Guarded Deep Learning using Scenario-based Modeling

Guy Katz
The Hebrew University of Jerusalem, Jerusalem, Israel

Keywords: Scenario-based Modeling, Behavioral Programming, Machine Learning, Deep Neural Networks.

Abstract: Deep neural networks (DNNs) are becoming prevalent, often outperforming manually-created systems. Unfortunately, DNN models are opaque to humans, and may behave in unexpected ways when deployed. One approach for allowing safer deployment of DNN models calls for augmenting them with hand-crafted *override rules*, which serve to override decisions made by the DNN model when certain criteria are met. Here, we propose to bring together DNNs and the well-studied *scenario-based modeling* paradigm, by expressing these override rules as simple and intuitive scenarios. This approach can lead to override rules that are comprehensible to humans, but are also sufficiently expressive and powerful to increase the overall safety of the model. We describe how to extend and apply scenario-based modeling to this new setting, and demonstrate our proposed technique on multiple DNN models.

1 INTRODUCTION

Deep machine learning (Goodfellow et al., 2016) is dramatically changing our world, by allowing engineers to create complex models using automated learning algorithms (Gottschlich et al., 2018). These learning algorithms generalize examples of how the desired system should behave into an artifact called a *deep neural network* (*DNN*), capable of correctly handling new inputs — even if it had not encountered them previously. In many instances, DNNs have been shown to *greatly outperform* manually-crafted software. Examples of note include AlphaGO (Silver et al., 2016), which defeated some of the world's strongest human Go players; DNN-based systems for image recognition with super-human precision (Simonyan and Zisserman, 2014); and systems in many other domains such as natural language processing (Collobert et al., 2011), recommender systems (Elkahky et al., 2015) and bioinformatics (Chicco et al., 2014). As DNNs are proving more accurate and easier to create than manually-crafted systems, their use is expected to continue and intensify in the coming decades. Indeed, there is now even a trend of using DNNs in *highly critical systems*, such as autonomous cars and unmanned aircraft (Bojarski et al., 2016; Julian et al., 2016).

DNNs have been demonstrating extraordinary performance, but they also pose new challenges (Amodei et al., 2016). A key difficulty is that DNNs are extremely *opaque*: because they are generated by computers and not by humans, we can empirically see that they perform well, but we do not fully understand their internal decision making. Consequently, it is nearly impossible for humans to reason about the correctness of DNNs. For instance, it has been observed that many state-of-the-art DNNs for image recognition, which at first glance seemed to perform spectacularly, could be fooled by slight perturbations to their inputs (Szegedy et al., 2013). This raises serious concerns about these networks' safety and reliability. Initial attempts are being made to automatically reason about DNNs using formal methods (Katz et al., 2017; Huang et al., 2017; Gehr et al., 2018; Wang et al., 2018; Katz et al., 2019a), but these approaches are currently of limited scalability. Further, these approaches do not specify how to correct an undesirable behavior in a DNN after it has been discovered, which is also a difficult task.

Consider, for example, the case of the DeepRM system (Mao et al., 2016a). The goal of this system is to perform resource allocation: the system has available resources (e.g., CPUs and memory), and a queue of pending jobs; and it needs to either schedule a pending job and assign some of the resources to it, or perform a "pass" action, in which no new jobs are assigned resources and the system waits for executing jobs to terminate and free up their assigned resources. The goal is to schedule jobs in a way that maximizes throughput. DeepRM achieves this by maintaining a model of the system (resources, incoming jobs), and using a pre-trained DNN to choose which action to

Katz, G.
Guarded Deep Learning using Scenario-based Modeling.
DOI: 10.5220/0009097601260136
In *Proceedings of the 8th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2020)*, pages 126-136
ISBN: 978-989-758-400-8; ISSN: 2184-4348

perform. DeepRM performs very well when compared to state-of-the-art, manually created software that tackles the same problem (Mao et al., 2016a).

Despite its overall satisfactory performance, the authors of DeepRM report that the system may sometimes behave in undesirable ways. For example, the controller might request that job x be allocated resources, although no job x exists in the job queue. In their implementation (Mao et al., 2016b), the authors address this situation by introducing an *override rule*: a piece of code that examines the current state of the system, and overrides the DNN's decision when this particular case is detected. Here, the override rule changes the controller's selection to "pass" whenever the controller requests to allocate resources to a non-existent job. There are additional override rules included in DeepRM (Mao et al., 2016b), and also in other systems (e.g., the Pensieve system (Mao et al., 2017)). Moreover, since DeepRM's release, additional undesirable behaviors have been discovered (Kazak et al., 2019), and addressing these might require augmenting the system with yet additional override rules in the future.

These cases, and others, indicate that override rules are becoming an integral component of DNN-based models. As erroneous behaviors may be discovered after the initial deployment phase, override rules may need to be added, extended, enhanced and refactored throughout the system's lifetime. We argue that this situation calls for leveraging suitable modeling techniques, in a way that will facilitate creating and maintaining override rules — leading to overall increased system reliability.

In this paper we advocate the use of the *scenario-based modeling* (*SBM*) framework (Harel et al., 2012b; Damm and Harel, 2001) for creating override rules. In SBM, individual system behaviors are modeled as independent scenarios, and are then automatically interwoven when the model is executed in order to produce cohesive system behavior. SBM has been shown to afford several benefits in system design and automated maintenance, and is particularly suitable for *incremental development* — which is a highly desirable trait when dealing with override rules. We propose here a method for applying SBM to systems with DNN components, in a way that allows to specify override rules as SBM scenarios. We discuss the benefits of the approach (in particular, those afforded by the amenability of SBM to automated analysis (Harel et al., 2015c)), and demonstrate its applicability to a few recently proposed systems. Although our focus here is on systems with DNN components, our approach could be extended to systems with different kinds of opaque components.

The rest of this paper is organized as follows. In Section 2 we provide the necessary background on SBM, DNNs and override rules. Next, in Section 3 we present our method for applying SBM to systems with DNN components. In Section 4 we describe an evaluation of our approach, followed by a discussion of related work in Section 5. We conclude in Section 6.

2 BACKGROUND

2.1 Deep Neural Networks and Override Rules

Deep neural networks (DNNs) are directed graphs, in which the nodes (neurons) are organized into layers. The first layer is the input layer, the last layer is the output layer, and the multiple remaining layers are the hidden layers. Each node in the network (except for input nodes) is connected to nodes from the preceding layer, using predetermined weight values (an illustration appears in Fig. 1). Selecting appropriate weight values is key, and is performed during a *training* phase, which is beyond the scope of this paper (for a survey, see, e.g., (Goodfellow et al., 2016)). A DNN is evaluated by assigning values to its input neurons, and then propagating these values forward through the network, each time computing values for a given layer from the values of its predecessor. Eventually, the output values (i.e., the values of neurons in the output layer) are computed, and are returned to the user. Often, DNNs are used as controllers or classifiers, in which case it is typical to return to the user the index of the output neuron that received the highest value. This neuron indicates the action, or classification, determined by the DNN.

For our purpose here, it is enough to regard a DNN as a black box, that transforms an input into an output. However, for completeness, we briefly describe how a DNN is evaluated. The value of each hidden node in the network is computed by calculating a weighted sum of the node values from the previous layer, according to the edge weights. Then, a non-linear *activation function* is applied to this weighted sum (Goodfellow et al., 2016), and its result becomes the value of the node being computed. For simplicity we focus here on the Rectified Linear Unit (ReLU) activation function (Nair and Hinton, 2010), given by $\mathrm{ReLU}(x) = \max(0, x)$. Thus, when a node uses the ReLU activation function, its value is calculated as the maximum of the linear combination of nodes from the previous layer and 0.

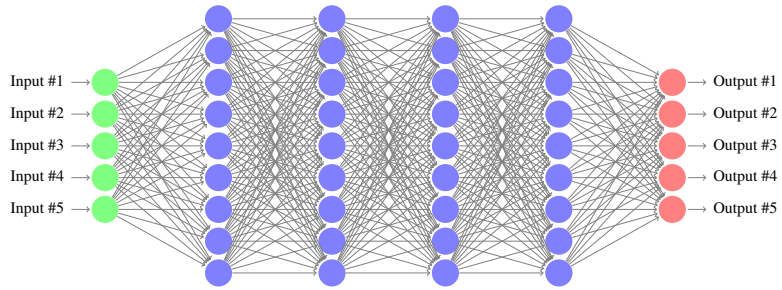

Figure 1: A fully connected DNN with 5 input nodes (in green), 5 output nodes (in red), and 4 hidden layers containing a total of 36 hidden nodes (in blue).

Fig. 2 depicts a small DNN that will serve as a running example. The network acts as a controller: it has two inputs, x_1 and x_2; three hidden neurons, v_1, v_2 and v_3, each with the ReLU activation functions; and it selects one of two possible actions through its output neurons, y_1 and y_2. We slightly abuse notation, and use y_1 and y_2 to denote both the neurons and the actions/classes those neurons represent. The selected action is the one assigned the highest score. We see, for example, that assigning $x_1 = 1, x_2 = 0$ results in output values $y_1 = 1, y_2 = 0$, i.e., action y_1 is selected; whereas $x_1 = 0, x_2 = 1$ leads to $y_1 = 0, y_2 = 3$, i.e. action y_2 is selected.

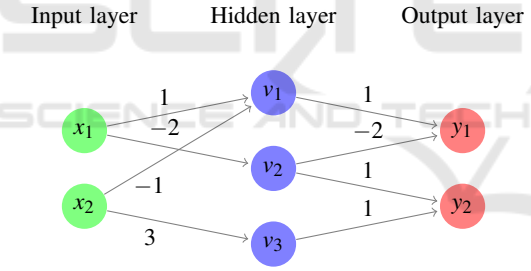

Figure 2: A small neural network with a single hidden layer.

An *override rule* is a triple $\langle P, Q, \alpha \rangle$, where P is a predicate over the network's inputs, Q is a predicate over the network's outputs, and α is an override action. The semantics of these rules is that if P and Q hold for a network's evaluation, then output action α should be selected — regardless of the network's output. For example, we might specify the rule

$$\langle x_1 > 0 \wedge x_2 < x_1, true, y_2 \rangle$$

which would be triggered for inputs $x_1 = 1, x_2 = 0$. As we saw previously, in this case the network outputs y_1; but with the override rule, this selection would be changed to y_2. By setting Q to true, we created an override rule the only examines the DNN's inputs. We could, for example, set Q to $y_2 > 10$, in which case the rule would not be triggered for $x_1 = 1, x_2 = 0$. By adjusting P and Q, the aforementioned formulation can express many common override rules, such as the rule in the DeepRM example described in Section 1.

2.2 Scenario-based Modeling

Scenario-based modeling (Harel et al., 2012b) is an approach for modeling complex reactive systems. At the core of the approach lies the notion of a *scenario object*: a description of a single behavior, whether desirable or undesirable, of the system being modeled. Each scenario object is created separately, and does not directly interact with the other scenarios; instead, it only interacts with a global execution mechanism. This execution mechanism can execute a set of scenarios in a way that produces cohesive, global behavior.

There are several flavors of scenario-based modeling, which may differ in the various idioms that a scenario object uses to interact with the execution mechanism and affect the overall execution of the system. Here, we focus on the commonly used idioms of *requesting*, *waiting-for* and *blocking* events (Harel et al., 2012b). When executed, each scenario object may declare it has reached a *synchronization point*, in which the execution infrastructure must trigger an event. The object then specifies which events it would like to have triggered (*requested* events); which events it forbids from being triggered (*blocked* events); and which events it does not actively request, but should be notified in case they are triggered by the execution mechanism (*waited-for events*). The execution infrastructure waits for all the scenario objects to synchronize (or just for a subset thereof, depending on the semantics used (Harel et al., 2013a)); selects an event that is requested and not blocked for triggering; and informs any relevant scenario object that this event has been triggered.

A toy example of a scenario-based model appears in Fig. 3. The model depicted therein belongs to a system that controls the water level in a tank with hot and cold water taps. Each scenario object is depicted as a transition system, where the nodes repre-

sent the predetermined synchronization points. The scenario object AddHotWater repeatedly waits for WaterLow events and requests three times the event AddHot; and the scenario object AddColdWater performs a symmetrical operation with cold water. In a model that includes only the objects AddHotWater and AddColdWater, the three AddHot events and three AddCold events may be triggered in any order during execution. In order to maintain the stability of the water temperature in the tank, the scenario object Stability enforces the interleaving of AddHot and AddCold events by using event blocking. The execution trace of the resulting model is depicted in the event log.

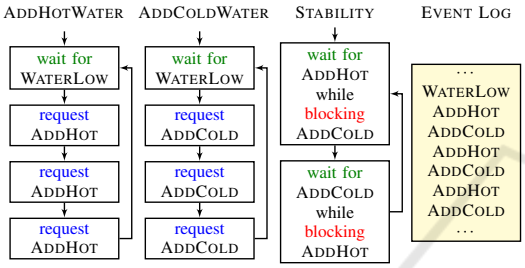

Figure 3: (From (Harel et al., 2014)) A scenario-based model of a system that controls the water level in a tank with hot and cold water taps.

SBM has been implemented in a variety of high-level languages, such as Java (Harel et al., 2010), C++ (Harel and Katz, 2014), JavsScript (Bar-Sinai et al., 2018) and ScenarioTools (Greenyer et al., 2017). The methodology has been successfully used in modeling complex systems, such as web-servers (Harel and Katz, 2014), cache coherence protocols (Harel et al., 2016a) and robotic controllers (Gritzner and Greenyer, 2018). For simplicity and generality, in the remainder of this paper we mostly describe scenario-based models in terms of transitions systems.

Following the definitions in (Katz, 2013), we formalize the SBM framework as follows. A scenario object O over event set E is a tuple $O = \langle Q, \delta, q_0, R, B \rangle$, where the components are interpreted as follows:

- Q is a set of states, each representing one of the predetermined synchronization points;
- q_0 is the initial state;
- $R : Q \to 2^E$ and $B : Q \to 2^E$ map states to the sets of events requested and blocked at these states (respectively); and
- $\delta : Q \times E \to 2^Q$ is a transition function, indicating how the object reacts when an event is triggered.

Scenario objects can be composed into a single, larger scenario object, as follows. For objects $O^1 = \langle Q^1, \delta^1, q_0^1, R^1, B^1 \rangle$ and $O^2 = \langle Q^2, \delta^2, q_0^2, R^2, B^2 \rangle$ over a common event set E, we define the composite scenario object $O^1 \parallel O^2$ as $O^1 \parallel O^2 = \langle Q^1 \times Q^2, \delta, \langle q_0^1, q_0^2 \rangle, R^1 \cup R^2, B^1 \cup B^2 \rangle$, where:

- $\langle \tilde{q}^1, \tilde{q}^2 \rangle \in \delta(\langle q^1, q^2 \rangle, e)$ if and only if $\tilde{q}^1 \in \delta^1(q^1, e)$ and $\tilde{q}^2 \in \delta^2(q^2, e)$; and
- The union of the labeling functions is defined in the natural way; e.g. $e \in (R^1 \cup R^2)(\langle q^1, q^2 \rangle)$ if and only if $e \in R^1(q^1) \cup R^2(q^2)$, and $e \in (B^1 \cup B^2)(\langle q^1, q^2 \rangle)$ if and only if $e \in B^1(q^1) \cup B^2(q^2)$.

A *behavioral model* M is defined as a collection of scenario objects $O^1, O^2, \ldots, O^n$, and the executions of M are the executions of the composite object $O = O^1 \parallel O^2 \parallel \ldots \parallel O^n$. Each such execution starts from the initial state of O, and in each state q along the run an enabled event is chosen for triggering, if one exists (i.e., an event $e \in R(q) - B(q)$). Then, the execution moves to a state $\tilde{q} \in \delta(q, e)$, and so on.

One extension of SBM, which will be useful in our context, is to treat events as *variables* (Katz et al., 2019b). For example, an event e can be declared to be of type integer. Then, one scenario object might *request* $e \geq 5$, while another object might block $e \geq 7$. The execution framework would then employ a *constraint solver*, such as an SMT solver (Barrett and Tinelli, 2018), to resolve the constraints and trigger, e.g., the event $e = 6$. We omit here the formal definition of this extension, which is straightforward, and refer the interested reader to (Katz et al., 2019b).

3 MODELING OVERRIDE SCENARIOS

In the DeepRM case, override rules have been added as unrestricted Python code within the module that invokes the DNN and processes its result (Mao et al., 2016b). Thus, while the DNN controller itself is clearly structured and well defined, override rules are phrased as arbitrary pieces of code. This could lead to several complications: (i) as the number of override rules increases, they might become convoluted and difficult to comprehend, extend and maintain; (ii) the semantics of override rules might be unclear. For example, in the case of multiple rules that can all be applied, which one prevails? Is there a particular order in which they should be checked? Can rules interact? etc; and (iii) the conditions employed within these override rules might become more complex, hiding away some of the model's logic where other developers might not expect to find it.

Here, we propose to model override rules using SBM, as a means for mitigating these difficulties. SBM is geared towards incremental modeling, which seems a particularly likely scenario when DNNs are involved: due to the opacity of DNNs, some undesirable behaviors are likely to be detected only after deployment, requiring the addition of new override rules. Further, SBM's simple semantics would guarantee that interactions between the override rules are well defined. Finally, there is a substantial body of work on automatically verifying, analyzing and optimizing SBM models, which could prove useful in detecting conflicts between override rules or simplifying them when their number increases.

3.1 Modeling DNNs and Override Rules in SBM

We propose the following method for creating SBM models that combine scenario objects and a DNN controller. The core idea is to represent the DNN as a dedicated scenario object, O_{DNN}, to be included in the scenario-based model. This O_{DNN} is a non-deterministic scenario that models the DNN controller, thus allowing it to interact with the other scenario objects. Let us assume, for the sake of simplicity (we relax this limitation later), that there is a finite set of possible inputs to the DNN, denoted $\mathbb{I}$; and let $\mathbb{O}$ denote the set of possible actions among which the DNN chooses. We introduce new events to our event set E: an event e_i for every $i \in \mathbb{I}$, and an event e_o for every $o \in \mathbb{O}$. We have our new scenario object O_{DNN} repeatedly wait for all events e_i, and then request all events e_o. This behavior represents the black-box nature of the DNN, as far as the rest of the model is concerned: we only know that after an input arrives, one of the outputs will be selected, without knowing which. However, when the model is executed, the execution infrastructure resolves this non-determinism by running the actual DNN and triggering the output event that corresponds to its selection. For example, assuming just two possible inputs, e.g. $i_1 = \langle 1,0 \rangle$ and $i_2 = \langle 0,1 \rangle$, the network depicted in Fig. 2 would be represented by the scenario object described in Fig. 4.

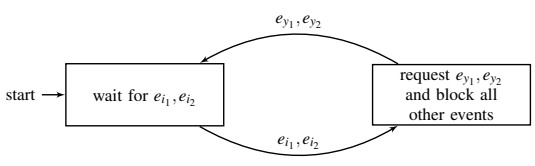

Figure 4: A scenario O_{DNN} for the neural network in Fig. 2. Events e_{i_1} and e_{i_2} represent the inputs to the neural network, and events e_{y_1} and e_{y_2} represent its outputs.

We introduce the convention that other scenario objects in the systems may wait-for, but may not block, the input events e_i. A single dedicated scenario, called a *sensor*, is responsible for requesting an input event when the DNN needs to be evaluated (e.g., following a user action). By another convention, no scenario object except O_{DNN} may request any of the output events e_o; however, other scenario objects may wait-for or block these events. During execution, if the DNN assigns the highest score to an event that is currently blocked, we resolve the non-determinism of O_{DNN} by selecting the event representing the output with the next-to-highest score, and so on. If no events are left unblocked, then the system is deadlocked and the execution terminates.

The motivation underlying our definitions is to allow scenario objects to monitor the inputs and outputs of the DNN controller, by waiting for their respective events; and then to interfere with the recommendation of the DNN, by blocking certain output events from being triggered, which is the main use-case of override rules. Note that a scenario object may force the DNN to produce a specific output, by blocking all other possibilities; or it may interfere more subtly, by blocking some events and allowing the DNN to choose among the remaining events.

In practice, our assumption that the sets $\mathbb{I}$ of possible DNN inputs and the set $\mathbb{O}$ of possible DNN outputs are finite might be a limiting factor: for example, in the override rule described in Section 2.1, the relative assignments to x_1 and x_2 affected whether the rule could be triggered or not, and so it is important to express in our model the exact assignments for x_1 and x_2. Of course, there are infinitely many possible assignments. To waive this limitation we again turn to the extension to SBM (Katz et al., 2019b) that allows us to treat events as variables of certain types. We change our formulation slightly: scenario objects in the system may wait-for a single, composite event that indicates that values have been assigned to (all of) the DNN's inputs or outputs, and may then act according to those values.

Using this extension, the override rule from Section 2.1 is expressed as a scenario object in Fig. 5. The scenario enforces the override rule that whenever $x_1 > 0$ and $x_2 < x_1$, output event y_2 (and not y_1) should be triggered. Here, $\langle e_{x_1}, e_{x_2} \rangle$ represents a single event that indicates that values have been assigned to the DNN's inputs. This event contains two real values, x_1 and x_2, that the scenario can then access and use to determine its transition. Event e_{y_1} indicates, as before, that the scenario forbids the DNN from selecting y_1 as its output action.

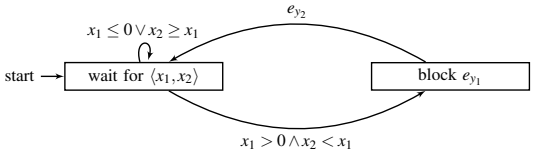

Figure 5: A scenario object enforcing the override rule that whenever $x_1 > 0$ and $x_2 < x_1$, output event y_2 should be triggered.

3.2 Liveness Properties

Override rules are typically used to enforce safety properties ("bad things never happen"). However, there is sometimes a need to enforce *liveness* properties ("good things eventually happen"). In particular, this can happen in the context of online reinforcement learning (Sutton and Barto, 1998) — where the DNN controller might change over time, and we wish to ensure that it eventually tries out new actions. If these actions turn out to be beneficial, the RL mechanism will ensure that the DNN repeats them in the future. Liveness properties are also relevant when there are fairness constraints; for example, if we wish to ensure that in a resource allocation system, every pending job eventually gets scheduled.

An example appears in (Kazak et al., 2019), where the authors discuss the *Custard* system: a congestion control system that uses a DNN to monitor the conditions of a computer network and select a sending bit rate, in order to minimize congestion (Jay et al., 2018). In (Kazak et al., 2019), Custard is examined to see if there are cases in which the DNN controller chooses a sub-optimal sending rate that does not utilize all available bandwidth, and never attempts to increase this bit rate. This kind of behavior constitutes a liveness violation, which we would like to prevent using an override rule.

SBM can encode the fact that one or multiple DNN output actions should eventually be blocked, thus forcing the DNN controller to pick a different alternative. This can be enforced by having a scenario object wait for n consecutive rounds where a particular output is triggered, and then block it; an example for $n = 3$ appears in Fig. 6, where a scenario looks for 3 consecutive DNN evaluations where y_2 is triggered, after which it blocks y_2 once, forcing the DNN to select another action. An alternative is to have the override rule block the particular output event with a very low probability (Harel et al., 2014), thus enforcing the fact that it will *eventually* be blocked with probability 1.

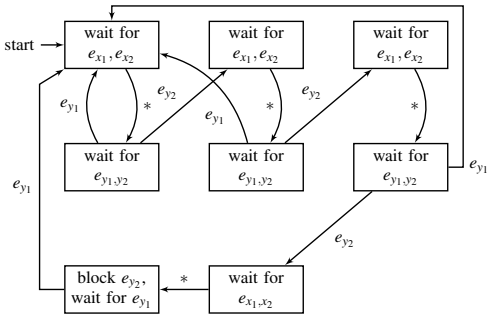

Figure 6: A scenario object that enforces a liveness property for the network from Fig. 4.

3.3 Automated Analysis

Scenario-based modeling has been shown to facilitate automated formal analysis (Harel et al., 2015c). Specifically, the simple synchronization constructs that scenario objects in SBM models use render tasks such as model checking (Katz et al., 2015), compositional verification (Harel et al., 2013b) and automated repair (Katz, 2013) simpler than they would be for less restricted models. We argue that these properties add to the attractiveness of SBM as a formalism for expressing override rules.

One particular use case that illustrates the aforementioned claim is deadlock freedom. As additional override rules are added, perhaps by different modelers, there is a risk that a certain sequence of inputs to the DNN might cause a deadlock. As a simple illustrative example, consider the override rule expressed in Fig. 5: whenever $x_1 > 0$ and $x_2 < x_1$, output y_2 should be selected. Suppose now that another modeler, concerned about the fact that the DNN might always advise y_2, adds the override rule depicted in Fig. 6: after 3 consecutive y_2 events, a different event must be selected. These two override rules together might result in a deadlock: for example, if the DNN receives the inputs $x_1 = 2, x_2 = 1$ three consecutive times, both override rule would be triggered, simultaneously blocking both output events e_{y_1} and e_{y_2}.

Such situations can be avoided by running a verification query that ensures that the system is deadlock free. This query can be run, e.g., after the addition of each new override rule. Should a deadlock be detected, the counter-example provided by the verification tool could guide the modeler in changing the conflicting rules — after which verification can be run again, to ensure that the system is now indeed deadlock free. Of course, additional system-specific properties, beyond deadlock freedom, could also be verified.

4 EVALUATION

For evaluation purposes, we implemented our approach on top of the BPC framework for scenario-based modeling in C++ (Harel and Katz, 2014) (of course, other SBM frameworks could also be used). The BPC package allows modelers to leverage many of the powerful constructs of C++, while forcing them to adhere to the SBM principles: each scenario is modeled as a separate object, and inter-scenario interactions are performed through a global execution mechanism that BPC provides. Here, we used BPC to model override rules for the DeepRM system for resource management, and for the Custard system for congestion control.

4.1 Override Rules for DeepRM

The DeepRM system (Mao et al., 2016a) (discussed in Section 1) performs resource allocation: it assigns available resources to pending jobs, with the goal of maximizing throughput. As part of our evaluation we implemented an override rule that prevents the DNN controller from attempting to assign resources to non-existing jobs, which is an undesirable behavior that occurs in practice (Mao et al., 2016b).

BPC code for our override rule, implemented as a scenario object, appears in Fig. 7. We assume that the queue of pending jobs is of size 5, and that the DNN's output actions are denoted $y_0, y_1, \ldots, y_5$. Action y_i for $1 \leq i \leq 5$ means that the job in slot i of the queue should be allocated resources, and the special action y_0 is the "pass" action, indicating that no job should be allocated resources at this time. We use x to denote an event indicating that the DNN needs to be evaluated on certain input values, available as parameters of x. The state of the job queue is part of the input to the DNN controller. Specifically, we use $x[i]$ for $1 \leq i \leq 5$ to denote a Boolean value that indicates whether or not there is currently a pending job in slot i of the queue.

The override scenario object is implemented as a class that inherits from BPC's special BThread class. The scenario object can then use the special bSync() method to initialize a synchronization point with the other scenarios in the model (including O_{DNN}, the scenario object that models the DNN controller). This method takes as input three Event vectors — the first containing the set of requested events, the second containing the set of waited-for events, and the third containing the set of blocked events. The bSync() call suspends the object's execution until the BPC mechanism selects and triggers an event; then, if the triggered event was requested or waited-for by the scenario object, the scenario resumes execution and can retrieve the triggered event using the lastEvent() method.

Our scenario object runs in an infinite loop, each time waiting for the input event x to be triggered. When that happens, it examines x to determine which slots of the job queue are occupied; and then synchronizes again to block event y_i for any unoccupied slots. Note that this scenario object can never cause a deadlock, because it never blocks event y_0.

```
class EnsureJobExists : public BThread {
  void entryPoint() {
    Vector<Event> emptySet = {};
    Vector<Event> allInputs = { x };
    Vector<Event> allOutputs = { y0,...,y5 };

    while ( true ) {
      bSync( emptySet, allInputs, emptySet );
      lastInput = lastEvent();
      Vector<Event> blocked = {};

      for ( int i = 1; i <= 5; ++i ) {
        if ( !lastInput[i] )
          blocked.append( yi )
      }

      bSync( emptySet, allOutputs, blocked )
    }
  }
}
```

Figure 7: A scenario object for preventing the DeepRM DNN controller from assigning resources to non-existing jobs.

4.2 Override Rules for Custard

As briefly discussed in Section 3.2, Custard is a DNN-based congestion control system. The DNN controller takes as input various readings about the current and previous state of the computer network (e.g., throughputs, loss rates, and latency), and selects the next sending bit rate. Custard is a reactive system, designed to be run continuously and use the results of its past decisions (as reflected in past network readings) when making its next choice of bit rate.

Due to the opacity of the DNN controller, one concern when using Custard is that it might be too *conservative*. Specifically, we may wish to avoid a situation where the state of the computer network is completely steady, and yet the DNN controller never tries to increase the sending bit rate — and thus never

finds out whether there is additional, currently unused bandwidth.

A scenario object that prevents this case is depicted in Fig. 8. The scenario looks for a situation where the DNN's inputs and outputs have been identical for the last $n = 10$ rounds, and when this is detected it blocks the previous output action. Event x represents here an input assignment (comprised of multiple input values) on which the DNN has been evaluated, and event y represents the DNN's output selection. Here, for simplicity, we do not examine the actual values of x, and only look for repeating assignments (in practice, we may want to apply this override rule only if the physical network's conditions are both *steady* and *good*, indicating that there may be unused bandwidth).

```
const int n = 10;

class PreventSteadyState : public BThread {
  void entryPoint() {
    Vector<Event> empty;
    Vector<Event> allInputs = { x };
    Vector<Event> allOutputs = { y };

    Event lastInput;
    Event lastOutput;

    while ( true ) {
      bSync( empty, allInputs, empty );
      lastInput = lastEvent();

      bSync( empty, allOutputs, empty );
      lastOutput = lastOutput();

      bool steadyState = true;
      int i = 1;
      while ( i < n && steadyState ) {
        bSync( empty, allInputs, empty );
        if ( lastInput != lastEvent() )
          steadyState = false;

        bSync( empty, allOutputs, empty );
        if ( lastOutput != lastEvent() )
          steadyState = false;

        ++i;
      }

      if ( steadyState ) {
        bSync( empty, allInputs, empty );
        bSync( empty, allOutputs, lastOuptut );
      }
    }
  }
}
```

Figure 8: A scenario object for enforcing the Custard DNN to choose a different action if the state has been steady for $n = 10$ iterations.

5 RELATED WORK

Override rules, sometimes also referred to as *shields*, have been applied ad-hoc in multiple DNN-enabled systems, such as DeepRM (Mao et al., 2016a) and Pensieve (Mao et al., 2017). Such rules, and related forms of runtime monitors, are also found in control systems for robots (Phan et al., 2017), drones (Desai et al., 2018), and in various other formalisms which are not directed particularly at deep learning (Hamlen et al., 2006; Falcone et al., 2011; Schierman et al., 2015; Ji and Lafortune, 2017; Wu et al., 2018). The formal methods community has recently taken an interest in override rules for systems with DNNs: for example, by proposing techniques to synthesize rules that affect the controller as little as possible (Avni et al., 2019; Wu et al., 2019).

Various aspects of the SBM formalism, especially those pertaining to the formal analysis of scenario-based models, have been studied over the years. These aspects include the automatic repair (Harel et al., 2012a), verification (Harel et al., 2015b), synthesis (Greenyer et al., 2016a) and optimization (Harel et al., 2015a; Greenyer et al., 2016b; Steinberg et al., 2017; Steinberg et al., 2018; Harel et al., 2020) of models. SBM is also a key component of the Wise Computing initiative (Marron et al., 2016; Harel et al., 2016b; Harel et al., 2018), which seeks to transform the computer into a proactive team member, capable of developing complex models alongside human engineers.

In this paper we focused on scenario-based modeling as a possible formalism for expressing override rules. There are other, related modeling schemes, which could also be used in similar contexts. For example, publish-subscribe is a related framework for parallel composition, which shares many traits with SBM (Eugster et al., 2003). Aspect oriented programming (Kiczales et al., 1997) is another formalism that allows to specify and execute cross-cutting program instructions on top of a base application. Both of these approaches, however, do not directly support specifying forbidden behavior, which appears quite useful for specifying override rules. Additional behavior- and scenario-based models, such as Brooks's subsumption architecture (Brooks, 1986), Branicky's be-

havioral programming (Branicky, 1999), and LEGO Mindstorms leJOS (see (Arkin, 1998)), all call for constructing systems from individual behaviors. One advantage that SBM affords compared to these formalisms is that it is language-independent, has been implemented on top of multiple platforms, and can extend in a variety of ways the coordination and arbitration mechanisms used by these architectures.

The BIP formalism (behavior, interaction, priority) uses the notion of glue for assembling components into a cohesive system (Bliudze and Sifakis, 2008). The goals that it pursues are similar to those of SBM, although BIP's focuses mostly on correct-by-construction systems — while SBM is more geared towards executing intuitively specified scenarios, and resolving the constraints that they pose at run-time.

6 DISCUSSION AND NEXT STEPS

With the increasing use of DNNs in various systems, there is an urgent need to ensure their safety, specifically by using override rules. We argue here that progress can be made towards this goal by using modeling schemes that model together the DNN and its override rules. We propose to use scenario-based modeling for this purpose, show how the basic scenario-based scheme can be extended to incorporate DNNs, and demonstrate the approach on several examples.

Moving towards a more structured methodology for modeling override rules raises the following question: as the number of override rules and their sophistication increases, could they fully capture the model's logic and render the DNN obsolete? We believe that the answer is negative, as override rules often forbid some specific behavior, but still rely on the DNN to prioritize among the remaining options. We believe that an optimal approach is to combine a DNN component with appropriately modeled override rules, while maintaining and enhancing both components throughout the system's lifetime.

Our work to date is but a first step, which we plan to extend. Specifically, we intend to work on (i) customizing the idioms of SBM, or related techniques, to better suit integration with DNNs and guard them in more subtle ways; and (ii) leveraging the other advantages of SBM, specifically its amenability to verification and automated analysis, in proving the overall correctness of DNN-enhanced models. In the longer run, we believe that work in this direction will lead to the creation of DNN-enabled systems that are more robust and easier to maintain and extend.

ACKNOWLEDGEMENTS

We thank Yafim (Fima) Kazak for his contributions to this project, and the anonymous reviewers for their insightful comments. The project was partially supported by grants from the Binational Science Foundation (2017662) and the Israel Science Foundation (683/18).

REFERENCES

Amodei, D., Olah, C., Steinhardt, J., Christiano, P., Schulman, J., and Mané, D. (2016). Concrete Problems in AI Safety. Technical Report. https://arxiv.org/abs/1606.06565.

Arkin, R. C. (1998). *Behavior-Based Robotics*. MIT Press.

Avni, G., Bloem, R., Chatterjee, K., Henzinger, T., Könighofer, B., and Pranger, S. (2019). Run-Time Optimization for Learned Controllers through Quantitative Games. In *Proc. 31st Int. Conf. on Computer Aided Verification (CAV)*, pages 630–649.

Bar-Sinai, M., Weiss, G., and Shmuel, R. (2018). BPjs: An Extensible, Open Infrastructure for Behavioral Programming Research. In *Proc. 21st ACM/IEEE Int. Conf. on Model Driven Engineering Languages and Systems (MODELS)*, pages 59–60.

Barrett, C. and Tinelli, C. (2018). Satisfiability Modulo Theories. In Clarke, E., Henzinger, T., Veith, H., and Bloem, R., editors, *Handbook of Model Checking*, pages 305–343. Springer International Publishing.

Bliudze, S. and Sifakis, J. (2008). A Notion of Glue Expressiveness for Component-Based Systems. In *Proc. 19th Int. Conf. on Concurrency Theory (CONCUR)*, pages 508–522.

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

Branicky, M. (1999). Behavioral Programming. In *Working Notes AAAI Spring Symposium on Hybrid Systems and AI*.

Brooks, R. (1986). A Robust Layered Control System for a Mobile Robot. *Robotics and Automation*, 2(1):14–23.

Chicco, D., Sadowski, P., and Baldi, P. (2014). Deep Autoencoder Neural Networks for Gene Ontology Annotation Predictions. In *Proc. 5th ACM Conf. on Bioinformatics, Computational Biology, and Health Informatics (BCB)*, pages 533–540.

Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., and Kuksa, P. (2011). Natural Language Processing (Almost) from Scratch. *Journal of Machine Learning Research (JMLR)*, 12:2493–2537.

Damm, W. and Harel, D. (2001). LSCs: Breathing Life into Message Sequence Charts. *Journal on Formal Methods in System Design (FMSD)*, 19(1):45–80.

Desai, A., Ghosh, S., Seshia, S., Shankar, N., and Tiwari, A. (2018). SOTER: Programming Safe Robotics

System using Runtime Assurance. Technical Report. https://arxiv.org/abs/1808.07921.

Elkahky, A., Song, Y., and He, X. (2015). A Multi-View Deep Learning Approach for Cross Domain User Modeling in Recommendation Systems. In *Proc. 24th Int. Conf. on World Wide Web (WWW)*, pages 278–288.

Eugster, P., Felber, P., Guerraoui, R., and Kermarrec, A. (2003). The Many Faces of Publish/Subscribe. *ACM Computing Surveys (CSUR)*, 35(2):114–131.

Falcone, Y., Mounier, L., Fernandez, J., and Richier, J. (2011). Runtime Enforcement Monitors: Composition, Synthesis, and Enforcement Abilities. *Journal on Formal Methods in System Design (FMSD)*, 38(3):223–262.

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

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press.

Gottschlich, J., Solar-Lezama, A., Tatbul, N., Carbin, M., Rinard, M., Barzilay, R., Amarasinghe, S., Tenenbaum, J., and Mattson, T. (2018). The Three Pillars of Machine Programming. In *Proc. 2nd ACM SIGPLAN Int. Workshop on Machine Learning and Programming Languages (MAPL)*, pages 69–80.

Greenyer, J., Gritzner, D., Gutjahr, T., König, F., Glade, N., Marron, A., and Katz, G. (2017). ScenarioTools — A Tool Suite for the Scenario-based Modeling and Analysis of Reactive Systems. *Journal of Science of Computer Programming (J. SCP)*, 149:15–27.

Greenyer, J., Gritzner, D., Katz, G., and Marron, A. (2016a). Scenario-Based Modeling and Synthesis for Reactive Systems with Dynamic System Structure in ScenarioTools. In *Proc. 19th ACM/IEEE Int. Conf. on Model Driven Engineering Languages and Systems (MODELS)*, pages 16–23.

Greenyer, J., Gritzner, D., Katz, G., Marron, A., Glade, N., Gutjahr, T., and König, F. (2016b). Distributed Execution of Scenario-Based Specifications of Structurally Dynamic Cyber-Physical Systems. In *Proc. 3rd Int. Conf. on System-Integrated Intelligence: New Challenges for Product and Production Engineering (SYSINT)*, pages 552–559.

Gritzner, D. and Greenyer, J. (2018). Synthesizing Executable PLC Code for Robots from Scenario-Based GR(1) Specifications. In *Proc. 4th Workshop of Model-Driven Robot Software Engineering (MORSE)*, pages 247–262.

Hamlen, K., Morrisett, G., and Schneider, F. (2006). Computability Classes for Enforcement Mechanisms. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(1):175–205.

Harel, D., Kantor, A., and Katz, G. (2013a). Relaxing Synchronization Constraints in Behavioral Programs. In *Proc. 19th Int. Conf. on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, pages 355–372.

Harel, D., Kantor, A., Katz, G., Marron, A., Mizrahi, L., and Weiss, G. (2013b). On Composing and Proving the Correctness of Reactive Behavior. In *Proc. 13th Int. Conf. on Embedded Software (EMSOFT)*, pages 1–10.

Harel, D., Kantor, A., Katz, G., Marron, A., Weiss, G., and Wiener, G. (2015a). Towards Behavioral Programming in Distributed Architectures. *Journal of Science of Computer Programming (J. SCP)*, 98:233–267.

Harel, D. and Katz, G. (2014). Scaling-Up Behavioral Programming: Steps from Basic Principles to Application Architectures. In *Proc. 4th SPLASH Workshop on Programming based on Actors, Agents and Decentralized Control (AGERE!)*, pages 95–108.

Harel, D., Katz, G., Lampert, R., Marron, A., and Weiss, G. (2015b). On the Succinctness of Idioms for Concurrent Programming. In *Proc. 26th Int. Conf. on Concurrency Theory (CONCUR)*, pages 85–99.

Harel, D., Katz, G., Marelly, R., and Marron, A. (2016a). An Initial Wise Development Environment for Behavioral Models. In *Proc. 4th Int. Conf. on Model-Driven Engineering and Software Development (MODELSWARD)*, pages 600–612.

Harel, D., Katz, G., Marelly, R., and Marron, A. (2016b). First Steps Towards a Wise Development Environment for Behavioral Models. *Int. Journal of Information System Modeling and Design (IJISMD)*, 7(3):1–22.

Harel, D., Katz, G., Marelly, R., and Marron, A. (2018). Wise Computing: Towards Endowing System Development with Proactive Wisdom. *IEEE Computer*, 51(2):14–26.

Harel, D., Katz, G., Marron, A., Sadon, A., and Weiss, G. (2020). Executing Scenario-Based Specification with Dynamic Generation of Rich Events. *Communications in Computer and Information Science (CCIS)*, 1161.

Harel, D., Katz, G., Marron, A., and Weiss, G. (2012a). Non-Intrusive Repair of Reactive Programs. In *Proc. 17th IEEE Int. Conf. on Engineering of Complex Computer Systems (ICECCS)*, pages 3–12.

Harel, D., Katz, G., Marron, A., and Weiss, G. (2014). Non-Intrusive Repair of Safety and Liveness Violations in Reactive Programs. *Transactions on Computational Collective Intelligence (TCCI)*, 16:1–33.

Harel, D., Katz, G., Marron, A., and Weiss, G. (2015c). The Effect of Concurrent Programming Idioms on Verification. In *Proc. 3rd Int. Conf. on Model-Driven Engineering and Software Development (MODELSWARD)*, pages 363–369.

Harel, D., Marron, A., and Weiss, G. (2010). Programming Coordinated Scenarios in Java. In *Proc. 24th European Conf. on Object-Oriented Programming (ECOOP)*, pages 250–274.

Harel, D., Marron, A., and Weiss, G. (2012b). Behavioral Programming. *Communications of the ACM (CACM)*, 55(7):90–100.

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

Jay, N., Rotman, N., Brighten Godfrey, P., Schapira, M., and Tamar, A. (2018). Internet Congestion Control via

Deep Reinforcement Learning. In *Proc. 32nd Conf. on Neural Information Processing Systems (NeurIPS)*.

Ji, Y. and Lafortune, S. (2017). Enforcing Opacity by Publicly Known Edit Functions. In *Proc. 56th IEEE Annual Conf. on Decision and Control (CDC)*, pages 12–15.

Julian, K., Lopez, J., Brush, J., Owen, M., and Kochenderfer, M. (2016). Policy Compression for Aircraft Collision Avoidance Systems. In *Proc. 35th Digital Avionics Systems Conf. (DASC)*, pages 1–10.

Katz, G. (2013). On Module-Based Abstraction and Repair of Behavioral Programs. In *Proc. 19th Int. Conf. on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, pages 518–535.

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

Katz, G., Barrett, C., and Harel, D. (2015). Theory-Aided Model Checking of Concurrent Transition Systems. In *Proc. 15th Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, pages 81–88.

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

Katz, G., Marron, A., Sadon, A., and Weiss, G. (2019b). On-the-Fly Construction of Composite Events in Scenario-Based Modeling Using Constraint Solvers. In *Proc. 7th Int. Conf. on Model-Driven Engineering and Software Development (MODELSWARD)*, pages 143–156.

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

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., and Irwin, J. (1997). Aspect-Oriented Programming. In *Proc. 11th European Conf. on Object-Oriented Programming (ECOOP)*, pages 220–242.

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

Mao, H., Alizadeh, M., Menache, I., and Kandula, S. (2016b). Resource Management with Deep Reinforcement Learning: Implementation. https://github.com/hongzimao/deeprm.

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

Marron, A., Arnon, B., Elyasaf, A., Gordon, M., Katz, G., Lapid, H., Marelly, R., Sherman, D., Szekely, S., Weiss, G., and Harel, D. (2016). Six (Im)possible Things before Breakfast: Building-Blocks and Design-Principles for Wise Computing. In *Proc. 19th ACM/IEEE Int. Conf. on Model Driven Engineering Languages and Systems (MODELS)*, pages 94–100.

Nair, V. and Hinton, G. (2010). Rectified Linear Units Improve Restricted Boltzmann Machines. In *Proc. 27th Int. Conf. on Machine Learning (ICML)*, pages 807–814.

Phan, D., Yang, J., Grosu, R., Smolka, S., and Stoller, S. (2017). Collision Avoidance for Mobile Robots with Limited Sensing and Limited Information about Moving Obstacles. *Journal on Formal Methods in System Design (FMSD)*, 51(1):62–68.

Schierman, J., DeVore, M., Richards, N., Gandhi, N., Cooper, J., Horneman, K., Stoller, S., and Smolka, S. (2015). Runtime Assurance Framework Development for Highly Adaptive Flight Control Systems. Technical Report. https://apps.dtic.mil/docs/citations/AD1010277.

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

Simonyan, K. and Zisserman, A. (2014). Very Deep Convolutional Networks for Large-Scale Image Recognition. Technical Report. http://arxiv.org/abs/1409.1556.

Steinberg, S., Greenyer, J., Gritzner, D., Harel, D., Katz, G., and Marron, A. (2017). Distributing Scenario-Based Models: A Replicate-and-Project Approach. In *Proc. 5th Int. Conf. on Model-Driven Engineering and Software Development (MODELSWARD)*, pages 182–195.

Steinberg, S., Greenyer, J., Gritzner, D., Harel, D., Katz, G., and Marron, A. (2018). Efficient Distributed Execution of Multi-Component Scenario-Based Models. *Communications in Computer and Information Science (CCIS)*, 880:449–483.

Sutton, R. and Barto, A. (1998). *Introduction to Reinforcement Learning*. MIT press Cambridge.

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

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

Wu, M., Wang, J., Deshmukh, J., and Wang, C. (2019). Shield Synthesis for Real: Enforcing Safety in Cyber-Physical Systems. Technical Report. https://arxiv.org/abs/1908.05402.

Wu, Y., Raman, V., Rawlings, B., Lafortune, S., and Seshia, S. (2018). Synthesis of Obfuscation Policies to Ensure Privacy and Utility. *Journal of Automated Reasoning*, 60(1):107–131.