DNNV: A Framework for Deep Neural Network Verification

David Shriver
dlshriver@virginia.edu
University of Virginia

Dong Xu
dx3yy@virginia.edu
University of Virginia

Sebastian Elbaum
selbaum@virginia.edu
University of Virginia

Matthew Dwyer
matthewbdwyer@virginia.edu
University of Virginia

ABSTRACT

Although many deep neural network (DNN) verification algorithms have been introduced in recent years, they use inconsistent input and output formats and provide varying support for DNN architectures. This makes it difficult to compare verifiers, to choose the best one for a task, and to re-use verification artifacts. In this work we present a toolset for running DNN verifiers using a common input and output format, facilitating application and comparison of DNN verifiers.

DNNV is available at https://github.com/dlshriver/DNNV. A video demonstration is available at https://youtube.com/XXXXXX.

KEYWORDS

verification, neural networks

1 INTRODUCTION

Deep neural networks (DNN) are being increasing applied to complex domains. As these techniques mature they are even being deployed in safety critical systems, such as autonomous driving [2, 5]. For such applications, it is often necessary to obtain behavioral guarantees about the safety of the system. To address this need, researchers have been actively exploring algorithms for verifying that the behavior of a trained DNN meets some correctness property. In the past two years alone, more than 20 DNN verification algorithms have been introduced [1, 3, 4, 6, 7, 9, 10, 13–23], and this number continues to grow. Many approaches also make their tools publicly availble, such as those shown in Table 1.

Unfortunately, several issues can make it difficult to utilize different verification tools on the same network and properties. This makes it difficult to compare verifiers, to choose the best one for a task, and to re-use verification artifacts. First, networks and properties vary greatly in how they are specified for each verifier. Many verifiers use their own custom input format, which can make the verification of new networks difficult, as they must be converted to non-standard formats. For example, *ReLuplex* uses a custom format, NNET, to specify networks, while *Planet* uses its own custom network format, RLV, which also includes a property specification. Second, verifiers differ in their support of network architectures. Most tools support fully connected layers, some support convolutional layers, and few support residual layers. For example, the *ReLuplex* verifier only supports fully connected layers, and requires all but the last layer to have ReLU activations. *MIPVerify* only supports a subset of convolutional layers and requires that they use a specific padding algorithm. Finally, verifiers are inconsistent in how they return and format results. All of these issues make it difficult to run and compare different verifiers.

In this work we introduce a new framework to reduce the burden of re-using artifacts, running DNN verification tools, and comparing their results for verifier researchers, developers and users. Our framework, DNNV, takes as input a network in a common, easily produced input format, a property written in an expressive domain-specific language for DNN properties, and the name of a verifier to run. DNNV transforms the network and property for the specified verifier and then runs the verifier to determine whether the property holds or not and returns results in a consistent format.

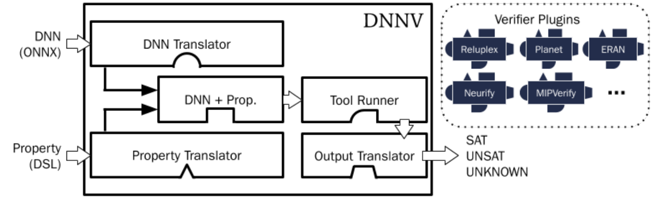

Figure 1: An overview of DNNV**.**

Next, we present an overview of DNNV and its use, discuss the implementation details, and present the results of a small study.

2 OVERVIEW

DNNV unifies DNN verifiers in a single framework with a common input format to facilitate verifier usage and comparison. In this section we discuss our property DSL and DNN input format, as well as the currently supported verifiers and basic usage. An overview of DNNV is shown in Figure 1. The framework takes in a DNN and a property, and the name of a verifier to run, selects a plugin for the specified verifier that applies input translators to the DNN and property specification, then runs the specified verifier and parses the results. The translators can make use of several utilities provided by DNNV to simplify plugin implementation.

2.1 Property DSL

Due to the lack of a standard format for specifying DNN properties, we develop a custom DSL for DNN properties. The goal was to develop a language that could be independent of a network being verified and be expressive enough to represent any property that can be verified by existing verification tools. The resulting property DSL is defined in Figure 2. It is implemented in the Python programming language and allows execution of arbitrary Python expressions. A property specification is a sequence of python module imports, followed by a sequence of assignments, and ends with a property expression, specified in a subset of first-order logic.

First, a property specification is allowed to import arbitrary python code, enabling users to re-use existing code for loading or processing input data. For example, the Python package numpy can be imported and used to load a dataset. Inputs can then be selected from the dataset, or statistics, such as the mean data point can be computed on the fly. Next, the DSL allows the definition of variables to be used in the final property specification. These assignments are not necessary, but allow for more readable property expressions by allowing repetitive expressions to be aliased. Finally,

Verifier	**Input Spec.**	Property Spec.	Layer Support	Property Support
Reluplex [11]	NNET (A)	Hard-coded	Fully connected layers; ReLU act.	Input interval constraints and linear inequalities over the output.
Planet [8]	RLV	Part of RLV	Fully connected, conv., residual, max pool layers; ReLU act.	Linear inequalities over the input and output.
ERAN [15, 16]	Custom PYT/TF	Hard-coded	Fully connected, conv., residual, max pool layers; ReLU, Sigmoid, Tanh act.	Local robustness.
Neurify [18]	NNET (B)	Hard-coded	Fully connected, conv. layers; ReLU act.	Local robustness
MIPVerify [17]	Model in Julia	Julia code	Fully connected, conv. (limited) layers; ReLu act.	Local robustness

Table 1: Currently supported verification tools.

⟨*property*⟩ ::= ⟨*python-imports*⟩ ⟨*assignment-list*⟩ ⟨*expr*⟩

⟨*python-imports*⟩ ::= "" | ⟨*python-imports*⟩ "import" ⟨*id*⟩
| ⟨*python-imports*⟩ "import" ⟨*id*⟩ "as" ⟨*id*⟩
| ⟨*python-imports*⟩ "from" ⟨*id*⟩ "import" ⟨*id*⟩

⟨*assignment-list*⟩ ::= "" | ⟨*assignment-list*⟩ ⟨*assignment*⟩

⟨*assignment*⟩ ::= ⟨*id*⟩ "=" ⟨*expr*⟩

⟨*expr*⟩ ::= "Forall(" ⟨*id*⟩ "," ⟨*expr*⟩ ")"
| "And(" ⟨*expr-seq*⟩ ")"
| "Or(" ⟨*expr-seq*⟩ ")"
| "Implies(" ⟨*expr*⟩ "," ⟨*expr*⟩ ")"
| ...
| ⟨*python-expr*⟩

⟨*expr-seq*⟩ ::= ⟨*expr*⟩ | ⟨*expr-seq*⟩ "," ⟨*expr*⟩

Figure 2: Grammar for the DNN property DSL.

the specification must end with a Boolean expression that defines the semantics of the property.

The property expression can make use of any python built-in functions or those from imported packages. The property DSL also provides operations for specifying properties in first order logic (e.g., Forall, Not, And, Or, Implies), creating symbolic variables (e.g., Symbol, Network, Image), and allowing external parameterization (e.g., Parameter).

An example of a property specifying local robustness is shown in Listing 1. This property defines a property over some network, N, and normalized image, x – discussed in §3. For every image that is within a hyperrectangle of radius $\frac{2}{255}$ centered at the image x, the network predicts the same class as the predicted class of x. Another property specification is shown in Listing 2. This is Property 1 of the ACAS network for aircraft collision avoidance [11]. The property specifies that when an intruding aircraft is more than 55947.69 feet away, and moving slower than 60 feet per second, and the velocity of the ownship is at least 1145 feet per second, then the output score of a Clear-Of-Conflict classification must be no more than 1500. The input constraints are specified as an interval over the input in lines 3-6.

2.2 DNN Format

As shown in Table 1, column 2, existing tools do not support a consistent, common input format. To enable support for specifying general deep neural network architectures, we choose to use an existing DNN format, ONNX [12]. ONNX is an open source format for representing DNNs with the ability to represent real-world

```
1 import numpy as np
2 N = Network("mnist_classifier")
3 x = (Image("image.npy") - 0.1307) / 0.3081
4 eps = Parameter("epsilon", default=2.0 / 255)
5 Forall(_x, Implies(x - eps < _x < x + eps,
6                    np.argmax(N(x)) == np.argmax(N(_x))))
```

Listing 1: Example of a local robustness property.

```
1 import numpy as np
2 N = Network("acas")
3 x_lb = np.array(
4         [55947.69, -3.14, -3.14, 1145.0, 0.0])
5 x_ub = np.array(
6         [60760.0, 3.14, 3.14, 1200.0, 60.0])
7 Forall(_x, Implies((x_lb <= _x <= x_ub), N(x_)[0] <= 1500))
```

Listing 2: Property 1 of the ACAS network [11].

networks. Many common frameworks (e.g., PyTorch, MXNet) allow exporting to ONNX natively, and conversion tools are available for most other frameworks (e.g., TensorFlow, Keras). While the ONNX format is expressive enough for any real-world DNN, our current implementation only supports a subset of the ONNX specification to handle all fully connected, convolutional, and residual networks that we have tried, including real-world networks such as VGG16 and ResNet34.

2.3 Verifier Support

DNNV currently includes support for 5 verification tools. These tools were selected based on the availability of their implementations. The DNN verifiers supported by the current version of DNNV are shown in the top half of Table 1.

We have made an effort to decrease the difficulty of integrating additional verification tools. Verifiers can be added by writing plugins for 3 primary components, as shown in Figure 1: an input translator (which includes a DNN translator plugin, a property translator plugin, and an DNN and property combination translation), a verifier runner, and an output translator. Adding a verifier involves writing a translator from our DNN and property formats to the input format of the new verifier, writing code to call the new verifier with the correct arguments, and parsing the verifier output to get the verification result.

In most cases the most difficult component to write is likely to be the translators for the DNN and property. To make development of

the translator easier, we provide several utilities for manipulating DNNs and properties.

First, we provide tools to parse and manipulate the property. The property is represented as an abstract syntax tree, which can be traversed using the visitor pattern. DNNV also provides utilities for simplifying and transforming the property AST. We provide utilities for performing constant propagation, as well as converting the property to conjunctive normal form.

Internally, DNNV represents DNNs as a graph of computations, such as matrix multiplications or convolutions, which can be traversed using the visitor pattern. Additionally, support is provided for performing pattern matching on the computation graph. Because most existing verification tools represent DNNs as a sequence of DNN layers, we also include a utility for converting from the computation graph format to a layer format. We include fully connected and convolutional layers in the base implementation, and allow additional layer types, such as residual layers to be specified by each tool. Each layer type is specified by a computation graph pattern. Layer types are matched to the end of the computation graph, with the longest match being chosen. All of the currently supported verifiers make use of the layer converter utility.

2.4 Usage

DNNV can be run from the command line with the following arguments: a DNN model in the ONNX format, a property written in the DNNV DSL, and the name of the verifier (or verifiers) to run. DNNV can be executed as follows:

```
python -m dnnv <dnn> <prop> <verifier>
```

Parameters specified in a property specification can be parameterized by supplying an option of the form `--prop.<param>`, where `<param>` is the name of the parameter. Additional options can be seen by specifying the `-h` option.

After execution, DNNV will report the results of verification. For each verifier, DNNV will report the verification result as one of SAT (if the property was falsified), UNSAT (if the property was proven to hold), UNKNOWN (if the verifier is incomplete and could not prove the property holds), or ERROR, along with the reason for error, if an error occurs during DNN and property translation, or during verifier execution. DNNV will also report the total time to translate and verify the property.

2.5 Limitations

DNNV currently has a few limitations in the properties and networks that it can handle. First, it currently only supports properties over a single network, with interval constraints over the input and output. However, this is powerful enough to specify many properties, including local robustness, by far the most commonly used DNN property. Additionally, our current implementation only supports a subset of the ONNX specification. However this is enough to handle all fully connected, convolutional, and residual networks that we have tried. Finally, translation does add some overhead (on the order of seconds for the networks in our study). Currently we do not attempt to otimize input translation, but future work could attempt to reduce this overhead, potentially by caching network translation results when verifying multiple properties.

Name	# Layers	# Neurons
ffnnRELU__Point_6_500	6	3010
convMedGRELU__Point	3	4804
convSmallRELU__Point	3	3604
mnist_conv_maxpool	9	13798
mnist_relu_3_50	3	160
mnist_relu_3_100	3	310
mnist_relu_4_1024	4	4106
mnist_relu_5_100	5	510
mnist_relu_6_100	6	610
mnist_relu_6_200	6	1210
mnist_relu_9_100	9	910
mnist_relu_9_200	9	1810

Table 2: The DNN to which DNNV **was applied.**

3 EVALUATION

We perform a simple study to demonstrate the ability of DNNV to apply a set of verifiers to a common set of DNN properties.

The networks and properties for this study were selected from those used in the evaluations of the *ERAN* tool. We selected a subset of the networks with ReLU activation functions. This resulted in 12 networks, shown in Table 2. The networks in the top of the table apply input pre-processing, while those in the bottom section do not. To account for this difference, we use 2 sets of properties, one for the top networks, and one for the bottom networks. The properties are the same, except for the application of input pre-processing. An example of a property for the first 3 networks in Table 2 is shown in Listing 1. The same property for the bottom 9 networks changes only line 3 of this property.

In addition to the networks, the *ERAN* tool includes 100 MNIST images for which it verifies local robustness. We select the first 10 images to create 10 local robustness properties to verify. Our local robustness properties used an epsilon of 0.008, which is within the range of epsilon values used by several verification tools [15–18].

For each of the 12 networks, we applied all 5 of the supported verification tools to check the validity of the 10 selected local robustness properties. Each verification task was allowed to run for up to 1 hour, with a memory limit of 64GB, and 4 processor cores.

The results of verification are shown in Table 3. Each row of the table are the results of one of the supported verification tools. The next 5 columns show the results by verification outcome: `UNSAT`, `SAT`, `UNKNOWN`, `TO`, and *other*. `TO` indicates that the tool exceeded the 1 hour time limit, while other indicates that the network or property are not supported by the verifier. For every result type, we report the number of properties for which the verify reported that result. We also report the mean time in seconds to verify the property for `UNSAT`, `SAT`, and `UNKNOWN` results.

As shown in Table 3, for 3 verifiers, DNNV reported that the verifier did not support the property. In the case of *ReLuplex*, 3 of the networks contain convolutional or max pooling layers, leading DNNV to report that 30 properties were not supported by the verifier. Similarly for *Neurify*, 1 of the networks contains max pooling layers, which are not supported. Finally, none of the properties specified are supported by *MIPVerify* due to an implicit assumption that inputs will always have values in the range [0, 1], which does not

Verifier	UNSAT		SAT		UNKNOWN		TO	*other*
	#	Time	#	Time	#	Time	#	#
ReLuplex	0	NA	0	NA	0	NA	90	30
Planet	0	NA	0	NA	0	NA	120	0
MIPVerify	0	NA	0	NA	0	NA	0	120
MIPVerify *	33+	133.13	0	NA	0	NA	10	22
Neurify	75	5.32	5	3.64	16	4.80	14	10
ERAN	72	4.81	0	NA	48	11.10	0	0

Table 3: Verification results for 10 properties across 12 networks using 5 verifiers. {DS: waiting for final MIPVerify* results}

hold for the specified properties. Because of this, DNNV correctly reports that the property is not supported by this verifier. In order to show *MIPVerify* verifying properties, we disable the input bounds check in the property translator and re-run verification. The results are reported in row 4. DNNV still reports errors on 22 property checks. Most of these are due to unsupported convolutional layers, but 2 occur during verification due to violation of the input bounds assumption.

As seen in Table 3, DNNV **facilitates the application of DNN verification tools by enabling users to write property specifications once for many verifiers.** For example, the same 10 property specifications could be used for 9 of the networks, across all of the verifiers in this study. DNNV can significantly simplify the process of running verification tools. For this study, we had a total of 12 network specifications and 20 property specifications. Previously, to run all 5 of the verifiers supported by DNNV would have required 60 network specifications (each verifier uses its own input specification), and at least 500 property specifications, since 3 of the verifiers specify networks and properties in a single input file.

4 CONCLUSION

In this work we present the DNNV toolset for verifying DNN. DNNV provides a common framework for running DNN verification tools. It utilizes a standard, easily produced DNN format to specify the network to be verified, and introduces a custom DSL to specify the property to verify. DNNV facilitates integration of verification tools and simplifies the application of verification tools to DNNs by both DNN developers and DNN verification researchers. It will lead to safer DNNs, as well as easier comparisons between verifiers and re-use of verification artifacts. Our experiments show that DNNV can significantly simplify the process of running verification tools by enabling specifications to be written once and used by many tools.

ACKNOWLEDGMENT

This material is based in part upon work supported by the NSF under Grant Number 1617916, 1900676, and 1901769, and the U. S. Army Research Laboratory and the U. S. Army Research Office under contract/grant number W911NF-19-1-0054-P00001.

REFERENCES

[1] Osbert Bastani, Yani Ioannou, Leonidas Lampropoulos, Dimitrios Vytiniotis, Aditya V. Nori, and Antonio Criminisi. 2016. Measuring Neural Net Robustness with Constraints. In *Proceedings of the 30th International Conference on Neural Information Processing Systems (NIPS'16)*. 2621–2629. http://dl.acm.org/citation.cfm?id=3157382.3157391
[2] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. 2016. End to End Learning for Self-Driving Cars. In *NIPS 2016 Deep Learning Symposium*.
[3] Akhilan Boopathy, Tsui-Wei Weng, Pin-Yu Chen, Sijia Liu, and Luca Daniel. 2019. CNN-Cert: An Efficient Framework for Certifying Robustness of Convolutional Neural Networks. In *AAAI*.
[4] Rudy R. Bunel, Ilker Turkaslan, Philip H. S. Torr, Pushmeet Kohli, and Pawan Kumar Mudigonda. 2018. A Unified View of Piecewise Linear Neural Network Verification. In *NeurIPS*. 4795–4804.
[5] F. Codevilla, M. Miiller, A. López, V. Koltun, and A. Dosovitskiy. 2018. End-to-End Driving Via Conditional Imitation Learning. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*. 1–9. https://doi.org/10.1109/ICRA.2018.8460487
[6] Souradeep Dutta, Susmit Jha, Sriram Sankaranarayanan, and Ashish Tiwari. 2018. Output Range Analysis for Deep Feedforward Neural Networks. In *NFM (Lecture Notes in Computer Science)*, Vol. 10811. Springer, 121–138.
[7] Krishnamurthy Dvijotham, Robert Stanforth, Sven Gowal, Timothy Mann, and Pushmeet Kohli. 2018. A dual approach to scalable verification of deep networks. In *Conference on Uncertainty in Artificial Intelligence (UAI-18)*. 162–171.
[8] Rüdiger Ehlers. 2017. Formal Verification of Piece-Wise Linear Feed-Forward Neural Networks. In *Automated Technology for Verification and Analysis - 15th International Symposium, ATVA 2017, Pune, India, October 3-6, 2017, Proceedings*. 269–286. https://doi.org/10.1007/978-3-319-68167-2_19
[9] T. Gehr, M. Mirman, D. Drachsler-Cohen, P. Tsankov, S. Chaudhuri, and M. Vechev. 2018. AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation. In *2018 IEEE Symposium on Security and Privacy (SP)*. 3–18. https://doi.org/10.1109/SP.2018.00058
[10] Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. 2017. Safety Verification of Deep Neural Networks. In *CAV (1) (Lecture Notes in Computer Science)*, Vol. 10426. Springer, 3–29.
[11] Guy Katz, Clark W. Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. 2017. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In *Computer Aided Verification, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*. 97–117. https://doi.org/10.1007/978-3-319-63387-9_5
[12] ONNX. 2017. Open Neural Network Exchange. https://github.com/onnx/onnx.
[13] Aditi Raghunathan, Jacob Steinhardt, and Percy Liang. 2018. Certified Defenses against Adversarial Examples. In *ICLR*. OpenReview.net.
[14] Wenjie Ruan, Xiaowei Huang, and Marta Kwiatkowska. 2018. Reachability Analysis of Deep Neural Networks with Provable Guarantees. In *IJCAI*. 2651–2659.
[15] Gagandeep Singh, Timon Gehr, Matthew Mirman, Markus Püschel, and Martin Vechev. 2018. Fast and Effective Robustness Certification. In *Advances in Neural Information Processing Systems 31*. 10802–10813. http://papers.nips.cc/paper/8278-fast-and-effective-robustness-certification.pdf
[16] Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin T. Vechev. 2019. An abstract domain for certifying neural networks. *PACMPL* 3, POPL (2019), 41:1–41:30.
[17] Vincent Tjeng, Kai Y. Xiao, and Russ Tedrake. 2019. Evaluating Robustness of Neural Networks with Mixed Integer Programming. In *International Conference on Learning Representations*. https://openreview.net/forum?id=HyGIdiRqtm
[18] Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. 2018. Efficient Formal Safety Analysis of Neural Networks. In *NeurIPS*. 6369–6379.
[19] Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. 2018. Formal Security Analysis of Neural Networks using Symbolic Intervals. In *USENIX Security Symposium*. USENIX Association, 1599–1614.
[20] Tsui-Wei Weng, Huan Zhang, Hongge Chen, Zhao Song, Cho-Jui Hsieh, Luca Daniel, Duane S. Boning, and Inderjit S. Dhillon. 2018. Towards Fast Computation of Certified Robustness for ReLU Networks. In *ICML (Proceedings of Machine Learning Research)*, Vol. 80. PMLR, 5273–5282.
[21] Eric Wong and J. Zico Kolter. 2018. Provable Defenses against Adversarial Examples via the Convex Outer Adversarial Polytope. In *ICML (Proceedings of Machine Learning Research)*, Vol. 80. PMLR, 5283–5292.
[22] W. Xiang, H. Tran, and T. T. Johnson. 2018. Output Reachable Set Estimation and Verification for Multilayer Neural Networks. *IEEE Transactions on Neural Networks and Learning Systems* 29, 11 (Nov 2018), 5777–5783. https://doi.org/10.1109/TNNLS.2018.2808470
[23] Huan Zhang, Tsui-Wei Weng, Pin-Yu Chen, Cho-Jui Hsieh, and Luca Daniel. 2018. Efficient Neural Network Robustness Certification with General Activation Functions. In *Advances in Neural Information Processing Systems 31, NeurIPS 2018, 3-8 December 2018, Montréal, Canada*. 4944–4953.