DNNV: A Framework for Deep Neural Network Verification

David Shriver(✉), Sebastian Elbaum, and Matthew B. Dwyer

University of Virginia, Charlottesville, VA, USA
{dls2fc,selbaum,matthewbdwyer}@virginia.edu

Abstract. Despite the large number of sophisticated deep neural network (DNN) verification algorithms, DNN verifier developers, users, and researchers still face several challenges. First, verifier developers must contend with the rapidly changing DNN field to support new DNN operations and property types. Second, verifier users have the burden of selecting a verifier input format to specify their problem. Due to the many input formats, this decision can greatly restrict the verifiers that a user may run. Finally, researchers face difficulties in re-using benchmarks to evaluate and compare verifiers, due to the large number of input formats required to run different verifiers. Existing benchmarks are rarely in formats supported by verifiers other than the one for which the benchmark was introduced. In this work we present DNNV, a framework for reducing the burden on DNN verifier researchers, developers, and users. DNNV standardizes input and output formats, includes a simple yet expressive DSL for specifying DNN properties, and provides powerful simplification and reduction operations to facilitate the application, development, and comparison of DNN verifiers. We show how DNNV increases the support of verifiers for existing benchmarks from 30% to 74%.

Keywords: Deep neural networks · Formal verification · Tool

1 Introduction

Deep neural networks (DNN) are being applied increasingly in complex domains including safety critical systems such as autonomous driving [3,7]. For such applications, it is often necessary to obtain behavioral guarantees about the safety of the system. To address this need, researchers have been exploring algorithms for verifying that the behavior of a trained DNN meets some correctness property. In the past few years, more than 20 DNN verification algorithms have been introduced [2,4,6,8–11,15,21,22,24–27,29–34,36], and this number continues to grow. Unfortunately, this progress is hindered by several challenges.

First, DNN verifier developers must contend with a rapidly changing field that continually incorporates new DNN operations and property types. While supporting more properties and operations may increase the applicable scope

A. Silva and K. R. M. Leino (Eds.) CAV 2021, LNCS 12759, pp. 137–150, 2021.
https://doi.org/10.1007/978-3-030-81685-8_6

Table 1. The network and property formats supported by each verifier. A * indicates that only a subset of the full input format specification is supported.

Verifier	Network format	Property format	Algorithmic approach
Reluplex [16]	*Reluplex*-NNET	Hard-coded	Search
Planet [10]	RLV	RLV	Search
BaB [6]	RLV	RLV	Search
BaBSB [6]	RLV	RLV	Search
MIPVerify [29]	MIPVerify Julia API	MIPVerify Julia API	Optimization
Neurify [30]	*Neurify*-NNET	Hard-coded	Search-optimization
DeepZono [25]	ONNX*, *ERAN*-PYT, *ERAN*-TF	*ERAN* Python API	Reachability
DeepPoly [26]	ONNX*, *ERAN*-PYT, *ERAN*-TF	*ERAN* Python API	Reachability
RefineZono [27]	ONNX*, *ERAN*-PYT, *ERAN*-TF	*ERAN* Python API	Reachability
RefinePoly [24]	ONNX*, *ERAN*-PYT, *ERAN*-TF	*ERAN* Python API	Reachability
Marabou [17]	*Reluplex*-NNET or ONNX*	*Marabou* Python API	Search
nnenum [1]	ONNX*	*nnenum* Python API	Search-reachability
VeriNet [14]	ONNX* or *Neurify*-NNET	*VeriNet* Python API	Search-optimization

of verifiers to real-world problems, it also increases a verifier's complexity. For example, for a verifier such as *DeepPoly*, supporting additional operations requires non-trivial effort to define and prove correctness of new abstract transformers. For verifiers such as *Reluplex* or *Neurify*, supporting new property types requires implementing a mapping from those properties onto internal verifier structures.

Second, DNN verifier users carry the burden of re-writing property specifications and transforming their models to match a chosen verifier's supported format. That burden is compounded by the diversity of input formats required by each verifier, as illustrated in Table 1. There is little overlap between input formats for verifiers (only *DeepZono* and *DeepPoly* or *BaB* and *BaBSB* which are algorithmically similar), and even when using the same format (as in the case of the popular ONNX format) we find that the underlying operations supported are different. This makes it difficult and costly to run multiple verifiers on a given problem since the user must understand the requirements of each verifier and translate inputs to their formats. While two new formats, VNNLIB [13] and SOCRATES [20], have been introduced in an attempt to standardize DNN verifier input formats, their expressiveness is currently limited and they can require writing new conversion tools for networks, as we discuss at the end of Sect. 3.1.

Finally, DNN verifier researchers face challenges in re-using benchmarks to evaluate and compare verifiers. Most benchmarks exist in the format of the verifier for which they were introduced, and running other verifiers on that benchmark requires writing custom tooling to translate the benchmark to other formats, or writing new input parsers for verifiers to support the given benchmark format. For example, the ACAS Xu benchmark (described in Sect. 5), was originally specified with networks in *Reluplex*-NNET format, and properties hard-coded into the verifier. The benchmark was converted, for example, into RLV format for *BaB* and *BaBSB*, as well as into ONNX with hard-coded properties

for *RefineZono*. Other benchmarks, such as the DAVE benchmark used by *Neurify*, has networks specified in *Neurify*-NNET, and properties hard-coded into the verifier. Due to its format, this potentially great benchmark has not been used by other verifiers.

We introduce a framework, DNNV*, to reduce the burden on verifier researchers, developers, and users.* DNNV helps to create and run more re-usable verification benchmarks by standardizing a network and property format, and it increases the applicability of a verifier to richer properties and real-world benchmarks by performing property reductions and simplifying DNN structures.

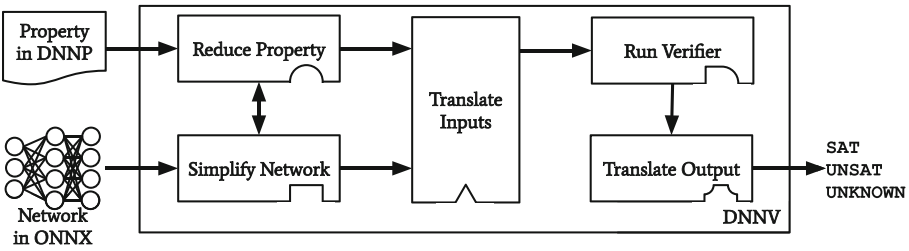

Fig. 1. DNNV architecture

As shown in Fig. 1, DNNV takes as input a network in the common ONNX input format, a property written in an expressive domain-specific language DNNP, and the name of a target verifier. Using the framework and plugins for the target verifier, DNNV transforms the problem by simplifying the network and reducing the property to enable the application of verifiers that otherwise would be unable to run. DNNV then translates the network and property to the input format of the desired verifier, runs that verifier on the transformed problem, and returns the results in a standardized format.

The primary contributions of this work are: (1) the DNNV framework to reduce the burden on DNN verifier researchers, developers, and users; DNNV includes a simple yet expressive DSL for specifying DNN properties, and powerful simplification and reduction operations to increase verifiers' scope of applicability, (2) an open source tool implementing DNNV[1], with support for 13 verifiers, and extensive documentation, and (3) an evaluation demonstrating the cost-effectiveness of DNNV to increase the scope of applicability of verifiers.

2 Background

A deep neural network $\mathcal{N}$ encodes an approximation of a target function $f : \mathbb{R}^n \to \mathbb{R}^m$. A DNN can be represented as a directed graph $G_{\mathcal{N}} = \langle V_{\mathcal{N}}, E_{\mathcal{N}} \rangle$, where nodes, $v \in V_{\mathcal{N}}$, represent operations and edges, $e \in E_{\mathcal{N}}$, represent input

[1] https://github.com/dlshriver/DNNV.

arguments to operations. A node without any incoming edges is an input to the DNN. The output of a DNN can be computed by looping over nodes in topological order and computing the value of the node given its inputs. The literature on machine learning has developed a broad range of rich operation types and explored the benefits of different combinations of operations in realizing accurate approximations of different target functions, e.g., [12].

Given a DNN, $\mathcal{N} : \mathbb{R}^n \rightarrow \mathbb{R}^m$, a property, $\phi(\mathcal{N})$, defines a set of constraints over the inputs, $\phi_{\mathcal{X}}$ – the pre-condition, and a set of constraints over the outputs, $\phi_{\mathcal{Y}}$ – the post-condition. Verification of $\phi(\mathcal{N})$ seeks to prove or falsify: $\forall x \in \mathbb{R}^n : \phi_{\mathcal{X}}(x) \rightarrow \phi_{\mathcal{Y}}(\mathcal{N}(x))$.

A widely studied class of properties is *robustness*, which originated with the study of adversarial examples [28,35]. These properties specify that inputs from a specific region of the input space must all produce the same output class. Detecting violations of robustness properties has been widely studied, and they are a common type of property for evaluating verifiers [10,25,26,29,30]. Another common class of properties is *reachability*, which define the post-condition using constraints over output values. Reachability properties specify that inputs from a given region of the input space must produce outputs within a given region of the output space. Such properties have been used to evaluate several DNN verifiers [16,17,30].

A recent survey on DNN verification [18] classifies these approaches based on their type: reachability, optimization, or search, or a combination of these. Reachability-based methods compute a representation of the reachable set of outputs from an encoding of the set of inputs that satisfy the pre-condition. The computed output set is often an over-approximation of the true reachable output region. The precision of the computed output region depends on the symbolic representation used, e.g., hyper-rectangles, zonotopes, polyhedra. Reachability-based methods include [11,22,24–27,34]. Optimization-based methods formulate property violations as a threshold for an objective function and use optimization algorithms to attempt to satisfy that threshold. Optimization-based methods include [2,9,21,29,33]. Search-based methods explore regions of the input space where they then formulate reachability or optimization sub-problems. Search-based methods include [6,10,15,16,31,32].

3 DNNV Overview

DNNV remedies several key challenges faced by the DNN verification community. A general overview of DNNV is shown in Fig. 1. DNNV takes in a property and network in a standard format, simplifies the network, reduces the property, translates the network and property to the input format of the verifier, runs the verifier, and translates its output. Each of these components can be customized by verifier specific plugins. We explain these components in more detail below.

3.1 Input Formats

As shown in Table 1, existing verifiers do not support a consistent, common input format for networks and properties. DNNV standardizes the input and output formats to aid the community in creating and running verification benchmarks.

Table 2. The number of ONNX operations supported by each verifier.

Verifier	# ONNX operations
DNNV	31
ERAN	22
nnenum	15
marabou	12
VeriNet	12

ONNX. For specifying general deep neural network architectures, we choose the open source DNN format ONNX [19]. ONNX can represent real-world networks, is supported by many common frameworks (e.g., PyTorch, MXNet) and conversion tools are available for other frameworks (e.g., TensorFlow, Keras). Our current implementation supports a subset of the ONNX specification that subsumes the subsets of ONNX implemented by the supported verifiers. Table 2 shows the number of ONNX operations supported by each of the verifiers included in DNNV. DNNV supports 40% more operations than the verifier with the next highest support. The ONNX subset supported by DNNV is sufficient for almost all existing verification benchmarks, as well as many real-world networks including VGG16 and ResNet34.

DNNP. Due to the lack of a standard format for specifying DNN properties, we develop a Python-embedded DSL for DNN properties, which we call DNNP. DNNP is designed to express any property that can be verified by existing DNN verifiers in a form that is independent of the network. DNNP is described in more detail in Appendix A of the extended version of this paper [23].

```
1  from dnnv.properties import *
2  from torchvision.datasets import FashionMNIST
3  from torchvision.transforms import ToTensor
4
5  N = Network("N")
6  data = FashionMNIST("/tmp", download=True,
7                      transform=ToTensor())
8  mean = 0.2860
9  std = 0.3530
10 i = Parameter("data_idx", type=int, default=1)
11 x = (data[i][0][None, :].numpy() - mean) / std
12 e = Parameter("epsilon", type=float) / std
13
14 Forall(
15  x_,
16  Implies(
17   And(
18    (-mean / std) <= x_ <= ((1 - mean) / std),
19    (x - e) < x_ < (x + e),
20   ),
21   argmax(N(x_)) == argmax(N(x)),
22  ),
23 )
```

Fig. 2. Example of a local robustness property specified with DNNP.

We demonstrate DNNP with an example of a local robustness property, shown in Fig. 2. The property specifies that, for all inputs, `x_` (Lines 14–23), in the input space (Line 18) and within a hyper-rectangle of radius `e` centered at the given input `x` (Line 19), the network should predict the same maximum class for both `x_` and `x` (Line 21). For Fashion MNIST, this means that for all images within an L_∞ distance of e (specified on Line 12) from

image 1 of the dataset (selected on Lines 10–11), the network should classify all of these images the same as it does for image 1. We first import several Python packages that will be useful for specifying the property (Lines 1–3), including the dataset used to train the network, and a method for data manipulation. Because DNNP allows importing arbitrary Python packages, it enables re-use of the same data loading and manipulation methods used to train a network. After importing the necessary utilities, we define several variables that will be used in the final property expression (Lines 5–12). Two of these variables, i on Line 10 and e on Line 12 are declared as parameters, which allows them to be specified on the command line at run time. The value for e must be provided at run time, since no default value is provided. Finally, we define the semantics of the property specification, using methods provided by DNNP, as well as variables defined above (Lines 14–23).

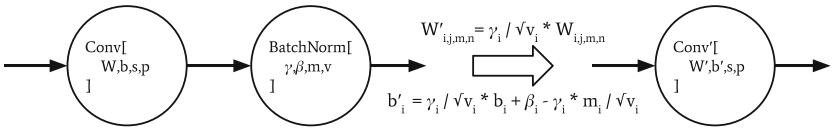

Fig. 3. Batch Normalization Simplification simplifies a batch norm following a convolution operation to an equivalent single convolution operation with modified weights and bias, while maintaining the strides and pads.

Other Input Formats. Since the creation of DNNV, two new input formats, VNNLIB [13] and SOCRATES [20], have emerged in an attempt to standardize the verifier input space. The current draft of VNNLIB also uses ONNX as the DNN input format, however it supports a much smaller set of operations than DNNV, supporting only 17 ONNX operations. The VNNLIB property format is a subset of SMTLIB in which variables of the form X_i are implicitly mapped to network inputs and variables of the form Y_i are implicitly mapped to network outputs. In its current form, this specification only supports DNN models with a single flat input tensor and single flat output tensor, whereas DNNP and ONNX can support DNN models with multiple inputs and output tensors of any shape. SOCRATES proposes a JSON format containing both the property and network specifications. Because DNNV treats networks and properties independently, properties can be re-used for multiple networks, and only a single network must be stored to check multiple properties, resulting in a lower storage cost, especially for large models. Additionally, while the custom JSON format used by SOCRATES requires new DNN translation tools to be written to convert to the required format, the ONNX format used by DNNV is commonly available in most machine learning frameworks. While we believe that ONNX and DNNP are currently the most expressive and easily accessible input formats currently proposed, DNNV can provide benefits to any format through DNN simplification and property reduction to increase the applicability of all verifiers.

3.2 Network Simplification

In order to allow verifiers to be applied to a wider range of real world networks, DNNV provides tools for network simplification. Network simplification takes in an operation graph and applies a set of semantics preserving transformations to the operation graph to remove unsupported structures, or to transform sequences of operations into a single more commonly supported operation.

An operation graph $G_{\mathcal{N}} = \langle V_{\mathcal{N}}, E_{\mathcal{N}} \rangle$ is a directed graph where nodes, $v \in V_{\mathcal{N}}$ represent operations, and edges $e \in E_{\mathcal{N}}$ represent inputs to those operations. Simplification, $simplify : \mathcal{G} \to \mathcal{G}$, transforms an operation graph $G_{\mathcal{N}} \in \mathcal{G}$, to an equivalent DNN with more commonly supported structure, $simplify(G_{\mathcal{N}}) = G_{\mathcal{N}'}$, such that the resulting DNN has the same behavior as the original $\forall x . \mathcal{N}(x) = \mathcal{N}'(x)$, and uses more commonly supported structures.

One such simplification is *batch normalization simplification*, which removes batch normalization operations from a network by combining them with a preceding convolution operation or generalized matrix multiplication (GEMM) operation. This is possible since batch normalization, convolution, and GEMM operations are all affine operations. The simplification of a batch normalization operation following a convolution operation is shown in Fig. 3. If no applicable preceding layer exists, the batch normalization layer is converted into an equivalent convolution operation. This simplification enables the application of verifiers without explicit support for batch normalization operations, such as *Neurify* and *Marabou*, to networks with these operations.

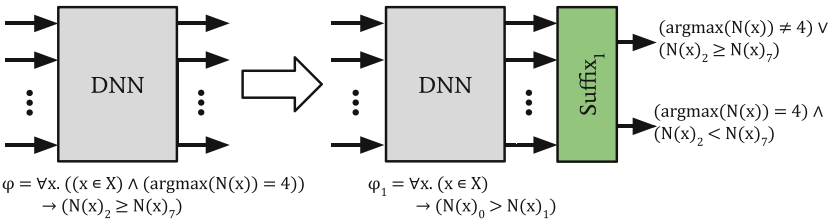

Fig. 4. Property reduction to a local robustness property adds a suffix that classifies outputs as violations or non-violations of the original output constraints, and changing the property to a common form of robustness property.

DNNV currently includes 6 additional DNN simplifications, enumerated and described in more detail in Appendix B of the extended version of this paper [23].

3.3 Property Reduction

In order to allow verifiers to be applied to more general safety properties, DNNV provides tools to reduce properties to a supported form. For instance, properties can be translated to local robustness properties, which are required by *MIPVerify* or reachability properties which are required by *Reluplex*.

Property reduction takes in a verification problem, which is comprised of a property specification and a network, and encodes it as an equivalid set of verification problems with properties in a form supported by a given verifier.

A *verification problem* is a pair, $\psi = \langle \mathcal{N}, \phi \rangle$, of a DNN, $\mathcal{N}$, and a property specification ϕ, formed to determine whether $\mathcal{N} \models \phi$ is *valid*. Reduction, $reduce : \Psi \rightarrow P(\Psi)$, aims to transform a verification problem, $\langle \mathcal{N}, \phi \rangle = \psi \in \Psi$, to an equivalid form, $reduce(\psi) = \{\langle \mathcal{N}_1, \phi_1 \rangle, \ldots, \langle \mathcal{N}_k, \phi_k \rangle\}$, in which property specifications are in a common supported form. As defined, reduction has two key properties. The first property is that the set of resulting problems is equivalid with the original verification problem. The second property is that the resulting set of problems all use the same property type. Applying reduction enables verifiers to support a large set of verification problems by implementing support for a single property type.

For example, given a network that classifies images of clothing items, a user may want to specify that, if the network classifies an image as a coat, then the score given to the class of a pullover is not less than the score for the sneaker class. The property is specified in the bottom left of Fig. 4. Such a verification problem can be difficult to specify for many verifiers. For example, *Neurify* would require writing code to specify linear constraints for the property and re-compiling the verifier, and *MIPVerify* cannot support this property as is. DNNV can reduce this verification problem to an equivalent problem with a robustness property.

A high level overview of this reduction is shown in Fig. 4; a more detailed description is provided in Appendix C of the extended version of this paper [23].

3.4 Input and Output Translation

Because of the large variety of input formats required by the verifiers, one of the primary components of DNNV translates from its internal representation of properties and networks to the input formats of each verifier.

DNNV also requires an output translator that can parse the results of running a verifier and returns `sat`, `unsat`, or `unknown`. If the result is `sat`, indicating a violation was found, DNNV also returns a counter example to the property, and validates that it does violate the property by performing inference with the network and confirming that the input and output do not satisfy the property.

4 Implementation

DNNV is written in 8400 lines of Python code and is available for download and re-use at https://doi.org/10.5281/zenodo.4883626. Python was chosen due to its ubiquitous use for developing deep neural networks. DNNV currently supports 13 verifiers, and was designed to facilitate the integration of new verifiers. The currently supported verifiers are shown in Table 1, along with their original input formats, and algorithmic approach. Around 2000 LOC (of the 8400 total LOC) are used to integrate these 13 verifiers into DNNV, with *Planet* requiring the most effort at 437 lines, and *BaB* and *BaBSB* requiring the least effort with 89 lines of code due to re-use of the *Planet* input translator.

4.1 Supporting Reuse and Extension

DNNV is designed to facilitate the integration of new verifiers. The 5 primary components of DNNV, DNN simplification, property reduction, input translation, verifier execution, and output translation are designed to be re-usable, and to facilitate the implementation of new components by providing utilities for traversing and manipulating operation graphs and properties.

Networks are represented as an operation graph, where nodes represent operations in the DNN and edges represent inputs and outputs to those operations. The operation graph can also be traversed using a visitor pattern. This pattern is particularly useful for the development of DNN simplifications and input translators. It allows developers to easily traverse computation graphs in order to translate operations to the required format. We provide built-in utilities for converting from our internal network representation to ONNX, PyTorch, and TensorFlow models. The implementation also includes utilities for performing pattern matching on operation graphs. We utilize this feature to provide utilities that transform a network from an operation graph representation to a sequential layer representation, which is particularly useful for the network input translator of *Neurify*, which requires DNNs to have a regular structure of a set of convolutional layers followed by fully connected layers, all with relu activations.

4.2 Usage

DNNV can be run from the command line as follows: `python -m dnnv <prop> <verifier> --network <name> <path>`, where the arguments correspond to a DNN model in the ONNX format, a property written in DNNP, and the verifier to run. Many additional options can be seen by specifying the `-h` option.

After execution, for each verifier, DNNV reports 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 also reports the time to translate and verify the property.

5 Study

We now examine the applicability of verifiers to existing verification benchmarks with and without DNNV. A verification benchmark consists of a set of verification problems which are used to evaluate the performance of a verifier. A problem is made of a DNN and a property specification and asks whether the property is valid for the given DNN. We consider a verifier to support a benchmark if it can be run on that benchmark out of the box. We consider a verifier to have support for a benchmark through DNNV if DNNV can be run on that benchmark with networks specified using ONNX and properties specified in DNNP, and can reduce, simplify, and translate the problem to work with the target verifier.

Benchmarks. To evaluate benchmark support, we collected the benchmarks used by each of the 13 verifiers supported by DNNV, and determined whether each verifier can run on the benchmark out of the box, and also whether they could be run on the benchmark when DNNV is applied. The verification benchmarks are shown in Table 3 and are also described in more detail in Appendix D of the extended version of this paper [23]. Each row of the table corresponds to a benchmark, to which we assign a short key for identifying the benchmark. For each benchmark, we give the name, some of the verifiers it evaluated, the number of properties (#P) and networks ($\#\mathcal{N}$), and features that can make it challenging for verifiers. These features include whether any properties cannot represent their input constraints using hyper-rectangles (¬HR), whether any network in the benchmark contains convolution operations (C), whether any network contains residual structures (R), and whether any network uses any non-ReLU activation functions (¬ReLU).

Results. The support of verifiers for each benchmark is shown in Table 4. Each row of this table corresponds to one of the 13 verifiers supported by DNNV, and each column corresponds to one of the 19 benchmarks identified in Table 3. Each

Table 3. Verifier benchmarks.

Key	Name	Uses	#P	$\#\mathcal{N}$	Features			
					¬HR	C	R	¬ReLU
AX	ACAS Xu	[1,6,16,17,30]	10	45				
CD	Collision Detection	[6,10,17]	500	1				
PM	*Planet* MNIST	[10]	7	1	✓	✓		
TS	TwinStream	[5]	1	81				
PCA	PCAMNIST	[6]	12	17				
MM	*MIPVerify* MNIST	[29]	10000	5		✓		
MC	*MIPVerify* CIFAR10	[29]	10000	2		✓	✓	
NM	*Neurify* MNIST	[14,30]	500	4		✓		
NDB	*Neurify* Drebin	[30]	500	3				
NDV	*Neurify* DAVE	[30]	200	1	✓	✓		
DZM	*DeepZono* MNIST	[25]	1700	10		✓	✓	✓
DZC	*DeepZono* CIFAR10	[25]	1700	5		✓		✓
DPM	*DeepPoly* MNIST	[14,26]	1500	8		✓		✓
DPC	*DeepPoly* CIFAR10	[26]	800	5		✓		
RZM	*RefineZono* MNIST	[27]	800	8		✓		
RZC	*RefineZono* CIFAR10	[27]	200	2		✓		
RPM	*RefinePoly* MNIST	[24]	600	6		✓		
RPC	*RefinePoly* CIFAR10	[24]	300	3		✓	✓	
VC	*VeriNet* CIFAR10	[14]	250	1		✓		

Table 4. Benchmark support by each verifier. The left half of the circle is black if the verifier can support the benchmark out of the box, and is white otherwise. The right half is black if the verifier supports the benchmark through DNNV, and is white otherwise. An absent circle indicates that the verifier can not be made to support some aspect of the benchmark.

	Benchmark																		
Verifier	AX	CD	PM	TS	PCA	MM	MC	NM	ND_B	ND_V	DZM	DZC	DPM	DPC	RZM	RZC	RPM	RPC	VC
Reluplex	●	◐	○	●	●	○	○	○	◑	○				○	○	○	○	○	○
Planet	●	◐	●	●	●	◑	◑	◑	◑	◑				◑	◑	◑	◑	◑	◑
BaB	●	◐	●	●	●	◑	○	◑	◑	◑				◑	◑	◑	◑	○	◑
BaBSB	●	◐	●	●	●	◑	○	◑	◑	◑				◑	◑	◑	◑	○	◑
MIPVerify	◑	○	○	◑	◑	●	◐	○	◑	○				○	○	○	○	○	○
Neurify	●	○	◑	◑	◑	◑	○	●	●	●				◑	◑	◑	◑	○	◑
DeepZono	●	○	○	◑	◑	◑	◑	◑	◑	○	●	●	●	●	●	●	●	●	◑
DeepPoly	●	○	○	◑	◑	◑	◑	◑	◑	○	●	●	●	●	●	●	●	●	◑
RefineZono	●	○	○	◑	◑	◑	◑	◑	◑	○	●	●	●	●	●	●	●	●	◑
RefinePoly	●	○	○	◑	◑	◑	◑	◑	◑	○	●	●	●	●	●	●	●	●	◑
Marabou	●	◐	◑	●	●	◑	○	◑	◑	◑				◑	◑	◑	◑	○	◑
nnenum	●	○	◑	◑	◑	◑	○	◑	◑	◑				◑	◑	◑	◑	○	◑
VeriNet	◑	○	○	◑	◑	◑	○	●	◑	○	○	◑	◑	◑	◑	◑	◑	○	●

cell of the table may contain a circle that identifies the support of the verifier for the benchmark. The left half of the circle is black if the verifier can support the benchmark out of the box, and is white otherwise. The right half is black if the verifier supports the benchmark through DNNV, and white otherwise. An absent circle indicates that the verifier can not be made to support some aspect of the benchmark. For the benchmarks shown here, this is always due to the presence of non-ReLU activation functions in some of the networks in the benchmarks.

As shown in Table 4, DNNV can dramatically increase the support of verifiers for benchmarks. For example, the *Planet* verifier could originally be run on 5 of the 19 benchmarks, but could be run on 16 using DNNV. Similarly, the *nnenum* verifier, could originally only be run on 1 of the existing benchmarks, but could be run on 13 using DNNV. **Of the 223 pairs of verifiers and benchmarks for which support may be possible, 166 of them are currently supported by DNNV, an increase of over 2.4 times the 68 pairs supported without DNNV.**

6 Conclusion

We present the DNNV framework for reducing the burden on DNN verifier researchers, developers, and users. DNNV standardizes input and output formats, includes a simple yet expressive DSL for specifying DNN properties, and provides powerful simplification and reduction operations to facilitate the application, development, and comparison of DNN verifiers. Our study showed the potential of DNNV and we made its implementation available, with support for 13 verifiers, and extensive documentation.

Acknowledgment. This material is based in part upon work supported by the National Science Foundation under Grant Number 1900676 and 2019239.

References

1. Bak, S., Tran, H.-D., Hobbs, K., Johnson, T.T.: Improved geometric path enumeration for verifying ReLU neural networks. In: Lahiri, S.K., Wang, C. (eds.) CAV 2020. LNCS, vol. 12224, pp. 66–96. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53288-8_4
2. Bastani, O., Ioannou, Y., Lampropoulos, L., Vytiniotis, D., Nori, A.V., Criminisi, A.: Measuring neural net robustness with constraints. In: Neural Information Processing Systems, NIPS 2016, pp. 2621–2629. Curran Associates Inc., USA (2016)
3. Bojarski, M., et al.: End to end learning for self-driving cars. In: NIPS 2016 Deep Learning Symposium (2016)
4. Boopathy, A., Weng, T.W., Chen, P.Y., Liu, S., Daniel, L.: CNN-Cert: an efficient framework for certifying robustness of convolutional neural networks. AAAI, January 2019
5. Bunel, R., Turkaslan, I., Torr, P.H.S., Kohli, P., Kumar, M.P.: Piecewise linear neural network verification: a comparative study. CoRR abs/1711.00455v1 (2017). http://arxiv.org/abs/1711.00455v1
6. Bunel, R.R., Turkaslan, I., Torr, P.H.S., Kohli, P., Mudigonda, P.K.: A unified view of piecewise linear neural network verification. In: NeurIPS, pp. 4795–4804 (2018)
7. Codevilla, F., Miiller, M., López, A., Koltun, V., Dosovitskiy, A.: End-to-end driving via conditional imitation learning. In: 2018 IEEE International Conference on Robotics and Automation (ICRA), pp. 1–9, May 2018. https://doi.org/10.1109/ICRA.2018.8460487
8. Dutta, S., Jha, S., Sankaranarayanan, S., Tiwari, A.: Output range analysis for deep feedforward neural networks. In: Dutle, A., Muñoz, C., Narkawicz, A. (eds.) NFM 2018. LNCS, vol. 10811, pp. 121–138. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-77935-5_9
9. Dvijotham, K., Stanforth, R., Gowal, S., Mann, T., Kohli, P.: A dual approach to scalable verification of deep networks. In: Conference on Uncertainty in Artificial Intelligence (UAI 2018), pp. 162–171. AUAI Press, Corvallis (2018)
10. Ehlers, R.: Formal verification of piece-wise linear feed-forward neural networks. In: D'Souza, D., Narayan Kumar, K. (eds.) ATVA 2017. LNCS, vol. 10482, pp. 269–286. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68167-2_19

11. Gehr, T., Mirman, M., Drachsler-Cohen, D., Tsankov, P., Chaudhuri, S., Vechev, M.: AI2: safety and robustness certification of neural networks with abstract interpretation. In: 2018 IEEE Symposium on Security and Privacy (SP), pp. 3–18, May 2018. https://doi.org/10.1109/SP.2018.00058
12. Goodfellow, I., Bengio, Y., Courville, A.: Deep Learning. MIT Press (2016). http://www.deeplearningbook.org
13. Guidotti, D., Barrett, C., Katz, G., Pulina, L., Narodytska, N., Tacchella, A.: The Verification of Neural Networks Library (VNN-LIB) (2019). www.vnnlib.org
14. Henriksen, P., Lomuscio, A.: Efficient neural network verification via adaptive refinement and adversarial search. In: Giacomo, G.D., et al. (eds.) ECAI 2020. Frontiers in Artificial Intelligence and Applications, vol. 325, pp. 2513–2520. IOS Press (2020). https://doi.org/10.3233/FAIA200385
15. Huang, X., Kwiatkowska, M., Wang, S., Wu, M.: Safety verification of deep neural networks. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 3–29. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_1
16. Katz, G., Barrett, C., Dill, D.L., Julian, K., Kochenderfer, M.J.: Reluplex: an efficient SMT solver for verifying deep neural networks. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 97–117. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_5
17. Katz, G., et al.: The Marabou framework for verification and analysis of deep neural networks. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 443–452. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_26
18. Liu, C., Arnon, T., Lazarus, C., Barrett, C., Kochenderfer, M.J.: Algorithms for verifying deep neural networks. CoRR abs/1903.06758 (2019)
19. ONNX: Open Neural Network Exchange (2017). https://github.com/onnx/onnx
20. Pham, L.H., Li, J., Sun, J.: SOCRATES: towards a unified platform for neural network verification. CoRR abs/2007.11206 (2020). https://arxiv.org/abs/2007.11206
21. Raghunathan, A., Steinhardt, J., Liang, P.: Certified defenses against adversarial examples. In: ICLR. OpenReview.net (2018)
22. Ruan, W., Huang, X., Kwiatkowska, M.: Reachability analysis of deep neural networks with provable guarantees. In: IJCAI, pp. 2651–2659. ijcai.org (2018)
23. Shriver, D., Elbaum, S., Dwyer, M.B.: DNNV: A framework for deep neural network verification (2021). http://arxiv.org/abs/2105.12841
24. Singh, G., Ganvir, R., Püschel, M., Vechev, M.T.: Beyond the single neuron convex barrier for neural network certification. In: Wallach, H.M., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E.B., Garnett, R. (eds.) Advances in Neural Information Processing Systems 32: NeurIPS 2019, pp. 15072–15083 (2019)
25. Singh, G., Gehr, T., Mirman, M., Püschel, M., Vechev, M.: Fast and effective robustness certification. In: Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., Garnett, R. (eds.) Advances in Neural Information Processing Systems 31, pp. 10802–10813. Curran Associates, Inc. (2018). http://papers.nips.cc/paper/8278-fast-and-effective-robustness-certification.pdf
26. Singh, G., Gehr, T., Püschel, M., Vechev, M.T.: An abstract domain for certifying neural networks. PACMPL **3**(POPL), 41:1–41:30 (2019)
27. Singh, G., Gehr, T., Püschel, M., Vechev, M.T.: Boosting robustness certification of neural networks. In: 7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, 6–9 May 2019. OpenReview.net (2019). https://openreview.net/forum?id=HJgeEh09KQ

28. Szegedy, C., et al.: Intriguing properties of neural networks. In: Bengio, Y., LeCun, Y. (eds.) 2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, 14–16 April 2014, Conference Track Proceedings (2014)
29. Tjeng, V., Xiao, K.Y., Tedrake, R.: Evaluating robustness of neural networks with mixed integer programming. In: International Conference on Learning Representations (2019). https://openreview.net/forum?id=HyGIdiRqtm
30. Wang, S., Pei, K., Whitehouse, J., Yang, J., Jana, S.: Efficient formal safety analysis of neural networks. In: NeurIPS, pp. 6369–6379 (2018)
31. Wang, S., Pei, K., Whitehouse, J., Yang, J., Jana, S.: Formal security analysis of neural networks using symbolic intervals. In: USENIX Security Symposium, pp. 1599–1614. USENIX Association (2018)
32. Weng, T., et al.: Towards fast computation of certified robustness for RELU networks. In: ICML, Proceedings of Machine Learning Research, vol. 80, pp. 5273–5282. PMLR (2018)
33. Wong, E., Kolter, J.Z.: Provable defenses against adversarial examples via the convex outer adversarial polytope. In: ICML, Proceedings of Machine Learning Research, vol. 80, pp. 5283–5292. PMLR (2018)
34. Xiang, W., Tran, H., Johnson, T.T.: Output reachable set estimation and verification for multilayer neural networks. IEEE Trans. Neural Netw. Learn. Syst. **29**(11), 5777–5783 (2018). https://doi.org/10.1109/TNNLS.2018.2808470
35. Yuan, X., He, P., Zhu, Q., Li, X.: Adversarial examples: attacks and defenses for deep learning. IEEE Trans. Neural Netw. Learn. Syst. **30**(9), 2805–2824 (2019)
36. Zhang, H., Weng, T., Chen, P., Hsieh, C., Daniel, L.: Efficient neural network robustness certification with general activation functions. Adv. Neural Inf. Process. Syst. **31**, 4944–4953 (2018)