verification because agents operate in a dynamic environment wherein the system state is influenced by external and uncontrolled factors, whereas pure NN verification is concerned with reasoning directly about the intrinsic mapping of inputs to outputs of the function modeled with a NN. This form of reasoning is concerned with low-level aspects such as the structure of activation functions, hidden layers, weights and biases that are far removed from the high-level specifications agents are intended to satisfy [38].

We view our work as a first step towards an efficient verification of learning-based networked systems, with a focus on practical, domain-specific guarantees that are critical to network engineers and operators. In particular, we establish a framework to explore and verify *domain-specific properties of neural control agents*.

Our main contributions are (i) the application of formal verification techniques to neural control agents in the networking context, (ii) establishing sound and complete proofs of interesting properties and (iii) a tool enabling network operators (without knowledge of formal verification) to verify neural agents based on high-level properties of interest.[2] In particular, we devise properties for two case study applications and show how to encode them through novel operators using a verification framework based on Mixed Integer Linear Programming (MILP), demonstrating the generality of formal verification for neural control agents. We also show how to verify and explore domain-based behaviors by developing a specific encoding that establishes formal guarantees of the agent against adversarial attacks or unreliable inputs. Finally, we illustrate how to use formal verification techniques to provide a visual representation of the agent behavior.

To demonstrate our framework, we provide an in-depth case study for Pensieve [23], an adaptive video bitrate application. Our results illustrate various properties that can be verified using simple building blocks and automated encoding within

[1] A property can be formulated as a statement that if the input belongs to some set $\mathcal{X}$, then the output will belong to some set $\mathcal{Y}$, where $\mathcal{X}$ and $\mathcal{Y}$ are domain-specific variables of the system.

[2] Available at https://github.com/sands-lab/dnn-verification-infocom2021.

seconds. We also verify a previous claim (that some output was rarely selected [4], [24]) and prove a stronger claim – that no input maps to that specific output being the most likely to be selected. Further, we extend our work to RL-Cache, a learned caching policy for CDNs [19], demonstrating the approach generality to a complex case involving inputs transformation.

II. Preliminaries

After introducing neural control agents, we give background on NN verification and MILP and discuss how property-based verification works as a foundation for proving correctness.

Control agents: Our focus is on a subclass of machine learning systems that we refer to as neural control agents. This class of systems considers an ML-based *agent* interacting with an *environment*. At each decision step t, the agent selects an action a_t based on the current state of the system s_t and agent's policy π; taking action a_t influences the system and leads to a new state. Generally, the system state is affected by both the agent's decision and external, uncontrolled factors of the environment that the agent interacts with. In turn, this creates uncertainty regarding the optimal agent decision.

The agent takes decisions based on a neural model of the system parameterized as θ. Thus, we describe the policy π of the agent as a function $\pi_\theta(s_t) = a_t$ that captures the long-term goal of maximizing the objective. Depending on the system, the action space of the policy can be discrete (e.g., select one out of a set of available bit-rates as in Pensieve) or continuous (e.g., allocate a share of bandwidth to a flow).

In this paper, we assume that the neural model is a ReLU-based feedforward NN. To express meaningful verification properties, we assume that the state s_t observed by the agent is made of *intelligible features*. We find that these assumptions hold for a range of recent NN applications in networking such as Pensieve [23], RL-Cache [19], and AuTO [2]. We leave it to future work to consider different kinds of ML systems.

A common approach to train control agents is Reinforcement Learning (RL). In RL, the agent receives a reward r_t for each decision a_t it takes. The objective of the agent is to find the policy that maximizes the *discounted sum* of expected future rewards: $\mathbb{E}_{\pi_\theta}[\sum_{t=0}^{\infty} \gamma^t r_t]$ where $\gamma \in [0, 1)$ is the discount factor. In some cases, such as with Pensieve, the RL model is trained using a simulator of the environment by replaying traces; Pensieve replays traces that contain available bandwidth along a network path and the simulator mimics streaming video at different bit rates. Our focus is orthogonal to how the neural agent is trained.

Review of mixed integer linear programming: MILP solves linear problems over a set of integer and real-valued variables. MILP contains a set of decision variables, a set of linear constraints over these variables and an objective function to be optimized (minimized or maximized) that is linear in decision variables. Without loss of generality, we consider a minimization formulation of a MILP. Let $x_1, \ldots, x_n$ be a set of decision variables, a mixed integer linear program can be written as

$$\min \sum_i c_i x_i \quad \text{subject to } \sum_i a_{ij} x_i \geq b_j, j \in [1, m] \\ \text{and } x_i \in \mathbb{Z}, i \in \mathcal{I}_1, \\ \text{and } x_i \in \mathbb{R}, i \in \mathcal{I}_2,$$

where $\mathcal{I}_1$ is a set of indices of integer variables and $\mathcal{I}_2$ is a set of indices of real variables, $\mathcal{I}_1 \cup \mathcal{I}_2 = [1, n]$. There exist very efficient general-purpose MILP solvers, like CPLEX [14], Gurobi [13] and SCIP [11]. Generic MILP solvers implement efficient solving algorithms up to a finite precision. For example, we used CPLEX with a precision of 10^{-8}. This order of error is negligible, so we emphasize that those solvers are essentially sound and complete for the purpose of this work. MILP-based solutions have been used extensively in formal verification of neural networks [6], [7], as well as purpose-built MILP-based solvers like MIPVerify [30] and Sherlock [5].

MILP encoding of NN and verification: We use a recently proposed MILP encoding [8] to encode a ReLU-based NN architecture to MILP. A neural network can be seen as a block-wise structure, where blocks are assembled sequentially to form the network architecture. We assume that variables $x = \{x_1, \ldots, x_n\}, x \in \mathbb{R}^n$ represent inputs of the network, i.e., inputs of the first layer, and $y = \{y_1, \ldots, y_m\}, y \in \mathbb{R}^m$ represent outputs of a network, i.e., outputs of the last layer. Hence, a NN is a mapping $\mathbb{R}^n \to \mathbb{R}^m$.

To define the property verification problem, we need to define restrictions on the inputs and outputs of a network. These conditions are usually domain-specific and come from the designer of the system. Let $pre(x)$ be a MILP formula that defines pre-conditions on the inputs. Similarly, let $post(x)$ be a MILP formula that defines post-conditions on the outputs. Given conditions pre and $post$, a property holds for a given NN if the following condition holds for all x and y: $(pre(x) \wedge y = NN(x)) \implies post(y)$. To verify the property using a MILP solver, we need to check whether a counterexample x' exists that satisfies the following constraint: $pre(x') \wedge (y = NN(x')) \wedge \neg post(y)$. If a counterexample x' does not exist, we conclude that the property holds for a given NN, otherwise we have a witness input that causes a property to fail.

Properties and correctness: The notion of correct agent behavior is domain- and application-specific. It is commonly assumed that the intended agent behavior is formalized as a specification. What is interesting about neural control agents, however, is that it is hard to bridge between a high-level notion of correctness and the low-level artifacts like learned parameters or activation functions that are embedded into the neural policy. As it is typically done in this area, we assume that there are properties that must hold at all times. In particular, we verify properties that can be expressed with linear conditions on the input and outputs of the agent. A simple example property is as follows: $x_1 \in [a_1, b_1] \wedge x_2 \in [a_2, b_2] \Rightarrow y_1 > k$. That is, when inputs x_1, x_2 are within some predefined range, then the output is above a threshold k.

III. Case study: Pensieve

Pensieve [23] trains RL agents that adapt the bit rate selection during video streaming over the Internet, leading to superior user experience. In video streaming environments, a server stores videos divided in *chunks*, which are fixed-length pieces of the video. Each chunk is available in multiple *bit rates*: encoding the chunk with a higher bit rate provides better video quality, but is also larger. The main constraint in streaming environments is the available *network bandwidth*, which limits how fast clients can fetch video chunks.

An *adaptive bit rate* (ABR) algorithm is an algorithm that selects the bit rate encoding for each chunk with the goal of maximizing the user's quality of experience (QoE). Selecting a high bit rate provides a better experience but consumes more bandwidth and might cause playback interruptions when the client buffer drains; conversely, selecting a lower bit rate provides a lower quality but avoids interrupting the video while additional chunks are downloaded.

Pensieve trains a neural control agent that selects bit rates for future video chunks based on observations collected by the client's video player. The agent selects the bit rate of the next chunk among six possible bit rates. The reward function is a widely used QoE metric [36], which increases with higher bit rate chunks (maximize quality) but decreases when there is rebuffering or the quality changes (maximize smoothness).

Because Pensieve does not make assumptions about the environment, it can learn algorithms tailored to different network conditions and different QoE metrics, leading to a 12%-25% increase in average QoE [23].

The reason we select Pensieve as a primary case study is two-fold: (i) its NN contains around 110,000 trainable parameters, which is a significant measure for NN verification, and (ii) there are important stability considerations around the agent behavior. In particular, because video streaming performance is relevant to billions of users worldwide, agents that misbehave may severely affect user experience, cause disproportionate revenue losses, and possibly be leveraged by adversaries for targeted attacks. Moreover, some recent works showed evidence that Pensieve agents exhibit several anomalies [4], [18]. Later, §VII generalizes our work to a second case study: RL-Cache.

Model architecture: Pensieve takes as input 25 features divided into six categories: previous bit rate selected, duration of video in the buffer, number of chunk remaining, measured throughput and download time for the past decisions and the size of the next chunks. Pensieve uses a CNN composed of two hidden layers with ReLU activation. For single-value features, the first layer is fully connected. For multiple-value features, the first layer is a convolutional layer. In both cases, the layer uses 128 units. The outputs of the first layer are concatenated and passed through a fully-connected layer with 128 units. The output layer is composed of a fully-connected layer with 6 outputs and softmax activation. The output is used as a probability distribution over each possible bit rate.

IV. Properties verification framework

We now describe how we encode the NN model and intended agent behavior as a MILP. To make the verification accessible to network operators with little or no knowledge of formal verification, our work is three-fold. We design a set of primitives that are powerful enough to express high-level specifications of the expected behavior, which we expose as building blocks in our verification framework. We also define simple restrictions based on practical domain-specific constraints that are not explicitly known to the ML agent (such as inputs being discrete or limited to a specific range). Finally, we automatically convert the NN model into formal constraints.

The encoding of the agent inside the solver is divided into two main components. The first one is the encoding of a network function. This function is defined by the structure of the NN, including the linear and ReLU operators, and the trained weights and biases of each operation. The encoding of the NN is independent of the encoding of the properties. Constraints encoding will vary between different agents, but is static for any trained agent: one encoding can be used to verify many properties.

The second part of the encoding deals with the verification properties. These properties formalize the intended behavior of the agent, which is expressed in terms of constraints over input and output variables. Recall from Section II that to verify whether a property holds, we solve a MILP that encodes both the network function and the negation of a property. If the MILP solver outputs that the problem has no solutions, then the property is verified.

Properties are usually written by domain experts (in our case, network operators) to encode specific expectations about the agent behavior. While those building blocks can express functions over any kind of inputs, it is usually difficult to express the intended behavior of the agent with unintelligible raw features. For this reason, we assume that all properties will be written over *intelligible* features, which can be directly mapped to concrete information, such as the amount of buffered video data at the client. We believe that this restriction is inherent to the task at hand (verifying high-level properties).

With regard to the encoding, the novelty of our work against prior works on NN verification lies in the observation that intended agent behavior typically is formalized at a higher level of concern than the low-level preconditions and postconditions about the mapping between inputs and outputs, which NN verification directly supports. As such, we devise a set of building-block operators that enable encoding intended agent behavior properties succinctly and naturally.

Network structure: To encode the NN, we define each layer as a set of constraints. We address two kinds of computational layers: fully-connected layers, and convolutional layers. For a fully-connected layer with l inputs and n neurons, suppose we want to encode the relation $y = \text{ReLU}(Ax + b)$. The encoding uses four different sets of variables: the inputs x (size l), outputs y (size n), the ReLU state indicator variable z and auxiliary real variable s (size n). The weights A and

biases b are extracted from the trained model. We can then express the constraints between those variables as:

$$\forall i \in \{1..n\}, \quad y_i = \textstyle\sum_j A_{i,j} x_j + b_i - s_i$$
$$z_i = 1 \Rightarrow s_i = 0$$
$$z_i = 0 \Rightarrow y_i = 0$$

where z_i is a Boolean variable, y_i, s_i are positive real variables.

Our encoding also extends to convolution layers, and is compatible with CNNs. To encode convolution layers with ReLU activations, instead of expressing y_i as a linear product of *all* variables, we restrict them to only the input variables selected in the convolution filter.

The encoding of activation functions requires some consideration since they capture nonlinear relations in the network structure. As described above, the RELU activation function can still be encoded in the MILP solver since it is a piecewise linear function. But, the softmax activation function $\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}}$ – which is commonly used in the network's output layer – is nonlinear. We observe that the softmax is however strictly increasing, meaning that we can use the property $o_i > o_j \Rightarrow \sigma(o_i) > \sigma(o_j)$ to find the highest value. Thus, we encode the nonlinear softmax function by encoding the rank of each neuron's value in that layer and represent it as a linear function.

Domain constraints: NNs usually take unconstrained continuous numerical values as inputs. However, the data representation often defines a larger space than the actual domain of possible values allowed by the application. We represent this discrepancy using a set of specific *domain constraints*. The most common domain constraints are limited range (for example, a value might have been normalized in the $\{0, 1\}$ interval during preprocessing) and discrete inputs. For the definition and encoding of these constraints, see the building blocks defined below.

One important distinction between domain constraints and properties is that, while properties represent the expected behavior of the agent (which we want to verify), domain constraints are inherent properties of the system and a violation is not possible (for example, a video client cannot buffer more data than the maximum buffer capacity, which gives an upper bound on the value of the input variable representing buffer occupancy). As such, we can significantly reduce the search space and speed up verification.

Domain constraints can either be defined manually (restricting the range of input variables) or automatically inferred from a training dataset. If discovered automatically, the following method is used: for each input and output variable in the dataset, collect all existing values; if the number of distinct values is lower than a configurable threshold, define the variable as discrete; otherwise, use the lowest and highest values as lower and upper bounds. We note that some domain constraints might not be discovered using this approach, such as distinct intervals or multivariable constraints. Those can instead be encoded manually as properties.

Properties: The properties to verify are defined over inputs and outputs of the control agent. For example, we can restrict a range of possible values for some inputs, and enforce a constraint that a certain output is expected. We now describe the typical constraints on input and output variables. These constraints are the *building blocks* used to compose the properties we wish to verify.

Constants and ranges. Many properties are defined by expressing conditions for specific values or input ranges. This type of constraint can be trivially encoded using inequality constraints.

Linear relations. This kind of constraint is the basis of MILP verification. We provide a simplified building block to express relations of the form $a_1 x_1 + a_2 x_2 + \cdots \leq b$.

Maximum value. Many agents use the output of the last layer as a weight vector such that the selected decision is the one that has the highest value (or the lowest value, which is an analogous property). To encode them in the solver, we use the operator defined as *"variable x_i is the maximum of $\{x_1, ..., x_n\}$"*. The encoding of this operator uses auxiliary variables $p_1, ..., p_n$ and is encoded as:

$$\forall j \in \{1..n\}, \quad p_j = 1 \Leftrightarrow x_i \geq x_j; \quad \textstyle\sum_j p_j = n$$

Intuitively, we see that if x_i is maximum, then all p_j will be equal to 1 and the second condition is fulfilled. We assume that ties are broken in favor of x_i.

Discrete variable. Control agents frequently take discrete variables as inputs, but the neural network treats all variables as continuous. To ensure that only possible values are selected for these variables, it is required to be able to encode the pre-existing knowledge that a variable can only be taken from a set of known constants.

To encode this information, we use an operator defined as *"variable x must be equal to some value c_i from the set $\{c_1, ..., c_k\}$"*. To do so, we define auxiliary variables $p_1, ..., p_k$ and express the property as:

$$\forall i \in \{1..k\}, \quad p_i = 1 \Leftrightarrow x = c_i; \quad \textstyle\sum_i p_i = 1$$

Intuitively, the first constraint expresses that the indicator variable p_i is equal to 1 if and only if $x = c_i$, while the second constraint ensures that exactly one of the values is selected.

V. IMPLEMENTATION

Our implementation of NN verification uses the CPLEX [14] MILP solver. The encoding of the NN structure is done in Python, by automatically transforming TensorFlow NN models into formal constraints.

For the transformation, we encode the NN inputs as a set of unbounded variables. We then add constraints representing the connections between variables in each layer (each operator in the NN yields a constraint). For example, a fully-connected layer is encoded using the following relation:

$$\forall i : y_i = \sum_j w_{ij} \cdot x_j + b_i$$

To express the RELU activation function, we use the encoding described in Section IV. Since the output layer is

normalized using the nonlinear softmax, it cannot be encoded in the MILP solver; instead, we directly use the value of the node before applying normalization, which nonetheless preserves the ordering of the variables.

To allow for testing of different model versions, the weights and biases are read from static model files. This makes it possible to verify a different model sharing the same structure (but with different weights) by simply replacing the checkpoint file. The trained models used for the evaluation were provided by the authors of their respective papers.

A. Building properties

As discussed in Section IV, properties can be encoded using simple building blocks. We described the MILP encoding for four different predicates, which we express as $constant$ (variable is fixed), $inrange$ (variable is bounded above and below), $inset$ (variable is restricted to a discrete set of possible values) and $maximum$ (variable is/isn't the maximum among other variables).

We now see how these blocks can be used to encode the properties that we verify in Section VI. Note that for all properties, the solver will only return a solution if all constraints are satisfied.

Dataset constraints. Some variables, such as the previously chosen bit rate in Pensieve, can only be selected from a finite set of values. In that case, we can directly use the "discrete variable" predicate to encode the set of possible values as a constraint on the input variables $x_1, ..., x_n$. This ensures that any solution found by the MILP solver only assigns valid values for these inputs.

Additionally, some variables are naturally bounded because of the information they represent. For example, in Pensieve, the network throughput is always positive and the buffer has a maximum size. Because those constraints are inherent and can be checked before observing a decision of the agent, we use the $inrange$ predicate to enforce natural bounds on input variables.

Adversarial perturbations. Given input variables $x_1, \ldots, x_n$ (continuous or discrete) and output variables $y_1, \ldots, y_m$, we take an initial point $P = p_1, ..., p_n$ mapping to output y_P. We suppose that the attacker's ability to change the inputs is asymmetrical, and consider the attack power over variables $x_1, \ldots, x_n$ to be proportional to $k_1, ..., k_n$. An attacker with attack power ϵ can change the value of x_i by at most $\epsilon \cdot k_i$.

To verify that an attacker with attack power ϵ cannot alter the decision, we use the following encoding:

$$\forall i \in \{1, \ldots, n\}, \quad x_i \ inrange(p_i - \epsilon \cdot k_i, p_i + \epsilon \cdot k_i)$$
$$y_P \neq maximum(y_1, \ldots, y_m)$$

By showing that no solution satisfies the constraints, we prove that an attacker with power ϵ is unable to change the decision.

To find the minimum value of ϵ, we perform an iterative search for possible values of ϵ with precision 0.001. Note that the run times reported in the next section are for the complete verification (including multiple calls to the solver).

Missing features. Given input variables $x_1, \ldots, x_n$ (continuous or discrete) and output variables $y_1, \ldots, y_m$, we take an initial point $P = p_1, \ldots, p_n$ mapping to output y_P. We suppose that a subset L of $x_1, \ldots, x_n$ is lost (all variables in L take arbitrary values within their valid ranges).

To verify that the decision does not change despite the missing inputs, we use the following encoding:

$$\forall x_i \in \{x_1, \ldots, x_n\} \setminus L, \quad x_i = constant(p_i)$$
$$y_P \neq maximum(y_1, \ldots, y_m)$$

If any solution satisfying those constraints is found, then the agent is not resilient to the loss of the features in L (it can lead to a different decision).

Decision boundaries. Given input variables $x_1, \ldots, x_n$ (continuous or discrete) and output variables $y_1, \ldots, y_m$, we want to display the decision boundaries for input variables x_a and x_b such that $l_a \leq x_a \leq u_a$ and $l_b \leq x_b \leq u_b$. Other variables are fixed to constant values $\{p_1, \ldots, p_n\} \setminus \{p_a, p_b\}$, and we want to draw the area such that the highest output is y_P.

The problem is thus encoded as:

$$\forall i \in \{1, \ldots, n\} \setminus \{a, b\}, \quad x_i = constant(p_i)$$
$$x_a \ inrange(l_a, u_a)$$
$$x_b \ inrange(l_b, u_b)$$
$$y_P = maximum(y_1, \ldots, y_m)$$

Each solution returned by the solver maps to one set of ReLU states in the neural network. By rewriting the neural network as a linear equation, we can then find the corresponding bounds on x_a and x_b. In Figures 1a and 1b, each area corresponds to one solution satisfying the above constraints.

VI. Results

We present several questions that can be answered by using the formal encoding of the Pensieve agent. These questions answer practical concerns of network operators which reflect possible roadblocks to the deployment of Pensieve in real environments and provide insightful information about the behavior of the model. While being specific to this application, the questions are a case in point illustrating that our approach enables answering formally regarding two important aspects of robustly trained agents: exact guarantees about the robustness of the model and accurate understanding of the model behavior over continuous inputs. Therefore we expect our approach to generalize to other applications and we illustrate one such extension later in Section VII.

The results below use different timeouts to cap verification time (mentioned individually for each block of results). While most of the properties are verified within 20 seconds, we note that in some cases we reach the timeout, whereas higher timeout values might have produced results. However, we set a limit to express the trade-off between available resources and accuracy. In practice, the timeout can be adjusted to fit different resources and needs.

	Case 1	Case 2
All inputs	0.081	0.042
Previous bit rate	0.331	0.056
Throughput, latency	0.200	0.062
Buffer, chunk sizes, remaining chunks	0.329	0.135
Buffer	0.335	0.140
All throughput	0.288	0.085
Throughput, latency (most recent value only)	0.342	0.104

TABLE I: Attacker power (ϵ) required to alter the decision.

A. Resilience to adversarial perturbations

A common goal of a neural agent like Pensieve is to direct the system in an uncontrolled environment. An adversary could gain partial control over that environment, which leads to variations in the inputs.

Thanks to formal verification, we can ensure that adversarial perturbations cannot cause a specific decision to cross the decision boundary and cause a different output of the NN. We also show how to find the minimum degree of control that an adversary should have over a particular set of inputs to cause an erroneous decision.

We evaluate the resilience of the Pensieve agent against adversarial perturbation by comparing attackers that alter different subset of the inputs in Table I. Given an attacker that can alter each input feature x_i by a factor no more than $\epsilon \cdot k_i$, we report the minimum value of ϵ such that the attacker can change the decision. For all parameters, we set the value $k_i = upper_i - lower_i$, where $upper_i$ and $lower_i$ are the upper and lower bound on possible values for x_i.

Because the attacker power is measured around some specific input, we cover two cases. Case 1 represents a fictional execution trace where the throughput measured by the agent is always constant and the buffer data is equal to the average value across the testing dataset. In this scenario, the expected behavior of the agent is known but must be verified, i.e., we want to ensure that an attacker could not affect the output with limited power. Case 2 represents a realistic case from the testing dataset that is observed to create uncertainty in the decision (the agent gives high probability to multiple bit rates). In this scenario, the exact behavior of the agent is uncertain, and the verification provides additional information about the behavior of the agent to understand its decision. The results are shown in Table I; Table II reports corresponding run times.

We observe that in case 1 (where the expected agent behavior is known), an adversary needs a strong power to be able to change the decision (ϵ is large). One interpretation is that the agent is confident in its decision, and only large changes in the inputs will cause a different decision. On the other hand, in case 2, the required power is much smaller (ϵ is lower than in case 1), showing that it is easy to change the agent decision with small changes. Those conclusions hold as long as the attacker's control over each variable $\epsilon \cdot k_i$ is roughly proportional to the range of the variable values.

Additionally, we also observe that different input features have different impact on the decision. For example, being able to change all measurements of throughput and latency makes it possible to change the decision with limited power, while having control only on the amount of buffered data requires higher attack power. As expected, being able to change all inputs at the same time requires a smaller change to the input values than controlling only part of the data.

	Case 1	Case 2
All inputs	21.23	16.36
Previous bit rate*	47.47	22.47
Throughput, latency	551.40	31.82
Buffer, chunk sizes, remaining chunks	92.80	51.94
Buffer	114.30	72.70
All throughput	134.90	73.46
Throughput**, latency**	163.30	85.16

TABLE II: Run times (in seconds) of Table I results.

	Case 1	Case 2
Throughput	False	False
Latency	Timeout	False
Latency (4 values)	True	False
Throughput (1 value)	True	False
Chunk size	True	True
Previous bit rate	False	False

TABLE III: Robustness to missing features. Cell indicates whether the agent is stable when the listed feature is missing.

One thing to note is that in the case of the previous bit rate, which is a discrete input, the required power corresponds to the power required to change to the next discrete value. In case 1, the attacker needs to change the input from 1850Kbps to 300Kbps (three steps), while in case 2, it changes from 1850Kbps to 1200Kbps (one step).

B. Robustness against missing features

In the case of learning-based networked systems, the inputs are often based on incomplete knowledge about the system state. For example, a congestion control agent chooses the best sending rate using a local estimate of the network utilization and available bandwidth. In some cases, this presents a risk that the input values are incorrect, either because the local information does not reflect the actual state of the network or because measurements are inaccurate. Due to the difficulty (or impossibility) of perfect system state knowledge, it is often necessary to ensure that, even under incorrect information, the agent maintains a robust policy.

Our approach allows operators to verify if erroneous input values would lead to incorrect behavior. We define erroneous inputs as one (or several) of the inputs taking arbitrary values independent of the actual state of the environment, and incorrect behavior as changing the decision taken by the agent when every other input value remains identical to the original.

To illustrate this property, we verify whether the Pensieve agent can maintain the same decision when some of its input features receive arbitrary values. Table III reports whether the agent decision changes by altering the value of a subset of all input features (run times in Table IV). We test the agent in two different cases, as previously described in Section VI-A.

We observe that in both cases, incorrect values for the past throughput or the previous decision can lead to a change in the decision. In other words, those values are always significant

	Case 1	Case 2
Throughput	7.74	7.91
Latency	Timeout	12.34
Latency (4 values)*	7.35	8.62
Throughput (1 value)*	7.37	7.71
Chunk size	7.26	7.63
Previous bit rate	7.46	7.87

TABLE IV: Run times (in seconds) of Table III results.

to the decision. We note that Pensieve has compound values among its inputs, as the throughput, latency, and previous bit rate are co-dependent.

On the other hand, arbitrary changes to the chunk size will never alter the decision. For example, if the inputs showed that the next chunk encoded at 750Kbps is as large as the one encoded at 4300Kbps, it would not affect the decision of the agent in either case.

Regarding the latency, we were not able to make any conclusion for the first case before reaching the timeout (set to one hour). This means that we did not find any inputs for which the decision changed, but were not able to cover all possible inputs. By reducing the search space to only the four most recent latency measurements, it was possible to confirm that changes in the input value of these features would never lead to a different decision.

C. Decision boundaries

Some of the input features are continuous both in Pensieve as well as in many other applications. However, minor changes in the inputs should not lead to drastically different outputs. This means that we can define contiguous areas of the input space in which all inputs lead to the same decision.

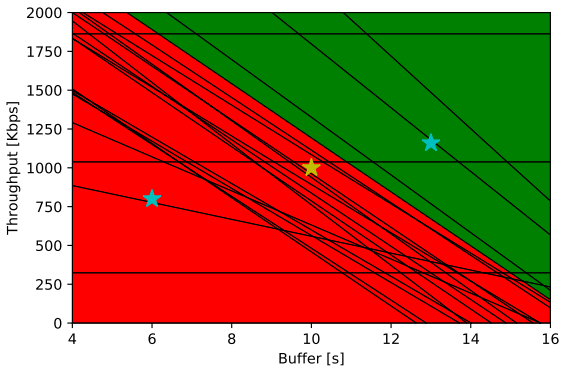
In turn, this means that the bounds of the area represent *decision boundaries*, which connect two different decision areas. For example, in the case of Pensieve, we can find a decision boundary in the input space where the decision shifts from a low-quality bit rate to a higher quality.

While the decision boundaries are very important to understand the behavior of an agent, there is no straightforward way to extract them from a NN. The main obstacle is that extracting the decision boundary requires handling an exponential number of possible values due to the nonlinear ReLU activations.

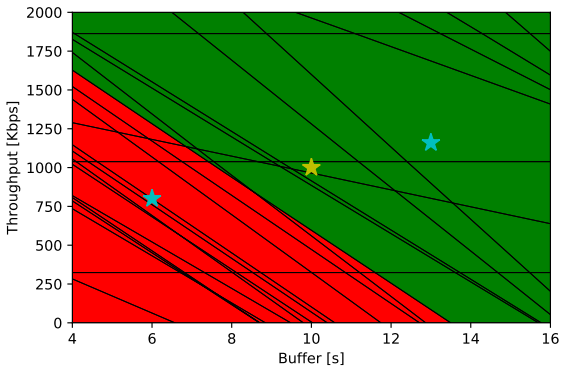
We solve this problem by restricting the output space only to *possible* representations while looking at pairs of input variables. This approach allows us to show the decision map of the agent in a 2D space corresponding to the chosen variables by extracting the linear function from the network for a given system state.

To illustrate the benefits of visual inspection, we provide an example with the Pensieve agent by displaying the output of the neural network for different values of the buffer and throughput. Figures 1a and 1b show the output boundaries when the previously selected bit rate is 300Kbps and 750Kbps, respectively. Other inputs are set to the average values in the dataset for the corresponding decision.

Visual inspection shows that there is a region between the 300Kbps and 750Kbps decisions where the agent will maintain the previous decision (assuming an average buffer value). For



(a) Previous selected bit rate is 300Kbps



(b) Previous selected bit rate is 750Kbps

Fig. 1: Pensieve decision boundaries. Red: 300Kbps; green: 750Kbps.

example, for the input highlighted in yellow, the agent will decide based on the previous decision, maintaining smoothness in video playback. This behavior is expected to avoid rapid changes between the bit rates when a decision is taken close to the boundary.

For other values, such as the two points highlighted in cyan, the agent will *always* decide to select the same bit rate. For example, if the buffer contains only 6 seconds of video and the network throughput is less than 1250Kbps, the agent will always select the 300Kbps bit rate. This behavior will cause the buffer to fill, preventing rebuffering in the future.

Visual inspection allows operators to assert that the region provides a sufficient transition space to ensure that the decisions are in line with the expected behavior, as well as understand in which regions the agent decides to take a safe behavior (filling the buffer) rather than switching to a higher bit rate.

D. Verifying specific properties

In some cases, possible pitfalls in a particular environment are known to domain experts. Thus, it is important to verify that the neural policy correctly avoids known pitfalls.

For example, in the case of Pensieve, it was observed in [4], [24] that the agent would usually not select some available bit rates even when the condition seemed optimal. However, the claim was based only on observations over a limited set

of traces and does not provide a complete verification or the conditions under which those bit rates would be selected.

We tried to verify that claim in the case of one of the two bit rates, 1200Kbps, which is *almost never* selected. We configured our tool to verify inputs that would be ideal to select that decision (i.e., the throughput measured is constant at 1200Kbps, the previously selected bit rate was identical, and the buffer value is not especially high or low). The property that we encoded was to return all possible inputs in which the target decision has a higher probability than any other one.

Our result shows that, in reality, the 1200Kbps bit rate is *never* the one with the highest probability. While the agent will select that bit rate occasionally, it is only because the decision is ultimately randomized over weighted probabilities.

Additionally, we also relaxed as many constraints as we could to find the minimum set of constraints under which we could safely conclude that this bit rate would never be selected. In those scenarios, our tools proves that the target bit rate will never be selected if the measured network throughput over the four most recent decisions was 1200Kbps, with every other variable left unbounded.

VII. Extension to complex cases

We now extend our verification approach to more complex neural agents for networking problems, in an attempt to explore the limits of our technique. To this end, we apply property verification to the case of RL-Cache [19], an admission control agent for cache control in CDNs. We test generic, trivial properties such as "find all possible ReLU assignment in the output region" to verify how our verification tool behaves when verifying complex NN structures.

RL-Cache accounts for a large number of features of cached objects, including object size, recency, and frequency of access, to decide whether to cache said object. Unlike traditional approaches to cache control such as LRU or SLRU which typically use only one or two of these features, RL-Cache uses ML to train an algorithm that uses all the information available to make a decision, with the goal of maximizing cache hit rate.

However, because it uses a NN to support its decision, it becomes impossible to determine through manual inspection if the learned policy is sound. For example, one might want to verify whether a small object that has been accessed recently be accepted in the cache over a large stale object. Formal verification of decision properties can answer this question while still maintaining the benefits of the ML approach.

One of the aspects that makes RL-Cache an interesting use case is that its implementation uses input pre-processing and non-ReLU activations, making the encoding challenging. In this section, we describe how we handle those specific aspects of RL-Cache to successfully verify its properties.

A. Non-linear activation functions

Unlike Pensieve, which uses ReLU activation functions, RL-Cache uses the *Exponential Linear Unit* (ELU) activation function [3]. ELU is a variant of ReLU in which the transition between the domains on which the unit is *on* and *off* is smoothed using an exponential function. The ELU activation function is defined as:

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha\ (\exp(x) - 1) & \text{if } x \leq 0 \end{cases}$$

where $\alpha > 0$ is a smoothing parameter. Using ELU activations improves training time by pushing the mean unit activation closer to zero, and improves noise resistance when close to the activation threshold.

However, ELU is not a linear function. Because our property verification framework relies on linear verification, the use of this activation function makes encoding the structure of the neural network more challenging.

To solve this, we use an approximation of the ELU function called SReLU (Shifted ReLU). The SReLU variant is similar to ELU but does not smooth the transition between activated and deactivated states. Figure 2 shows the difference between the ReLU, ELU (with $\alpha = 1$) and SReLU functions.

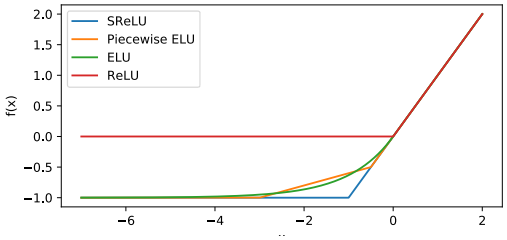


Fig. 2: ReLU, ELU, SReLU and Piecewise ELU.

Because we must use a linear approximation of the true function, we are forced to weaken the guarantees of our verification framework, as the formal encoding of the network can differ from the actual values. However, we observed that in practice, the difference between the actual ELU functions and SReLU representation, in the trained network, is generally negligible and does not change the output.

If higher accuracy is needed, it is possible to replace the use of SReLU by a piecewise linear approximation of ELU, as depicted in Figure 2. In this approach, a trade-off is made by sacrificing the efficiency of the solver to increase the accuracy of the linear model.

B. Non-intelligible inputs transformation

To be able to verify meaningful properties, it is required that inputs be intelligible, i.e., refer to concrete concepts that a human operator can reason about to express the properties.

In the RL-Cache implementation, each input is intelligible (object size, recency and frequency of access, etc). However, to facilitate learning, the inputs are first transformed into *non-intelligible* variants, which are then fed to the NN. Consequently, it is not possible to directly express properties on the input features and to use the verification predicates described in Section V-A.

To achieve verification on transformed inputs, we leverage a property of the transformation that any linear property

expressed on intelligible features, can be verified on a non-intelligible transformation of these features if, given a convex subspace of the input space, the transformation converts it into a finite set of convex subspaces.

Intuitively, this approach consists of rewriting any convex subspace of the intelligible space into a convex subspace of the transformed space. By preserving the convexity, we ensure that any value falling between the bounds in the original space will also fall inside the bounds after transformation.

We convert all properties using the $inrange$ predicate to use the transformed space instead. Similarly, all properties using the $inset$ predicate are converted to the set of transformed values. For properties in which the original convex subspace is transformed into *multiple* subspaces, we split them into separate properties, each independently verified for each transformation.

VIII. Related work

A sound and complete verification framework guarantees that a method either proves that the property holds or finds a counterexample to this property. For example, frameworks like Reluplex [16], Marabou [17], MIPVerify [30] provide complete verification algorithms. These frameworks are based on Satisfiability Modulo Theories (SMT) or/and Mixed Integer Linear Programming (MILP) search engines. The main issue with this approach is scalability. For example, NNs that are used for computer vision tasks contain millions of parameters. Theoretically, large NNs allow us to generate their formal specifications. However, in practice, these formalizations are challenging to reason about for modern solvers.

To counter this challenge, we focus on expressing concrete properties based on domain knowledge that encapsulates desired or unwanted behavior and can be verified in a reasonable time. *A sound and incomplete verification framework* guarantees that it either proves a property or it remains unknown whether the property holds. Examples of such frameworks are FastLin [33], Crown [37], DeepZ [29], etc. The main underlining idea is to perform safe approximate reasoning about the behavior of a neural network. If the approximate reasoning is sufficient to prove a property then incomplete methods succeed otherwise they fail. Another line of work focuses on training robust networks. This line of work focuses on robustness to small input perturbations. This research direction includes techniques like adversarial training [12], [22] and designing certified defenses [26], [27], [35]. The idea of adversarial training is to take stochastic gradient steps at an approximation of worst-case perturbations rather than original inputs. These techniques help to improve robustness empirically, but they cannot provide any guarantees. Certified defense methods compute an upper bound of loss function under the adversarial attack by relaxation of the network function during the training procedure. During the inference, these methods are able to compute a certificate of robustness. However, they might fail to certify robust inputs. [18] presented results on the verification of RL controllers. For example, the authors defined and verified several properties of the Pensieve agent. Our work builds on top of this work, and we perform an in-depth analysis of two different controllers. [10] focuses on robustness properties for Pensieve and proposes a new training procedure to enhance the robustness of the network controller. Finally, our work on finding decision boundaries is based on the reachability analysis work by [31] as we perform an enumeration of feasible ReLU assignments.

A parallel field of work is to explain the neural networks based on post-hoc explanations. Anchors [28] provides explanations of a model's decisions based on local approximations and can be used to predict model behavior. L2X [1] learns a model that provides instance-wise feature selection and can be used to verify some model properties such as ensuring that the model reacts to specific inputs. However, those approaches can only provide a small subset of desirable properties and do not provide any guarantee that their predictions are correct.

A different approach aims to produce interpretable versions of a trained NN by retraining a new, separate agent, using the original model as a training oracle. For example, PIRL [32] uses the oracle to create a program in a high-level, domain-specific language that attempts to minimize the distance to the oracle while being interpretable. Metis [24] makes interpretation of the original model possible by training a decision tree that emulates the oracle, or by generating a hypergraph representation that captures the critical results of the original system. These systems build an interpretable model as a surrogate, which might not capture the original model perfectly (indeed, they generally cannot model the same complexity, because the surrogate model has a smaller number of parameters). Unlike those approaches, we directly verify properties on the true representation of the original network, without retraining a separate model.

DiffAI [25] proposes a solution to provably verify the correct classification of instances via abstract interpretation, which guarantees that inputs will be mapped to the correct output under adversarial noise. However, DiffAI is mainly focused on neighborhood searches and it is not possible to express all the constraints that represent high-level properties we wish to verify with this approach, while we showed that the same properties can be encoded in a MILP solver using a small set of operators.

IX. Conclusion

We are currently at the onset of ML applications in networking. Yet, early learning-based networked systems are producing performance improvements and we expect more such systems will be developed. It is crucial that, together with performance benefits, assurances and other provable properties be established about these systems or else they might not be deployed in practice. We described what to our best knowledge is the first framework that demonstrates how to build on the formal guarantees of NN verification to prove practical properties of interests. We showed two case studies of our framework, establishing for instance the resilience of Pensieve to adversarial inputs within minutes.

REFERENCES

[1] J. Chen, L. Song, M. Wainwright, and M. Jordan. Learning to Explain: An Information-Theoretic Perspective on Model Interpretation. In *International Conference on Machine Learning, ICML*, 2018.

[2] L. Chen, J. Lingys, K. Chen, and F. Liu. AuTO: Scaling deep reinforcement learning for datacenter-scale automatic traffic optimization. In *Conference of the ACM Special Interest Group on Data Communication, SIGCOMM*, 2018.

[3] D.-A. Clevert, T. Unterthiner, and S. Hochreiter. Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs), 2015. arXiv 1511.07289. http://arxiv.org/abs/1511.07289.

[4] A. Dethise, M. Canini, and S. Kandula. Cracking Open the Black Box: What Observations Can Tell Us About Reinforcement Learning Agents. In *Workshop on Network Meets AI and ML, NetAI*, 2019.

[5] S. Dutta, X. Chen, S. Jha, S. Sankaranarayanan, and A. Tiwari. Sherlock - A tool for verification of neural network feedback systems. In *International Conference on Hybrid Systems: Computation and Control, HSCC*, 2019.

[6] S. Dutta, S. Jha, S. Sankaranarayanan, and A. Tiwari. Output Range Analysis for Deep Feedforward Neural Networks. In *NASA Formal Methods Symposium, NFM*, 2018.

[7] R. Ehlers. Formal Verification of Piece-Wise Linear Feed-Forward Neural Networks. In *International Symposium on Automated Technology for Verification and Analysis, ATVA*, 2017.

[8] M. Fischetti and J. Jo. Deep Neural Networks and Mixed Integer Linear Optimization. *Constraints*, 23(3):296–309, 2018.

[9] T. Gilad, N. H. Jay, M. Shnaiderman, B. Godfrey, and M. Schapira. Robustifying Network Protocols with Adversarial Examples. In *Workshop on Hot Topics in Networks, HotNets*, 2019.

[10] T. Gilad, N. H. Jay, M. Shnaiderman, B. Godfrey, and M. Schapira. Robustifying Network Protocols with Adversarial Examples. In *Workshop on Hot Topics in Networks, HotNets*, 2019.

[11] A. Gleixner, M. Bastubbe, L. Eifler, T. Gally, G. Gamrath, R. L. Gottwald, G. Hendel, C. Hojny, T. Koch, M. E. Lübbecke, S. J. Maher, M. Miltenberger, B. Müller, M. E. Pfetsch, C. Puchert, D. Rehfeldt, F. Schlösser, C. Schubert, F. Serrano, Y. Shinano, J. M. Viernickel, M. Walter, F. Wegscheider, J. T. Witt, and J. Witzig. The SCIP Optimization Suite 6.0. Technical report, Optimization Online, 2018.

[12] I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and Harnessing Adversarial Examples. In *International Conference on Learning Representations, ICLR*, 2015.

[13] Gurobi. Gurobi Optimizer Reference Manual, 2019.

[14] IBM. ILOG CPLEX Optimization Studio, 2019.

[15] 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, ICML*, 2019.

[16] G. Katz, C. W. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In *International Conference on Computer Aided Verification, CAV*, 2017.

[17] G. Katz, D. A. Huang, D. Ibeling, K. Julian, C. Lazarus, R. Lim, P. Shah, S. Thakoor, H. Wu, A. Zeljic, D. L. Dill, M. J. Kochenderfer, and C. W. Barrett. The Marabou Framework for Verification and Analysis of Deep Neural Networks. In *International Conference on Computer Aided Verification, CAV*, 2019.

[18] Y. Kazak, C. W. Barrett, G. Katz, and M. Schapira. Verifying Deep-RL-Driven Systems. In *Workshop on Network Meets AI and ML, NetAI*, 2019.

[19] V. Kirilin, A. Sundarrajan, S. Gorinsky, and R. K. Sitaraman. RL-Cache: Learning-based Cache Admission for Content Delivery. *IEEE Journal on Selected Areas in Communications*, 38(10):2372–2385, 2020.

[20] F. Leofante, N. Narodytska, L. Pulina, and A. Tacchella. Automated Verification of Neural Networks: Advances, Challenges and Perspectives, 2018. arXiv 1805.09938. http://arxiv.org/abs/1805.09938.

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

[22] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu. Towards Deep Learning Models Resistant to Adversarial Attacks. In *International Conference on Learning Representations, ICLR*, 2018.

[23] H. Mao, R. Netravali, and M. Alizadeh. Neural Adaptive Video Streaming with Pensieve. In *Conference of the ACM Special Interest Group on Data Communication, SIGCOMM*, 2017.

[24] Z. Meng, M. Wang, J. Bai, M. Xu, H. Mao, and H. Hu. Interpreting Deep Learning-Based Networking Systems. In *Conference of the ACM Special Interest Group on Data Communication, SIGCOMM*, 2020.

[25] M. Mirman, T. Gehr, and M. Vechev. Differentiable Abstract Interpretation for Provably Robust Neural Networks. In *International Conference on Machine Learning, ICML*, 2018.

[26] M. Mirman, T. Gehr, and M. T. Vechev. Differentiable Abstract Interpretation for Provably Robust Neural Networks. In *PInternational Conference on Machine Learning, ICML*, 2018.

[27] A. Raghunathan, J. Steinhardt, and P. Liang. Semidefinite Relaxations for Certifying Robustness to Adversarial Examples. In *Conference on Neural Information Processing Systems, NeurIPS*, 2018.

[28] M. T. Ribeiro, S. Singh, and C. Guestrin. Anchors: High-precision model-agnostic explanations. In *AAAI Conference on Artificial Intelligence*, 2018.

[29] G. Singh, T. Gehr, M. Mirman, M. Püschel, and M. T. Vechev. Fast and Effective Robustness Certification. In *Conference on Neural Information Processing Systems, NeurIPS*, 2018.

[30] V. Tjeng, K. Y. Xiao, and R. Tedrake. Evaluating Robustness of Neural Networks with Mixed Integer Programming. In *International Conference on Learning Representations, ICLR*, 2019.

[31] H. Tran, D. M. Lopez, P. Musau, X. Yang, L. V. Nguyen, W. Xiang, and T. T. Johnson. Star-Based Reachability Analysis of Deep Neural Networks. In *Third World Congress on Formal Methods*, 2019.

[32] A. Verma, V. Murali, R. Singh, P. Kohli, and S. Chaudhuri. Programmatically Interpretable Reinforcement Learning. In *International Conference on Machine Learning, ICML*, 2018.

[33] T. Weng, H. Zhang, H. Chen, Z. Song, C. Hsieh, L. Daniel, D. S. Boning, and I. S. Dhillon. Towards Fast Computation of Certified Robustness for ReLU Networks. In *International Conference on Machine Learning, ICML*, 2018.

[34] C. Wierzynski. The Challenges and Opportunities of Explainable AI, 2018. https://ai.intel.com/the-challenges-and-opportunities-of-explainable-ai/.

[35] E. Wong and J. Z. Kolter. Provable Defenses against Adversarial Examples via the Convex Outer Adversarial Polytope. In *International Conference on Machine Learning, ICML*, 2018.

[36] X. Yin, A. Jindal, V. Sekar, and B. Sinopoli. A Control-Theoretic Approach for Dynamic Adaptive Video Streaming over HTTP. In *Conference of the ACM Special Interest Group on Data Communication, SIGCOMM*, 2015.

[37] H. Zhang, T. Weng, P. Chen, C. Hsieh, and L. Daniel. Efficient Neural Network Robustness Certification with General Activation Functions. In *Conference on Neural Information Processing Systems, NeurIPS*, 2018.

[38] H. Zhu, Z. Xiong, S. Magill, and S. Jagannathan. An Inductive Synthesis Framework for Verifiable Reinforcement Learning. In *Conference on Programming Language Design and Implementation, PLDI*, 2019.