DeepAuto: A First Step Towards Formal Verification of Deep Learning Systems

Yuteng Lu* ,Weidi Sun*, Guangdong Bai† and Meng Sun*
*School of Mathematical Sciences, Peking University, Beijing, China
Email: {luyuteng, weidisun, sunm}@pku.edu.cn
†School of Information Technology and Electrical Engineering, The University of Queensland, Brisbane, Australia
Email: g.bai@uq.edu.cn

***Abstract*—Deep Learning (DL) offers a data-driven programming paradigm in which Deep Neural Networks (DNNs) can be constructed through a set of training data. It has been widely adopted in many real-world applications. However, many studies have shown that DL systems suffer from adversarial attacks, especially when they are applied to security- and safety-critical domains. Given that formal verification has proved a great success in many areas such as software engineering, using it to achieve a high-level security assurance in DL systems is considered promising. In this paper, we design and implement *DeepAuto* which makes the significant bridge between automata and DNNs. With the aid of *DeepAuto*, we demonstrate how DNNs can be modeled as automata and be verified formally in the widely used model checker UPPAAL. The potential usefulness of *DeepAuto* shows the connection between DNNs and automata and provides a solution for the construction of more trustworthy DL systems.**

***Index Terms*—Formal Verification, Deep Neural Network, Timed Automata, DeepAuto**

I. Introduction

Deep Learning (DL) has enjoyed tremendous success over the past few years, achieving or exceeding human-level performance in various areas, including security-critical applications, such as autonomous vehicles [3], computer vision [6], speech recognition [10], robotics and competitive games such as Go [11]. However, DL systems often exhibit incorrect and unexpected behavior [12], carrying the risk of endangering human lives such as a fatal accident of self-driving car [9]. Thus, more concerns have been raised about the wide adoption of DL systems in security- and safety-critical systems.

To mitigate such concerns, one of the most active research areas in recent years is to design testing coverage for DL systems. Nevertheless, testing could be a quality metric for DL systems, but it cannot guarantee the correctness of the systems, which means numerous blunders may be concealed in DL systems and the testing approach cannot prove the absence of such errors. Towards addressing the aforementioned limitations, the application of formal verification techniques in DL systems appears to be a potential solution. In those traditional areas like software engineering, formal verification has achieved great success in guaranteeing that a system is free of certain defects or satisfies certain properties, and has played an important role in ensuring the correctness and reliability of increasingly complex software and hardware systems [4].

In the DL area, various types of DNNs have been widely used, such as Feed-forward Neural Networks (FNNs), Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs). Among these networks, a problem existing ubiquitously is that the output result may deviate from the expectation because of small perturbations to the input, which are so-called adversarial examples [7]. We need to ensure that DNNs deployed in security- and safety-critical domains satisfy expected properties against such infinitely many perturbations. In this work, we propose a formal framework named *DeepAuto*. It extracts the *data flow* of DNNs and name the *parameters* in DNNs, which results in expressible properties. It then models the *data flow* in automata, and converts the properties of DNNs into the properties of automata.

Since timed automata is a special type of automata and modeling RNNs needs to consider the temporal properties, we use timed automata for modeling DNNs in this paper. We link timed automata and DNNs, model the *data flow* process inside DNNs with timed automata. With the help of timed automata, we can observe the behavior inside the DNNs step-by-step and verify the properties of DNN-based systems. The model checker UPPAAL [2] is a tool designed to verify systems that can be modeled as timed automata. In UPPAAL, we can assign arbitrary value to input relying on demand. We use UPPAAL in our work to put *DeepAuto* into practice. Overall, our contributions are summarized as follows: (1) We introduce *DeepAuto*, the first white-box modeling and formal verification framework for DL systems. *DeepAuto* is practical and universal, and could be used to model trained DNNs. (2) *DeepAuto* provides a methodology to reduce the complex problem of formally verifying DNNs into a model checking problem which can be addressed by off-the-shelf model checkers such as UPPAAL. (3) Our work reveals the connection between DNNs (i.e., FNNs, CNNs and RNNs) and timed automata, and sheds light on applying formal verification to DL systems.

II. Data Flow and Timed Automata

To systematically ensure the correctness and reliability of DNNs in safety- and security-critical domains, it is crucial to apply formal verification techniques on DNNs. We first give the formal definition of DNNs as follows.

DOI reference number: 10.18293/SEKE2021-090.

Definition 1 (Deep Neural Network): A Deep Neural Network *is a tuple $N = (L, W, F, T)$ where $L = \langle l_1, l_2, \cdots, l_n \rangle$ is a sequence of layers and each layer l_k contains s_k neurons, $W = \langle w_1, w_2, \cdots, w_{n-1} \rangle$ is a sequence of matrices and the elements of matrices are weights between layers, $F = \langle f_2, f_3, \cdots, f_n \rangle$ is a sequence of* activation functions*, and T is the time variable which is needed in RNN.*

Intuitively, the properties of DNNs can be affected by W and F. To specify and verify properties of DNNs formally, we define the *data flow*, which is a record of variable changes within a DNN. Let $X = \langle X_1, X_2, \cdots \rangle$ be a sequence of variable vectors in each stage of the feed-forward process for a given DNN N, where X_1 is a set of variables in input stage and X_{i+1} is dependent on X_i within N. The definition of data flow is as follows.

Definition 2 (Data Flow): For a given DNN N and each X_i, there is a function ϕ_i mapping X_i to X_{i+1}, i.e., $X_{i+1} = \phi_i(X_i)$. We call $X_{i+1} = \phi_i(X_i)$ a data transformation. The sequence of all such data transformations is a data flow.

Since there are multiple mapping relationships between X_{i+1} and X_i in different DNNs, we use $X_{i+1} = \phi_i(X_i)$ to represent this data transformation process uniformly. *Data flow* records the transformation process of X, and we can investigate the interior of DNNs thoroughly based on *data flow*. We extract the *data flow* within DNNs and model DNNs as timed automata. Timed automata [1] are proposed to model the behavior of systems over time, expanding clock variables on the basis of finite-state automaton. It is decidable to verify whether states in timed automata are reachable.

Definition 3 (Timed Automaton): A timed automaton is a tuple (S, s_0, A, C, G, E), where S is a finite set of states, $S_0 \subseteq S$ is a set of initial states, A is a finite set of actions, C is a finite set of clocks, $I : S \rightarrow \Phi(C)$ assigns invariant to each state, and $E \subseteq S \times A \times 2^C \times \Phi(C) \times S$ is a set of transitions between states.

A switch $\langle s, a, \lambda, \varphi, s' \rangle$ represents an edge from state s to state s' with an action a, where φ is a clock constraint over C that specifies when the switch is enabled, and $\lambda \subseteq C$ gives the clocks to be reset with this transition.

Through the above definitions, we can easily derive the mapping relationships between DNNs and time automata. The update processes within DNNs can be mapped to update assignments (i.e., actions) in automata's edges, and the layers of DNN can be mapped to states of automaton.

III. Overview Of *DeepAuto*

An overview of the *DeepAuto* framework for modeling and verifying DNNs is shown in Fig. 1. We first use weights and activation function of a given DNN to construct the *data flow*. Then, the DNN is modeled by timed automaton, and its properties are formalized and converted into the properties of timed automaton. Finally, we can check whether the DNN meets expected properties. The pseudo-code for *DeepAuto* is given as in Algorithm 1.

Algorithm 1 takes the weights of DNN (the W), activation function of DNN (the F), structure information of DNN (e.g., the number of layers in DNN and the activation function of each layer) and expected property as inputs. In addition, circulation of time round is also needed for input when modeling RNN. Lines 1 to 6 of Algorithm 1 describe the abstract and model process, whose details are given later in Section IV. After obtaining the automaton, we can convert the properties and construct queries needed for the model checker, and finally use the model checker to query whether DNN meets the expected properties. If so, the DNN must satisfy the verified properties. Otherwise, users can analyze and debug the DNN according to unsatisfied properties.

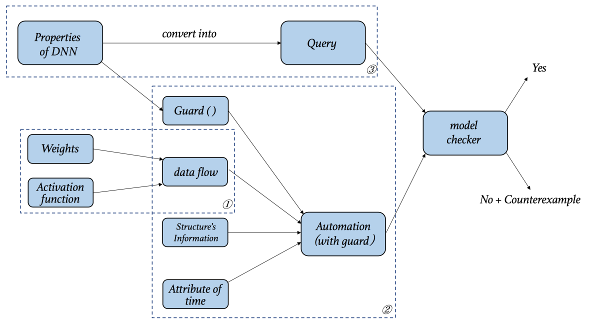

Fig. 1. Overview of *DeepAuto*

Algorithm 1 DeepAuto

```
Input: Weights of DNN $W$; Structure's information $S_l$; Activation functions of DNN $F$; Expected property $\phi$; Time information $t$ (optional, needed for RNN)
Output: Formal verification result r
1: $Data\_flow \leftarrow construct\_data\_flow(W, F)$
2: $Automaton \leftarrow Model(Data\_flow, S_l, t)$
3: if $Finish\_model(S_l, t) = true$ then
4:     $Guard() \leftarrow convert\_property\_into\_function(\phi)$
5:     $Automaton.add(Guard())$
6: end if
7: $Query \leftarrow convert\_property\_for\_automaton(\phi)$
8: r $\leftarrow Model\_checker(Automaton, Query)$
9: return r
```

IV. Modeling of DNN

More details of the modeling process are given in this section. Since commonly used DNNs (i.e., FNN, CNN, and RNN) have different internal structures, there are differences in the specific modeling processes.

A. Modeling of FNN

We use a FNN with a hidden layer as an example. The hidden layer has two neurons. The weights of the neuron in the input layer to the two neurons in the hidden layer are 1, -1 respectively. The rest of the weights are all 1. The activation function is ReLU function. When the input value is non-negative, the output value is always identical to the input value.

The given FNN can be modeled as automaton based on Algorithm 2. In all the algorithms of this work, the index of layer l_{i+1} is i. Using weights, structure information and activation

Algorithm 2 Modeling FNN (CNN) as automaton

```
Input: The weights of FNN (CNN) $W$; List of layers for FNN (CNN) $L = \langle l_1, l_2, \cdots, l_n \rangle$; Activation functions of FNN (CNN) $F = \langle f_2, f_3, \cdots, f_n \rangle$
Output: Abstract automaton $\mathbf{A_{NN}}$
 1: $node_1 \leftarrow L[0]$
 2: for index $i$ of layers do
 3:     if $L[i]$ is Convolution layer then
 4:         $Update \leftarrow X_{i+2} = W_i * X_{i+1}$
 5:     end if
 6:     if $L[i]$ is Max-pooling layer then
 7:         $Update \leftarrow X_{i+2} = Max(X_{i+1})$
 8:     end if
 9:     if $L[i]$ is Activation layer then
10:         $Update \leftarrow X_{i+2} = f_i(W_i \otimes X_{i+1})$
11:     else
12:         $Update \leftarrow X_{i+2} = W_i \otimes X_{i+1}$
13:     end if
14:     $\mathbf{T_{(i+1)(i+2)}} \leftarrow Update$ /*$\mathbf{T_{(i+1)(i+2)}}$ is the transition between $node_{i+1}$ and $node_{i+2}$.*/
15:     $node_{i+2} \leftarrow L[i+1]$
16: end for
17: $\mathbf{T_{(n+1)1}} \leftarrow Guard()$
18: return $\mathbf{A_{NN}}$
```

function, we can obtain the automaton shown in Fig. 2. In this example, $node_1$, $node_2$, and $node_3$ correspond to the states Input_layer, Hidden_layer, and Output_layer in Fig. 2 respectively. We construct equations, which are actually *data flows*, such as $X_{i+2} = f(W_i \otimes X_{i+1})$, $X_{i+2} = W_i \otimes X_{i+1}$. Each of these equations is assigned to the *Update* of a transition between states that are created based on the layer information of the FNN. Specifically, if *Update* corresponds to hidden layer, the *data flow* assigned should be $X_{i+2} = f(W_i \otimes X_{i+1})$. Otherwise, the *data flow* assigned should be $X_{i+2} = W_i \otimes X_{i+1}$. $\mathbf{Guard}()$ is a function with an if-else structure, whose returned type is **Bool**. We abstract the properties that DNNs need to satisfy into quantitative relationships by the aid of *data flow*. After generating such quantitative relationships, we encode them into the conditions of if-else statements, which enable us to transform the verification of DNNs' properties into the verification of automata's properties. The process of constructing $\mathbf{Guard}()$ is similar to constructing formal specifications in traditional software, and requires prior knowledge of the system.

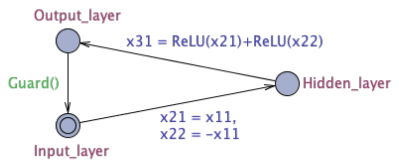

Fig. 2. FNN Model

B. Modeling of CNN

Algorithm 2 can also be used to model CNNs with the common structure (i.e., CNNs with convolution layers and max-pooling layers), which takes the weights of CNN, structure information and activation function as inputs and produces an automaton as output. The *data flows* in CNNs have forms like $X_{i+2} = f(W_i \otimes X_{i+1})$, $X_{i+2} = W_i * X_{i+1}$ or $X_{i+2} = Max(X_{i+1})$. According to the type of each layer, the *data flows* are assigned to the corresponding *transition Updates*. Specifically, the above-mentioned *data flows* correspond to activation layer, convolution layer and max-pooling layer respectively. The structure of automaton is constructed using structure's information.

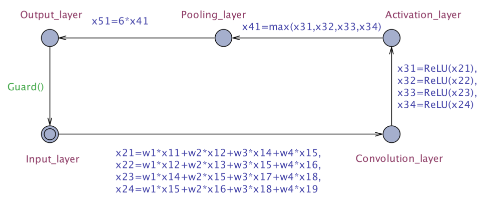

Fig. 3. CNN Model

$\mathbf{Guard}()$ is also a function with an if-then-else structure and its return type is **Bool**. Based on the properties of CNNs, we can obtain quantitative relationships. By encoding such relationships into the condition of the if-then-else statement, we transform the verification of CNNs' properties into the verification of automata's properties. We use a CNN with one convolution layer, one activation layer and one max-pooling layer as the running example. The corresponding automaton in UPPAAL is as shown in Fig. 3.

C. Modeling of RNN

Using Algorithm 3, we can model Vanilla RNNs of any size and simulate their running process, so as to check the properties. Algorithm 3 takes time round of RNN, RNN's weights, structure information and activation functions as inputs and produces an automaton simulating RNN as output. Based on the number of time round and weights within RNN, we can construct value update statements $Propagate_1(time)$, $\cdots$, and $Propagate_{n-1}(time)$. Whereafter, time update statement $time = time + 1$ and $Propagate_1(time)$ will be assigned to $\mathbf{T_{12}}$. Simultaneously, $Propagate2(time)$, $\cdots$, and $Propagate_{n-1}(time)$ will be assigned to the corresponding transitions. We use $t <= time \&\& time < t_0$ to control rounds of the RNN. The approach to construct $Guard()$ is the same as that in FNN and CNN modeling. By adding $Guard()$ to the transition, we can affect the property of automaton.

Here we use a three-cycle RNN as an example. We assume that weights of the RNN is $\langle [1], [1], [1] \rangle$ and activation function is $ReLU$. The time round t_0 is set as 3. To model such an RNN, we follow Algorithm 3. We first construct $Propagate1\ (time)$ and $Propagate2(time)$ to simulate the data processing. Time update statement $time = time + 1$ and $Propagate1(time)$ are assigned to the transition from $Input\_layer$ to $Hidden\_\ layer$, and $Propagate2(time)$ is assigned to the transition from $Hidden\_layer$ to $Output\_layer$. Using $t <= time \&\& time < t_0$, we limit the number of rounds to 3. After time 3, the transition from the state $Output\_layer$ to the state $Input\_layer$ will pass from the state $Success$ because of the invari-

Algorithm 3 Modeling RNN as automaton

```
Input: Time round of RNN $t_0$; The weights of RNN $W$;
    List of layers for RNN $L = \langle l_1, l_2, \cdots, l_n \rangle$; Activation
    functions of RNN $F = \langle f_2, f_3, \cdots, f_n \rangle$
Output: Abstract automaton $\mathbf{A_{RNN}}$
 1: $node_1 \leftarrow L[0]$
 2: $L.append(Success)$
 3: for Time round of RNN $t$ do
 4:     Construct $Propagate_1(time), \cdots, Propagate_{n-1}(time)$.
 5: end for
 6: for index $i$ of layers do
 7:     if $i == 0$ then
 8:         $\mathbf{T_{12}} \leftarrow time = time + 1, Propagate_1(time)$
 9:     else
10:         $\mathbf{T_{(i+1)(i+2)}} \leftarrow Propagate_{i+1}(time)$
11:     end if
12:     $node_{i+2} \leftarrow L[i+1]$
13: end for
14: $\mathbf{T_{(n+1)1}} \leftarrow t <= time \&\& time < t_0$
15: $\mathbf{T_{(n+1)(n+2)}} \leftarrow Guard()$
16: return $\mathbf{A_{RNN}}$
```

ant in $Output\_layer$ (i.e., $t \leqslant 3$). Finally, we assign $\mathbf{Guard}()$ to transition from $Output\_layer$ to $Success$. Now we get the model of the RNN (as shown in Fig. 4) and can do verification in the following. Note that the number of RNN rounds can be assigned by any value, which means our modeling methods can scale up and be used to model RNNs of any size.

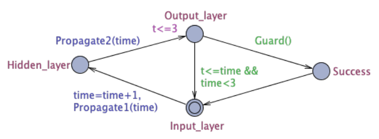

Fig. 4. RNN Model

For LSTM [5], We model the $forgotgate$, $inputgate$ and $outputgate$ into automaton. Using idea similar to Algorithm 3, update processes within LSTM can be abstracted and assigned to corresponding transitions between states. And we assign time update assignment $time = time + 1$ to transition between state $Wait\_for\_input$ and state $Forget\_gate\_and\_input\_gate\_updated$. Guard $t <= time \&\& time < t_0$, same as Algorithm 3, is constructed to control the number of rounds. Actually, Guard $t <= time \&\& time < t_0$ can be reduced to $time < t_0$. The We model a two-layer LSTM as the running example. The Automaton obtained is shown in Fig. 5.

V. Formal verification

In this section, we show how to verify the properties of DNNs. Intuitively, the properties of DNNs can be converted into the properties of automata. These properties can be verified by existing model checkers such as UPPAAL.

For FNN and CNN, because they have similar structures, methods of transforming their properties are inspired by the same idea. To be specific, we use the $Guard()$ function to control the operation of the automaton and encode the

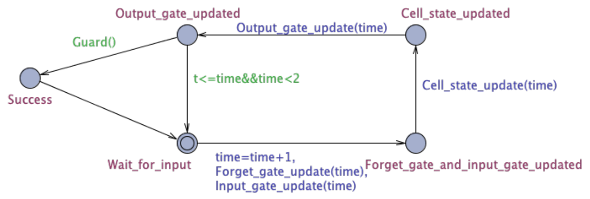

Fig. 5. LSTM Model

properties of FNN (or CNN) into the condition of if-else statement. The $Guard()$ is designed in such a way that it returns **True** when the properties are as expected, and **False** otherwise. In other words, if and only if FNN (or CNN) meets the property, the automaton is deadlock-free, otherwise the automaton will deadlock (i.e., stop on state Output_layer). As a result, we are able to verify the properties of FNN or CNN by using UPPALL to check whether the system is deadlock-free with the query $\mathbf{A}[\,]$ **not deadlock**.

For Vanilla RNN and LSTM, we construct $Guard()$ in a similar way. $Guard()$ returns **True** if the property of the RNN is as expected. Otherwise, $Guard()$ returns **False**. Naturally, when the RNN meets the expected property, the automaton always returns to the state $Success$ for both Vanilla RNN and LSTM. Otherwise, the automaton eventually stops in the state $Output\_layer$. With the help of the model checker, we can check the property "the automaton can always reach the state $Success$" with the query $\mathbf{E} <> \mathbf{Process.Success}$.

The traditional definition for local robustness based on distance is "for every input x_1 and x_2 such that $||x_1 - x_2||_\infty \leqslant \delta$, if the network is able to assign the same label to x_1 and x_2, then the network is robust." In UPPAAL, we can assign arbitrary value to input relying on demand. Thus, when exploring the robustness of the DNN, we could change inputs' value and observe whether the corresponding property (i.e., the output situation), could remain unchanged. If so, we could say the local robustness of the DNN for specific input could be proved.

VI. Case Studies

To examine the effectiveness of *DeepAuto*, we conduct case studies on both primitive operations and medium-sized DNNs. In our experiments, we extract the parameters required by modeling from the DNNs, and then use *DeepAuto* to model and verify them. To show the practicability of *DeepAuto*, there are two critical points that we need to highlight: (1) We can extract weights of trained DNNs based on Keras with TensorFlow. (2) The properties of DNNs can be converted into the properties of automata with the aid of data flow.

We start with a small-scale experiment in which we train FNNs which are capable of simulating $\wedge$ (and), $\vee$ (or), $\sim$ (not) and the Exclusive-OR gate. They are the four primitive operations the DL seminal work [8] uses neural networks to simulate. With *DeepAuto*, the weights of trained FNN can be automatically imported into UPPAAL, and then the formal models of these FNNs can be given automatically. Now we can obtain formal model of FNNs which are capable of simulating

all propositional forms with $\wedge$, $\vee$ and $\sim$. We can also model trained CNN and trained RNN as automaton automatically.

A standardized procedure for modeling and verifying a trained DNN is presented in Fig. 6. Following the procedure, we can model and verify the DNN before deployment, and verify whether the DNN is running according to its expected properties formally. When the DNN does not meet its expected properties, the automata can be used to observe the data flow inside the DNN, so as to point the way for debugging.

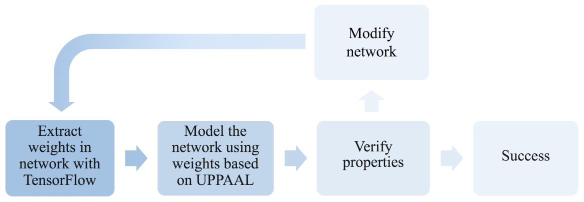
Modify network
Extract weights in network with TensorFlow
Model the network using weights based on UPPAAL
Verify properties
Success

Fig. 6. The Standardized Procedure for Modeling and Verifying a Trained DNN

Theoretically, *DeepAuto* can be scaled up to verify practical DNNs. However, UPPAAL doesn't accept real arithmetic. The weights of DNNs which we study in experiments are INT. We show *DeepAuto* can be scaled up to verify medium-sized DNNs. Several critical properties, as shown in Table I, can be verified based on *DeepAuto* framework. Property P_1 states that if the input is not perturbed by adversarial attacks, the output of the DNN will always be as expected. Property P_2 deals with the problem whether the output will be affected when the input is perturbed. Property P_3 "when we modify the internal weights, will the output be as expected?" probes into the interpretability of DNNs. As presented in Table I, we are able to verify all these properties based on the *DeepAuto* framework. The details can be found at https://github.com/Yuteng-Lu/UPPAAL_NN.

The concrete constructing and verifying processes of these three properties are as follows: (1) For property P_1, the expected relationship of input data and output data could be written in the $\mathbf{Guard}()$. (2) For property P_2, we could change input's value based on adversarial attack's behavior. After generating the perturbed input, we could check the properties of our formal model. If the properties change, we could say the output will be affected when the input is perturbed. (3) For property P_3, we can alter the formal model to modify the internal weights. After getting the formal model with modified weights, we could check its properties to indirectly check P_3.

TABLE I
CRITICAL PROPERTIES

	Specific Properties	Capabilities
P_1	When the input is not perturbed, the output is as expected.	✓
P_2	Whether the output will be affect when the input is perturbed.	✓
P_3	What happens to the output when the internal weights are changed.	✓

VII. Conclusion

In this paper, we propose the modeling and formal verification framework *DeepAuto* for DL systems. The UPPAAL model checker is used to implement *DeepAuto*. The experiments show that *DeepAuto* can dramatically guide modeling and verification procedure of DNNs, and the method covers the three most common DNN structures: FNNs, CNNs and RNNs. In addition, our work illustrates that there is a connection between the states of the timed automaton and the DNN's neurons. On the other hand, *DeepAuto* is actually a white-box verification framework. How can we do formal verification without all internal information (i.e., black-box or gray-box) will be considered in future work.

Acknowledgments.

This work was partially supported by the Guangdong Science and Technology Department (Grant no. 2018B010107004) and the National Natural Science Foundation of China under grant no. 61772038 and 61532019.

References

[1] R. Alur. Timed automata. In *Computer Aided Verification, 11th International Conference, CAV '99, Trento, Italy, July 6-10, 1999, Proceedings*, pages 8–22, 1999.

[2] G. Behrmann, A. David, and K. G. Larsen. A tutorial on uppaal. In *Formal Methods for the Design of Real-Time Systems, International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM-RT 2004, Bertinoro, Italy, September 13-18, 2004, Revised Lectures*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–236. Springer, 2004.

[3] B. Biggio and F. Roli. Wild patterns: Ten years after the rise of adversarial machine learning. *Pattern Recognition*, 84:317–331, 2018.

[4] E. M. Clarke Jr, O. Grumberg, D. Kroening, D. Peled, and H. Veith. *Model checking*. MIT press, 2018.

[5] F. A. Gers, J. Schmidhuber, and F. A. Cummins. Learning to forget: Continual prediction with LSTM. *Neural Computation*, 12(10):2451–2471, 2000.

[6] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, 2017.

[7] A. Kurakin, I. J. Goodfellow, and S. Bengio. Adversarial examples in the physical world. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Workshop Track Proceedings*, 2017.

[8] G. F. Miller, P. M. Todd, and S. U. Hegde. Designing neural networks using genetic algorithms. In *Proceedings of the 3rd International Conference on Genetic Algorithms, George Mason University, Fairfax, Virginia, USA, June 1989*, pages 379–384, 1989.

[9] K. Pei, Y. Cao, J. Yang, and S. Jana. Deepxplore: Automated whitebox testing of deep learning systems. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 1–18, 2017.

[10] H. Sak, A. W. Senior, and F. Beaufays. Long short-term memory recurrent neural network architectures for large scale acoustic modeling. In *INTERSPEECH 2014, 15th Annual Conference of the International Speech Communication Association, Singapore, September 14-18, 2014*, volume 10629 of *Lecture Notes in Computer Science*, pages 338–342. Springer, 2014.

[11] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. P. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[12] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. J. Goodfellow, and R. Fergus. Intriguing properties of neural networks. In *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*, 2014.