NEUROSPF: A tool for the Symbolic Analysis of Neural Networks

Muhammad Usman
University of Texas at Austin, USA
muhammadusman@utexas.edu

Yannic Noller
National University of Singapore
yannic.noller@acm.org

Corina S. Păsăreanu
Carnegie Mellon University, KBR Inc., NASA Ames
corina.s.pasareanu@nasa.gov

Youcheng Sun
Queen's University Belfast, UK
youcheng.sun@qub.ac.uk

Divya Gopinath
KBR Inc., NASA Ames
divya.gopinath@nasa.gov

***Abstract*—This paper presents NEUROSPF, a tool for the symbolic analysis of neural networks. Given a trained neural network model, the tool extracts the architecture and model parameters and translates them into a Java representation that is amenable for analysis using the Symbolic PathFinder symbolic execution tool. Notably, NEUROSPF encodes specialized peer classes for parsing the model's parameters, thereby enabling efficient analysis. With NEUROSPF the user has the flexibility to specify either the inputs or the network internal parameters as symbolic, promoting the application of program analysis and testing approaches from software engineering to the field of machine learning. For instance, NEUROSPF can be used for coverage-based testing and test generation, finding adversarial examples and also constraint-based repair of neural networks, thus improving the reliability of neural networks and of the applications that use them. Video URL: https://youtu.be/seal8fG78LI**

I. INTRODUCTION

Deep Neural Networks (DNNs) have gained popularity in recent years, being used in a variety of applications including banking, health-care, image and speech recognition, and perception in self-driving cars. With this widespread use of DNNs also come serious safety and security concerns. As a result, several techniques for testing [1], [2], [3] and verification [4], [5], [6] of neural networks have been developed recently, the majority of which have built dedicated tools.

In this work, we take a different approach, and we present NEUROSPF, which builds on a mature, widely used, program analysis tool, namely, Symbolic PathFinder (SPF) [7], to support analysis of neural networks, while leveraging the techniques that are already incorporated in SPF.

SPF [7] combines symbolic execution [8] with model checking for automated test case generation and error detection in Java byte-code programs. It supports both classical as well as concolic execution, it measures coverage and it is integrated with different constraint solvers, implementing also incremental analysis and solving – all these features could be useful for the analysis of neural networks as well. NEUROSPF extends SPF to support analysis of neural network models *efficiently*. To this end, NEUROSPF first translates a trained neural network model specified in Keras into Java and uses specialized peer classes to enable efficient parsing of the model's parameters. Furthermore, NEUROSPF enables users to make both the network inputs (e.g., input pixels for an image classifier) and the network parameters (i.e., weights and biases) symbolic, via special annotations. This flexibility opens up the possibility for several interesting applications.

For instance, NEUROSPF can be used for testing and test input generation with respect to coverage criteria that are relevant for neural networks [1], [9], [10], [11], [12]. This can be achieved by solving the relevant constraints collected by NEUROSPF. NEUROSPF can also be used for analyzing the robustness and for generating adversarial examples in neural networks, as studied in [13], [14], [15], which all propose specialized symbolic execution techniques for adversarial testing. Furthermore, the symbolic analysis in NEUROSPF can enable the automatic inference of neural network properties as advocated by Gopinath et al. [16]. These properties are network preconditions built based on the constraints collected with a symbolic analysis of the network. We also envision that NEUROSPF can enable automated repair for neural networks, by leveraging existing constraint-based repair techniques from the software engineering community [17], [18], [19], [20] and adapting them to the specifics of neural networks. We summarize our contributions as follows:

- We present NEUROSPF, a tool which facilitates the symbolic analysis of neural networks; NEUROSPF can handle feed-forward neural networks with dense, convolutional, and pooling layers, with ReLU activations and Softmax functions.
- To achieve efficient analysis, NEUROSPF encodes specialized peer classes for parsing and storing the model's parameters.
- We evaluate NEUROSPF on three neural networks (MNIST low quality, MNIST high quality, and CIFAR-10), showcasing NEUROSPF's ability of handling complex neural network models and highlighting the importance of using the peer classes.
- We also provide a detailed demonstration on how to use NEUROSPF, illustrating robustness analysis for a neural network model trained on the MNIST dataset.

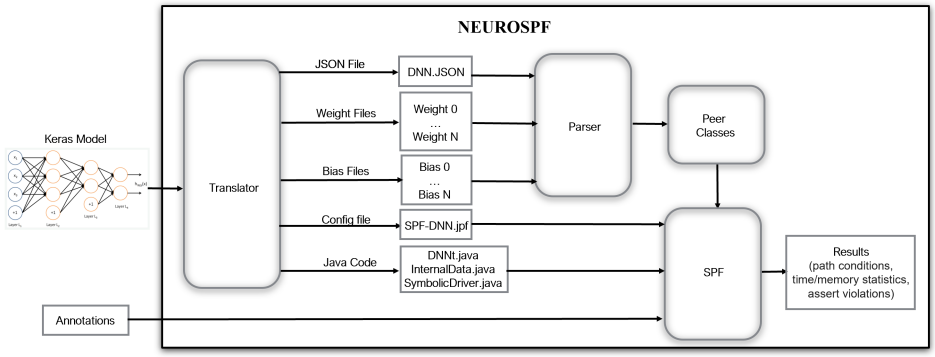

Fig. 1. Overview of NEUROSPF

The **envisioned users** for NEUROSPF include researchers and software engineers interested in applying symbolic execution for testing and debugging neural network models. The **challenge** we propose to address stems from the need to better understand and debug neural networks which are essentially black boxes. The **methodology** it implies for its users is described in detail in Section III. The results of preliminary **validation** are described in Section IV. We further plan for in-depth studies on the use of NEUROSPF in the applications that we outlined: testing, attack generation, property inference and automated repair for neural networks.

II. BACKGROUND

A. Neural Networks

Neural networks (NNs) [21] are machine learning algorithms that can be trained to perform different tasks such as classification and regression. NNs consist of multiple layers, starting from the *input* layer, followed by one or more *hidden* layers (such as convolutional, dense, activation, and pooling), and a final *decision* layer. Each layer consists of a number of computational units, called *neurons*. Each neuron applies an activation function on a weighted sum of its inputs; $N(X) = \sigma(\sum_i w_i \cdot N_i(X) + b)$ where N_i denotes the value of the i^{th} neuron in the previous layer of the network and the coefficients w_i and the constant b are referred to as *weights* and *bias*, respectively; σ represents the activation function. For instance, the ReLU (rectified linear unit) activation function returns its input as is if it is positive, and returns 0 otherwise, i.e., $\sigma(X) = max(0, X)$. The final decision layer (*logits*) typically uses a specialized function (e.g., max or *softmax*) to determine the decision or the output of the network.

B. Symbolic Execution and Symbolic PathFinder

In symbolic execution [8] a program is executed with symbolic (i.e., unspecified) inputs rather than concrete inputs. The goal is to generate mathematical constraints from the conditions in the program, which can be solved to generate test inputs. Symbolic PathFinder (SPF) [7] builds on top of Java PathFinder model checker to enable symbolic execution of Java bytecode programs. SPF can perform both standard symbolic execution and concolic execution, by collecting symbolic constraints along concrete executions. Both can be leveraged for neural network analysis [13], [14], [15].

```
1 double[] layer7=new double[128];
2 for(int i=0; i<128; i++)
3     if(layer6[i]>0) layer7[i]=layer6[i];
4     else layer7[i]=0;
```

Fig. 2. DNNt.java - Activation Layer (ReLU)

III. TOOL DESCRIPTION

1) Methodology: Figure 1 shows the overall framework of NEUROSPF. Users need to input the Keras model into the NEUROSPF. The *translator* component will generate: (1) a JSON file that provides critical information about the network model such as the dimensions of the weights and biases, number and types of layers, (2) *Weights and Biases files* that contain the values of the weights and biases for all layers, (3) *Config file* (SPF-DNN.jpf) which contains configuration settings for SPF i.e., minimum and maximum range of symbolic variables, type of constraint solver to be used etc. and (4) the *Java code* representation of Keras model.

The JSON file along with weights and biases files are the inputs to the *Parser* component. The parser reads the dimensions of each layer from the JSON file and loads the weights and biases from the respective files via specialized *Peer* classes. Once the files have been read via *Peer* classes, SPF is executed. The Java code along with the Config file form the input to SPF. The user has the option to edit the Config file according to their requirements.

The analysis can be further configured via user *annotations* which specify which inputs or parameters to be considered symbolic. Such a selection can be done based on attribution methods as described in previous work [13]. The output of the tool consists of the analysis results computed by SPF (assert violations, coverage information, time and memory statistics).

2) Translation to Java: Figure 2 shows the Java representation of an activation layer with ReLUs. Layer 7 is an array with size *128*. The *for loop* traverses through the output of previous layer i.e., layer 6. If the output is greater than 0, the neuron is activated by setting it to the output value of the

```
1 target=neurospf.SymbolicDriver
2 classpath=${jpf-symbc}/build/examples/
3 sourcepath=${jpf-symbc}/src/examples/
4 symbolic.min_double=0.0
5 symbolic.max_double=255.0
6 symbolic.dp=z3
```

Fig. 3. Config File (SPF-DNN.jpf)

```
1 Method to load image in image[28][28] array
2 InternalData internaldata = new InternalData();
3 DNNGeneralize.readDataFromFiles(path+"params\\",
      path+"dnn.json");
4 internaldata.biases0 = (double[]) DNNGeneralize.
      get_data("biases0");
5 ...
6 internaldata.weights0 = (double[][][][])
      DNNGeneralize.get_data("weights0");
7 ...
8 DNNt model=new DNNt(internaldata);
9 int label = model.run(image);
```

Fig. 4. SymbolicDriver.java

corresponding neuron of previous layer otherwise the value is set to 0.

The *translator* creates an *InternalData* class that contains arrays to store the weights and biases of neural network layers. *InternalData* also provides a function to read weights/biases using I/O libraries. This is helpful if the user wants to use the standalone Java code to run the neural network model without symbolic execution. Otherwise, *SymbolicDriver* reads the weights/biases files using specialized *Peer* classes for efficient symbolic execution. A JSON file is also generated. It encodes information about the architecture of the model.

As mentioned, the *translator* also generates a Config file (SPF-DNN.jpf) to specify configuration settings for SPF. Figure 3 shows a sample Config file. Line *1* specifies the target class to be executed using SPF. Lines *2* and *3* specify the classpath and sourcepath respectively. Lines *4* and *5* specify the minimum and maximum range of symbolic variables i.e., 0 and 255 respectively. Line *6* specifies the type of constraint solver to be used, i.e., Z3 [22].

3) Parser and Peer Classes: There are three inputs to the *Parser* component, i.e., the JSON file (which contains the description of the architecture of the neural network), weights and biases files for the neural network. The *parser* reads the JSON file and loads data from weights and biases files. NEUROSPF reads these files via *Peer* classes in SPF. This is a mechanism for executing Java code in the native VM instead of SPF's custom VM, which is much more efficient than to read files directly, as I/O operations significantly slow down the symbolic execution in SPF (see the next section for a comparison between NEUROSPF and plain (Vanilla) SPF). *Peers* avoid the bytecode interpretation in the custom VM and directly execute the Java bytecode.

DNNLayer is an abstract class and there are concrete classes for each type of layers. For example, *ActivationLayer*, *ConvolutionLayer* and *DenseLayer* are all included in NEUROSPF. There are specific member variables and methods for these layers depending on their functionality. This information is

```
1 image[15][15][0]=Debug.addSymbolicDouble(image
      [15][15][0],"sym_15_15");
2 ....
3 if(label!=8) {
4    Debug.getSolvedPC();
5    Debug.getSymbolicRealValue(image[15][15][0]);
6     assert(false);}
```

Fig. 5. SymbolicDriver.java - Code for Adversarial Generation

'8' '9'

Fig. 6. Label '8' → '9', by changing pixel([15][15]) value from 0 to 218

filled by parsing the JSON file. Specific layers for other neural network types can be added as needed. NEUROSPF currently supports Keras models but we plan to add support for other machine learning libraries in the future.

4) Symbolic Driver: The symbolic driver provides the main entry point for running the neural network. Figure 4 shows an example *SymbolicDriver* created for an image classifier.

IV. EVALUATION

In this section, we present the application of NEUROSPF in the popular field of adversary generation for neural network models. Specifically, we demonstrate how NEUROSPF can be used to generate adversarial examples for a neural network model trained on MNIST dataset (henceforth named MNIST-LowQuality). We also measure the run-time overhead incurred by the Java translation on neural networks trained for image classification (using MNIST and CIFAR-10 data sets) and compare the performance (execution time) of Vanilla SPF (SPF without the peer classes) and NEUROSPF.

Experiments were performed on a Windows 10.0 with Intel Core-i9 and 64GB RAM. The NEUROSPF tool along with neural network models are available at https://github.com/muhammadusman93/neurospf.

A. Robustness Analysis

We analyzed a trained MNIST model with NEUROSPF using a randomly selected input image (784 pixels). The image is represented using a 2D array of size [28][28]. For illustration purposes, we made one pixel (15,15) symbolic and added an assertion that triggers an error when the output class is not '8'. Figure 5 shows the sample annotations for making one pixel symbolic. NEUROSPF took 54 seconds to generate a counterexample, i.e., an adversarial image that led the neural network to change its output to label '9' from label '8'. Figure 6 shows the adversarial input found. This simple example demonstrates how NEUROSPF can be used to assess the neural network model's robustness to adversarial perturbations and to generate adversarial examples. A similar analysis is described in [14], which uses a dedicated symbolic execution tool, and modifies a small set of pixels (in some cases one or two). These pixels are discovered with an attribution analysis which can potentially be leveraged in NEUROSPF as well.

TABLE I
COMPARISON BETWEEN KERAS MODEL REPRESENTATION, JAVA CODE REPRESENTATION, VANILLA SPF AND NEUROSPF; "-" INDICATES TIME-OUT OF 1 HOUR

Model	Acc %	Time (s)			
		Keras	Java	SPF	NEUROSPF
MNIST-LowQ	96.0	0.1	0.2	-	34.1
MNIST-HighQ	100.0	0.1	0.2	2424.3	43.3
CIFAR-10	87.0	0.1	4.2	-	1908.1

B. Measuring Run-time Overhead

Table I compares the performance (execution time) of the following: (1) Keras model representation, (2) Java code representation, (3) Vanilla SPF (SPF without peer classes) and (4) NEUROSPF. Our experiments are based on the commonly used datasets MNIST and CIFAR-10. The MNIST models are 10-layer convolutional neural networks (CNNs) and have the typical structure of modern neural networks such as convolutional, dense, max-pooling and softmax layers. The CIFAR-10 model is a 15-layer CNN with 890k trainable parameters. The results show that all 4 settings give the same accuracy confirming the correctness of our implementation.

For MNIST-LowQuality model, the Keras representation is able to predict labels for 100 images in 0.1 seconds whereas the Java representation takes 0.2 seconds. Vanilla SPF times out while NEUROSPF takes just 34.1 seconds. For MNIST-HighQuality model, the Keras representation is able to predict labels for 100 images in 0.1 seconds whereas Java representation takes 0.2 seconds. Vanilla SPF takes 2424.3 seconds (40 minutes) while NEUROSPF takes just 43.3 seconds. For CIFAR-10 model, the Keras representation is able to predict labels for 100 images in 0.1 seconds whereas Java representation takes 4.2 seconds. Vanilla SPF times out while NEUROSPF takes 1908.1 seconds (32 minutes).

As expected, the Keras representation outperforms the Java representation in execution time but the major benefit of the Java representation is that existing software testing and verification techniques and tools for Java can be applied to deep learning models, with reasonable effort. Our results also show that NEUROSPF significantly outperforms the original SPF in terms of execution time. This is because NEUROSPF encodes specialized peer classes for efficient parsing of NN model parameters.

The results indicate that NEUROSPF is effective in executing neural networks with complex features (convolutional, dense, max-pooling, ReLU and softmax layers). They do not give an indication of the scalability of a symbolic analysis, which depends on the number of variables that are marked as symbolic and the number of symbolic paths and constraints that are generated.

We view NEUROSPF as an open source platform for researchers and software engineers who want to experiment with different symbolic analysis on neural networks using familiar languages and techniques. Simply running symbolic execution over the whole network (with all inputs symbolic) will likely not scale and specialized heuristics will be needed to enable NEUROSPF to perform specific analyses.

V. CONCLUSION

We presented the NEUROSPF tool that analyzes neural networks using Symbolic PathFinder. We showed how NEUROSPF can be used to check for adversarial robustness in neural networks. In the future, we plan to investigate other applications of NEUROSPF, such as test input generation, property inference and automated constraint-based repair.

REFERENCES

[1] 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*, 2017, pp. 1–18.

[2] Y. Tian, K. Pei, S. Jana, and B. Ray, "DeepTest: Automated testing of deep-neural-network-driven autonomous cars," in *ICSE*, 2018, pp. 303–314.

[3] H. F. Eniser, S. Gerasimou, and A. Sen, "DeepFault: Fault localization for deep neural networks," in *Fundamental Approaches to Software Engineering*, R. Hähnle and W. van der Aalst, Eds. Cham: Springer International Publishing, 2019, pp. 171–191.

[4] X. Huang, M. Kwiatkowska, S. Wang, and M. Wu, "Safety verification of deep neural networks," in *CAV*. Springer, 2017, pp. 3–29.

[5] G. Katz, C. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer, "Reluplex: An efficient smt solver for verifying deep neural networks," in *CAV*. Springer, 2017, pp. 97–117.

[6] L. Pulina and A. Tacchella, "An abstraction-refinement approach to verification of artificial neural networks," in *CAV*. Springer, 2010, pp. 243–257.

[7] C. S. Păsăreanu, W. Visser, D. H. Bushnell, J. Geldenhuys, P. C. Mehlitz, and N. Rungta, "Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis," *Autom. Softw. Eng.*, vol. 20, no. 3, pp. 391–425, 2013.

[8] J. C. King, "Symbolic Execution and Program Testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, jul 1976.

[9] L. Ma, F. Juefei-Xu, F. Zhang, J. Sun, M. Xue, B. Li, C. Chen, T. Su, L. Li, and Y. Liu, "DeepGauge: Multi-granularity testing criteria for deep learning systems," in *ASE*, 2018.

[10] Y. Sun, X. Huang, D. Kroening, J. Sharp, M. Hill, and R. Ashmore, "Structural test coverage criteria for deep neural networks," *ACM TECS*, vol. 18, no. 5s, pp. 1–23, 2019.

[11] J. Kim, R. Feldt, and S. Yoo, "Guiding deep learning system testing using surprise adequacy," in *ICSE*. IEEE, 2019, pp. 1039–1049.

[12] Y. Feng, Q. Shi, X. Gao, J. Wan, C. Fang, and Z. Chen, "DeepGini: Prioritizing massive tests to enhance the robustness of deep neural networks," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 177–188.

[13] D. Gopinath, C. Pasareanu, K. Wang, M. Zhang, and S. Khurshid, "Symbolic execution for attribution and attack synthesis in neural networks," *ICSE (ICSE-Companion)*, pp. 282–283, 2019.

[14] D. Gopinath, M. Zhang, K. Wang, I. B. Kadron, C. Pasareanu, and S. Khurshid, "Symbolic execution for importance analysis and adversarial generation in neural networks," in *ISSRE*, 2019, pp. 313–322.

[15] Y. Sun, M. Wu, W. Ruan, X. Huang, M. Kwiatkowska, and D. Kroening, "Concolic testing for deep neural networks," in *ASE*, ser. ASE 2018. New York, NY, USA: ACM, 2018, p. 109–119.

[16] D. Gopinath, H. Converse, C. Pasareanu, and A. Taly, "Property inference for deep neural networks," in *ASE*, 2019, pp. 797–809.

[17] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "SemFix: Program repair via semantic analysis," in *ICSE*, ser. ICSE '13. IEEE Press, 2013, p. 772–781.

[18] S. Ma, Y. Liu, W.-C. Lee, X. Zhang, and A. Grama, "MODE: automated neural network model debugging via state differential analysis and input selection," in *ESEC/FSE*, 2018, pp. 175–186.

[19] M. J. Islam, R. Pan, G. Nguyen, and H. Rajan, "Repairing deep neural networks: Fix patterns and challenges," *arXiv preprint arXiv:2005.00972*, 2020.

[20] J. Sohn, S. Kang, and S. Yoo, "Search based repair of deep neural networks," *arXiv preprint arXiv:1912.12463*, 2019.

[21] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[22] L. de Moura and N. Bjorner, "Z3: An efficient SMT solver," in *TACAS*, 2008.