DNNV: A Framework for Deep Neural Network Verification

David Shriver[0000−0003−0208−6517], Sebastian Elbaum[0000−0001−9592−1352], and Matthew B. Dwyer[0000−0002−1937−1544]

University of Virginia, Charlottesville, VA, USA
{dls2fc,matthewbdwyer,selbaum}@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 [4, 8]. 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 [3, 5, 7, 9–12, 15, 23–28, 30–35, 37], 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 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

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 [17]	*Reluplex*-NNET	hard-coded	Search
Planet [11]	RLV	RLV	Search
BaB [7]	RLV	RLV	Search
BaBSB [7]	RLV	RLV	Search
MIPVerify [30]	MIPVerify Julia API	MIPVerify Julia API	Optimization
Neurify [31]	*Neurify*-NNET	hard-coded	Search-Optimization
DeepZono [26]	ONNX*, *ERAN*-PYT, *ERAN*-TF	*ERAN* Python API	Reachability
DeepPoly [27]	ONNX*, *ERAN*-PYT, *ERAN*-TF	*ERAN* Python API	Reachability
RefineZono [28]	ONNX*, *ERAN*-PYT, *ERAN*-TF	*ERAN* Python API	Reachability
RefinePoly [25]	ONNX*, *ERAN*-PYT, *ERAN*-TF	*ERAN* Python API	Reachability
Marabou [18]	*Reluplex*-NNET or ONNX*	*Marabou* Python API	Search
nnenum [2]	ONNX*	*nnenum* Python API	Search-Reachability
VeriNet [14]	ONNX* or *Neurify*-NNET	*VeriNet* Python API	Search-Optimization

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.

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 Section 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.

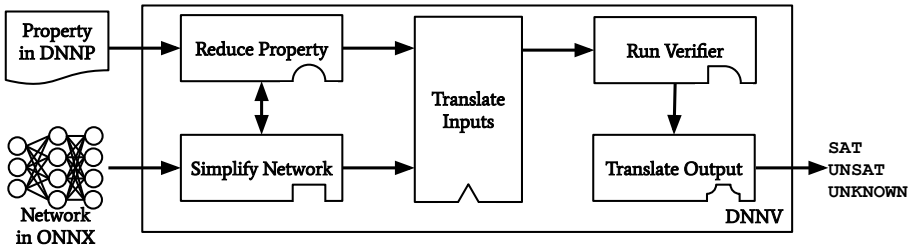

Fig. 1. DNNV Architecture

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 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., [13].

Given a DNN, $\mathcal{N} : \mathbb{R}^n \to \mathbb{R}^m$, 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: $\forall x \in \mathbb{R}^n : \phi_{\mathcal{X}}(x) \to \phi_{\mathcal{Y}}(\mathcal{N}(x))$.

A widely studied class of properties is *robustness*, which originated with the study of adversarial examples [29, 36]. These properties specify that inputs from

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

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 [11,26,27,30,31]. 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 [17,18,31].

A recent survey on DNN verification [21] 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 [12, 24–28, 35]. 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 [3,10,23,30,34]. Search-based methods explore regions of the input space where they then formulate reachability or optimization sub-problems. Search-based methods include [7,11,15,17,32,33].

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.

ONNX For specifying general deep neural network architectures, we choose the open source DNN format ONNX [22]. 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

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

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

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

```
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 1 (specified on Line 11) from image 1 of the dataset (selected on Lines 9-10), 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).

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

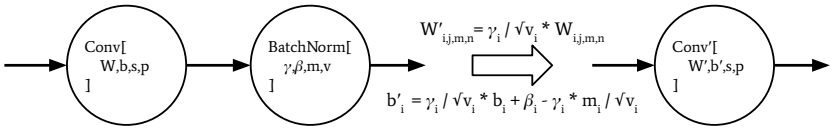

Fig. 3. Batch Normalization Simplification simplifies a batch norm following a convolution operation to an equivalent single convolution operation with modified weights and bias, while maintaining the strides and pads.

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, $\mathit{simplify}(G_{\mathcal{N}}) = G_{\mathcal{N}'}$, 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.

DNNV currently includes 6 additional DNN simplifications, enumerated and described in more detail in Appendix B.

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, $\mathit{reduce} : \Psi \to P(\Psi)$, aims to transform a verification problem, $\langle \mathcal{N}, \phi \rangle = \psi \in \Psi$, to an equivalid form, $\mathit{reduce}(\psi) = \{\langle \mathcal{N}_1, \phi_1 \rangle, \ldots, \langle \mathcal{N}_k, \phi_k \rangle\}$, 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

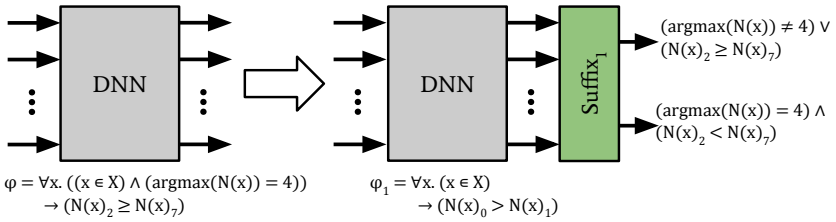

Fig. 4. Property reduction to a local robustness property adds a suffix that classifies outputs as violations or non-violations of the original output constraints, and changing the property to a common form of robustness property.

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.

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. 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.

Table 3. Verifier benchmarks.

Key	Name	Uses	#P	#$\mathcal{N}$	Features ¬HR	C	R	¬ReLU
AX	ACAS Xu	[2, 7, 17, 18, 31]	10	45				
CD	Collision Detection	[7, 11, 18]	500	1				
PM	*Planet* MNIST	[11]	7	1	✓	✓		
TS	TwinStream	[6]	1	81				
PCA	PCAMNIST	[7]	12	17				
MM	*MIPVerify* MNIST	[30]	10000	5		✓		
MC	*MIPVerify* CIFAR10	[30]	10000	2		✓	✓	
NM	*Neurify* MNIST	[14, 31]	500	4		✓		
NDB	*Neurify* Drebin	[31]	500	3				
NDV	*Neurify* DAVE	[31]	200	1	✓	✓		
DZM	*DeepZono* MNIST	[26]	1700	10		✓	✓	✓
DZC	*DeepZono* CIFAR10	[26]	1700	5		✓		✓
DPM	*DeepPoly* MNIST	[14, 27]	1500	8		✓		✓
DPC	*DeepPoly* CIFAR10	[27]	800	5		✓		
RZM	*RefineZono* MNIST	[28]	800	8		✓		
RZC	*RefineZono* CIFAR10	[28]	200	2		✓		
RPM	*RefinePoly* MNIST	[25]	600	6		✓		
RPC	*RefinePoly* CIFAR10	[25]	300	3		✓	✓	
VC	*VeriNet* CIFAR10	[14]	250	1		✓		

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. 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 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.

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	NDB	NDV	DZM	DZC	DPM	DPC	RZM	RZC	RPM	RPC	VC
Reluplex	●	◐	○	●	●	○	○	○	◑	○				○	○	○	○	○	○
Planet	●	◐	●	●	●	◑	◑	◑	◑	◑				◑	◑	◑	◑	◑	◑
BaB	●	◐	●	●	●	◑	○	◑	◑	◑				◑	◑	◑	◑	○	◑
BaBSB	●	◐	●	●	●	◑	○	◑	◑	◑				◑	◑	◑	◑	○	◑
MIPVerify	◑	○	○	◑	◑	●	◐	○	◑	○				○	○	○	○	○	○
Neurify	●	○	◑	◑	◑	◑	○	●	●	●				◑	◑	◑	◑	○	◑
DeepZono	●	○	○	◑	◑	◑	◑	◑	◑	○	●	●	●	●	●	●	●	●	◑
DeepPoly	●	○	○	◑	◑	◑	◑	◑	◑	○	●	●	●	●	●	●	●	●	◑
RefineZono	●	○	○	◑	◑	◑	◑	◑	◑	○	●	●	●	●	●	●	●	●	◑
RefinePoly	●	○	○	◑	◑	◑	◑	◑	◑	○	●	●	●	●	●	●	●	●	◑
Marabou	●	◐	◑	●	●	◑	○	◑	◑	◑				◑	◑	◑	◑	○	◑
nnenum	●	○	◑	◑	◑	◑	○	◑	◑	◑				◑	◑	◑	◑	○	◑
VeriNet	◑	○	○	◑	◑	◑	○	●	◑	○	○	◑	◑	◑	◑	◑	◑	○	●

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.

References

1. Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., Rieck, K.: DREBIN: effective and explainable detection of android malware in your pocket. In: 21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014. The Internet Society (2014), https://www.ndss-symposium.org/ndss2014/drebin-effective-and-explainable-detection-android-malware-your-pocket
2. 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.) Computer Aided Verification. pp. 66–96. Springer International Publishing, Cham (2020)
3. Bastani, O., Ioannou, Y., Lampropoulos, L., Vytiniotis, D., Nori, A.V., Criminisi, A.: Measuring neural net robustness with constraints. In: Proceedings of the 30th International Conference on Neural Information Processing Systems. pp. 2621–2629. NIPS'16, Curran Associates Inc., USA (2016), http://dl.acm.org/citation.cfm?id=3157382.3157391
4. Bojarski, M., Testa, D.D., Dworakowski, D., Firner, B., Flepp, B., Goyal, P., Jackel, L.D., Monfort, M., Muller, U., Zhang, J., Zhang, X., Zhao, J., Zieba, K.: End to end learning for self-driving cars. In: NIPS 2016 Deep Learning Symposium (2016)
5. Boopathy, A., Weng, T.W., Chen, P.Y., Liu, S., Daniel, L.: Cnn-cert: An efficient framework for certifying robustness of convolutional neural networks. In: AAAI (Jan 2019)
6. 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
7. 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)
8. 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
9. Dutta, S., Jha, S., Sankaranarayanan, S., Tiwari, A.: Output range analysis for deep feedforward neural networks. In: NFM. Lecture Notes in Computer Science, vol. 10811, pp. 121–138. Springer (2018)
10. Dvijotham, K., Stanforth, R., Gowal, S., Mann, T., Kohli, P.: A dual approach to scalable verification of deep networks. In: Proceedings of the Thirty-Fourth Conference Annual Conference on Uncertainty in Artificial Intelligence (UAI-18). pp. 162–171. AUAI Press, Corvallis, Oregon (2018)
11. Ehlers, R.: 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. pp. 269–286 (2017). https://doi.org/10.1007/978-3-319-68167-2_19, https://doi.org/10.1007/978-3-319-68167-2_19
12. 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
13. Goodfellow, I., Bengio, Y., Courville, A.: Deep Learning. MIT Press (2016), http://www.deeplearningbook.org

14. Henriksen, P., Lomuscio, A.: Efficient neural network verification via adaptive refinement and adversarial search. In: Giacomo, G.D., Catalá, A., Dilkina, B., Milano, M., Barro, S., Bugarín, A., Lang, J. (eds.) ECAI 2020 - 24th European Conference on Artificial Intelligence, 29 August-8 September 2020, Santiago de Compostela, Spain, August 29 - September 8, 2020 - Including 10th Conference on Prestigious Applications of Artificial Intelligence (PAIS 2020). Frontiers in Artificial Intelligence and Applications, vol. 325, pp. 2513–2520. IOS Press (2020). https://doi.org/10.3233/FAIA200385, https://doi.org/10.3233/FAIA200385
15. Huang, X., Kwiatkowska, M., Wang, S., Wu, M.: Safety verification of deep neural networks. In: CAV (1). Lecture Notes in Computer Science, vol. 10426, pp. 3–29. Springer (2017)
16. Hudak, P.: Modular domain specific languages and tools. In: Proceedings. Fifth International Conference on Software Reuse. pp. 134–142. IEEE (1998)
17. Katz, G., Barrett, C.W., Dill, D.L., Julian, K., Kochenderfer, M.J.: Reluplex: An efficient SMT solver for verifying deep neural networks. In: Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I. pp. 97–117 (2017). https://doi.org/10.1007/978-3-319-63387-9_5, https://doi.org/10.1007/978-3-319-63387-9_5
18. Katz, G., Huang, D.A., Ibeling, D., Julian, K., Lazarus, C., Lim, R., Shah, P., Thakoor, S., Wu, H., Zeljić, A., et al.: The Marabou framework for verification and analysis of deep neural networks. In: International Conference on Computer Aided Verification. pp. 443–452. Springer (2019)
19. Krizhevsky, A., Hinton, G.: Learning multiple layers of features from tiny images (2009)
20. LeCun, Y., Cortes, C., Burges, C.J.: The mnist database of handwritten digits
21. Liu, C., Arnon, T., Lazarus, C., Barrett, C., Kochenderfer, M.J.: Algorithms for verifying deep neural networks. CoRR **abs/1903.06758** (2019)
22. ONNX: Open Neural Network Exchange. https://github.com/onnx/onnx (2017)
23. Raghunathan, A., Steinhardt, J., Liang, P.: Certified defenses against adversarial examples. In: ICLR. OpenReview.net (2018)
24. Ruan, W., Huang, X., Kwiatkowska, M.: Reachability analysis of deep neural networks with provable guarantees. In: IJCAI. pp. 2651–2659. ijcai.org (2018)
25. 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: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada. pp. 15072–15083 (2019), http://papers.nips.cc/paper/9646-beyond-the-single-neuron-convex-barrier-for-neural-network-certification
26. 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
27. Singh, G., Gehr, T., Püschel, M., Vechev, M.T.: An abstract domain for certifying neural networks. PACMPL **3**(POPL), 41:1–41:30 (2019)
28. 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, May 6-9, 2019. OpenReview.net (2019), https://openreview.net/forum?id=HJgeEh09KQ

29. Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I.J., Fergus, R.: Intriguing properties of neural networks. In: Bengio, Y., LeCun, Y. (eds.) 2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings (2014)
30. 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
31. Wang, S., Pei, K., Whitehouse, J., Yang, J., Jana, S.: Efficient formal safety analysis of neural networks. In: NeurIPS. pp. 6369–6379 (2018)
32. 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)
33. Weng, T., Zhang, H., Chen, H., Song, Z., Hsieh, C., Daniel, L., Boning, D.S., Dhillon, I.S.: Towards fast computation of certified robustness for relu networks. In: ICML. Proceedings of Machine Learning Research, vol. 80, pp. 5273–5282. PMLR (2018)
34. 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)
35. Xiang, W., Tran, H., Johnson, T.T.: Output reachable set estimation and verification for multilayer neural networks. IEEE Transactions on Neural Networks and Learning Systems **29**(11), 5777–5783 (Nov 2018). https://doi.org/10.1109/TNNLS.2018.2808470
36. Yuan, X., He, P., Zhu, Q., Li, X.: Adversarial examples: Attacks and defenses for deep learning. IEEE Transactions on Neural Networks and Learning Systems **30**(9), 2805–2824 (2019)
37. Zhang, H., Weng, T., Chen, P., Hsieh, C., Daniel, L.: 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. pp. 4944–4953 (2018)

A DNNP

A property specification defines the desired behavior of a DNN in a formal language. DNNV uses a custom Python-embedded DSL for writing property specifications, which we call DNNP. Embedding DNNP in Python allows for the rich ecosystem of the host language to be used in writing specifications [16]. However, DNNV is still of a work-in-progress, so some expressions (such as star expressions) are not yet supported by our property parser. We are still working to fully support all Python expressions, but the current version supports the most common use cases.

⟨*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*⟩ ‘`)`’
| ‘`Parameter(`’ ⟨*id*⟩ ‘`, type=`’ ⟨*id*⟩ ‘`)`’
| ...
| ⟨*python-expr*⟩

⟨*expr-seq*⟩ ::= ⟨*expr*⟩ | ⟨*expr-seq*⟩ ‘`,`’ ⟨*expr*⟩

Fig. 5. Subset of the grammar for DNNP.

Fig. 5 shows the definition of the DNNP grammar. The general structure of a property specification is as follows:

1. A set of python module imports
2. A set of variable definitions
3. A property expression

A.1 Imports

Imports have the same syntax as Python import statements, and they can be used to import arbitrary Python modules and packages. This allows re-use of datasets or input pre-processing code. For example, the Python package `numpy` can be imported 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.

A.2 Definitions

After any imports, DNNP allows a sequence of assignments to define variables that can be used in the final property specification. For example, `i = 0`, will define the variable `i` to a value of 0.

These definitions can be used to load data and configuration parameters, or to alias expressions that may be used in the property formula. For example, if the `torchvision.datasets` package has been imported, then `data = datasets.MNIST("/tmp")` will define a variable `data` referencing the MNIST dataset from this package. Additionally, the `Parameter` class can be used to declare parameters that can be specified at run time. `eps = Parameter("epsilon", type=float)`, will define the variable `eps` to have type float and will expect a value to be specified at run time. This value can be specified to DNNV with the option `--prop.epsilon`.

Definitions can also assign expressions to variables to be used in the property specification later. For example, `x_in_unit_hyper_cube = 0 <= x <= 1` can be used to assign an expression specifying that the variable `x` is within the unit hyper cube to a variable. This could be useful for more complex properties with a lot of redundant sub-expressions.

A network can be defined using the `Network` class. `N = Network("N")`, specifies a network with the name `N` (which is used at run time to concretize the network with a specific DNN model). All networks with the same name refer to the same model.

A.3 Property Expression

Finally, the last part of the property specification is the property formula itself. It must appear at the end of the property specification. All statements before the property formula must be either import or assignment statements.

The property formula defines the desired behavior of the DNN in a subset of first-order-logic. It can make use of arbitrary Python code, as well as any of the expressions defined before it.

DNNP provides many functions to define expressions. The function `Forall(symbol, expression)` can be used to specify that the provided expression is valid for all values of the specified symbol. The function `And(*expression)`, specifies that all of the expressions passed as arguments to the function must be valid. `And(expr1, expr2)` can be equivalently specified as `expr1 & expr2`. The function `Or(*expression)`, specifies that at least one of the expressions passed as arguments to the function must be valid. `Or(expr1, expr2)` can be equivalently specified as `expr1 | expr2`. The function `Implies(expression1, expression2)`, specifies that if `expression1` is true, then `expression2` must also be true. The `argmin` and `argmax` functions can be used to get the argmin or argmax value of a network's output, respectively.

In property expressions, networks can be called like functions to get the outputs for the network for a given input. Networks can be applied to symbolic variables (such as universally quantified variables), as well as numpy arrays.

B DNN Simplifications

In this section, we describe the DNN simplifications currently performed by DNNV. This is not a full list of all possible simplifications, but have been useful for some networks we have encountered in practice.

B.1 BatchNormalization Simplification

BatchNormalization simplification removes BatchNormalization operations from a network by combining them with a preceeding Conv operation or Gemm operation. If no applicable preceeding layer exists, the batch normalization layer is converted into an equivalent Conv operation. This simplification can decrease the number of operations in the model and increase verifier support, since many verifiers do not support BatchNormalization operations.

B.2 Identity Removal

DNNV removes many types of identity operations from DNN models, including explicit Identity operations, Concat operations with a single input, and Flatten operations applied to flat tensors. Such operations can occur in DNN models due to user error, or through automated processes, and their removal does not affect model behavior.

B.3 Convert MatMul followed by Add to Gemm

DNNV converts instances of MatMul (matrix multiplication) operations, followed immediately by Add operations to an equivalent Gemm (generalized matrix multiplication) operation. The Gemm operation generalizes the matrix multiplication and addition, and can simplify subsequent processing and analysis of the DNN.

B.4 Combine Consecutive Gemm

DNNV combines two consecutive Gemm operations into a single equivalent Gemm operation, reducing the number of operations in the DNN.

B.5 Combine Consecutive Conv

In special cases, DNNV can combine consecutive Conv (convolution) operations into a single equivalent Conv operation, reducing the number of operations in the DNN. Currently, DNNV can combine Conv operations when the first Conv uses a diagonal 1 by 1 kernel with a stride of 1 and no zero padding, and the second Conv has no zero padding. This case can occur after converting a normalization layer (such as BatchNormalization) to a Conv operation.

B.6 Bundle Pad

DNNV can bundle explicit Pad operations with an immediately succeeding Conv or MaxPool operation. This both simplifies the DNN model, and increases support, since many verifiers do not support explicit Pad operations (but can support padding as part of a Conv or MaxPool operation).

B.7 Move Activations Backward

DNNV moves activation functions through reshaping operations to immediately succeed the most recent non-reshaping operation. This is possible since activation functions are element-wise operations. This transformation can simplify pattern matching in later analysis steps by reducing the number of possible patterns.

C Property Reduction

In this section, we provide the algorithm for reducing properties to reachability properties, as well as proofs for the equivalidity of the resulting set of reachability properties and original property. Algorithm 1 is the overall reduction algorithm, while Algorithm 2 and 3 are subprocedures used by the main algorithm. The algorithm and proofs for reduction to other property types (such as robustness) are very similar.

We assume that properties are of the form $\forall x \in \mathbb{R}^n : \phi_{\mathcal{X}}(x) \rightarrow \phi_{\mathcal{Y}}(\mathcal{N}(x))$, where $\phi_{\mathcal{X}}$ is a set of constraints over the inputs – the pre-condition, and $\phi_{\mathcal{Y}}$ is a set of constraints over the outputs – the post-condition. We also assume that constraints are represented as linear inequalities.

Algorithm 1: Property Reduction

Input: Correctness problem $\langle \mathcal{N}, \phi \rangle$
Output: A set of robustenss problems $\{\langle \mathcal{N}_1, \phi_1 \rangle, ..., \langle \mathcal{N}_i, \phi_i \rangle\}$

```
begin
    φ' ← DNF(¬φ)
    Ψ ← ∅
    for disjunct ∈ φ' do
        φ_X ← extract_input_constraints(disjunct)
        φ_Y ← extract_output_constraints(disjunct)
        hspoly ← disjunct_to_hpolytope(φ_Y)
        suffix ← construct_suffix(hspoly)
        N' ← suffix ∘ N
        φ' ← ∀x.(x ∈ φ_X ⟹ N'(x)_0 > N'(x)_1)
        Ψ ← Ψ ∪ ⟨N', φ'⟩
    return Ψ
```

Algorithm 2: disjunct_to_hpolytope

Input: Conjunction of linear inequalities ϕ_i
Output: Halfspace polytope H

```
begin
    H ← (A, b) where A is an (|φ_i|) × (m) matrix where columns correspond
      to the output variables N(x)_0 to N(x)_{m−1}
    for ineq_j ∈ φ_i do
        if ineq_j uses ≥ then
            swap lhs and rhs; switch inequality to ≤
        else if ineq_j uses > then
            swap lhs and rhs; switch inequality to <
        move variables to lhs; move constants to rhs
        if ineq_j uses < then
            decrement rhs; switch inequality to ≤
        A_j ← coefficients of variables on lhs
        b_j ← rhs constant
    return H
```

Algorithm 3: construct_suffix

Input: Halfspace polytope $H = (A, b)$
Output: A DNN with 2 fully connected layers S

```
begin
    S_h ← ReLU(FullyConnectedLayer(A, −b))
    W ← [1 1 ... 1]
        [0 0 ... 0]
    S_o ← FullyConnectedLayer(W, 0⃗)
    S ← S_o ∘ S_h
    return S
```

C.1 Proofs

In order to prove that the property reduction produces a set of correctness problems equivalid to the original problem, we first prove the following lemmas:

Lemma 1. *Let ϕ be a conjunction of linear inequalities over the variables x_i for i from 0 to $n-1$. We can construct a halfspace polytope $H = (A, b)$ with Algorithm 2 such that $(Ax \leq b) \Leftrightarrow (x \models \phi)$.*

Proof. We first show that every linear inequality in the conjunction can be reformulated to the form $a_0x_0 + a_1x_1 + ... + a_{n-1}x_{n-1} \leq b$. It is trivial to show that inequalities with a $\geq$ comparison can be manipulated to an equivalent form with $\leq$, and $>$ can be manipulated to become $<$. It is also trivial to show that the inequality can be manipulated to have variables on lhs and a constant value on rhs. This results in a conjunction of linear inequalities of the form $a_0x_0 + a_1x_1 + ... + a_{n-1}x_{n-1} < b$ and $a_0x_0 + a_1x_1 + ... + a_{n-1}x_{n-1} \leq b$. Finally,

the $<$ comparison can be changed to a $\leq$ comparison by decrementing the constant on the right-hand-side from b to b' where b' is the largest representable number less than b.

We prove that linear inequalities using the $<$ comparison can be reformulated to use a $\leq$ comparison using a proof by contradiction. Assume that either $a_0x_0 + a_1x_1 + ... + a_{n-1}x_{n-1} < b$ and $a_0x_0 + a_1x_1 + ... + a_{n-1}x_{n-1} > b'$ or $a_0x_0 + a_1x_1 + ... + a_{n-1}x_{n-1} \geq b$ and $a_0x_0 + a_1x_1 + ... + a_{n-1}x_{n-1} \leq b'$. Then one of two cases must be true. Either $b' < a_0x_0 + a_1x_1 + ... + a_{n-1}x_{n-1} < b$, a contradiction, since $a_0x_0 + a_1x_1 + ... + a_{n-1}x_{n-1}$ cannot be both larger than the largest representable number less than b and also less than b.[2] Or $b \leq a_0x_0 + a_1x_1 + ... + a_{n-1}x_{n-1} \leq b'$, a contradiction, since $b' < b$ by definition.

Given a conjunction of linear inequalities in the form $a_0x_0 + a_1x_1 + ... + a_{n-1}x_{n-1} \leq b$, Algorithm 2 constructs A and b with a row in A and value in b corresponding to each conjunct. There are two cases to prove: $(Ax \leq b) \rightarrow (x \models \phi)$ and $(x \models \phi) \rightarrow (Ax \leq b)$.

We prove case 1 by contradiction. Assume $(Ax \leq b)$ and $(x \not\models \phi)$. By the construction of H in Algorithm 2, each conjunct of ϕ is exactly 1 constraint in H. If $Ax \leq b$, then all constraints in H must be satisifed, and thus all conjuncts in ϕ must be satisfied and $x \models \phi$, a contradiction.

We prove case 2 by contradiction. Assume $(x \models \phi)$ and $(Ax \not\leq b)$. By the construction of H in Algorithm 2, each conjunct of ϕ is exactly 1 constraint in H. If $x \models \phi$, then all conjuncts in ϕ must be satisfied, and thus all constraints in H must be satisifed and $Ax \leq b$, a contradiction.

Lemma 2. *Let $H = (A, b)$ be a halfspace polytope such that $Ax \leq b$. Then, a DNN, $\mathcal{N}_s$, can be built with Algorithm 3 that classifies whether its outputs satisfy $A(\mathcal{N}(x)) \leq b$ or not. Formally, $\mathcal{N}(x) \in H \Leftrightarrow \mathcal{N}_s(x)_0 \leq \mathcal{N}_s(x)_1$.*

Proof. There are 2 cases:

1. $\mathcal{N}(x) \in H \rightarrow \mathcal{N}_s(x)_0 \leq \mathcal{N}_s(x)_1$
2. $\mathcal{N}_s(x)_0 \leq \mathcal{N}_s(x)_1 \rightarrow \mathcal{N}(x) \in H$

We prove case 1 by contradiction. Assume $\mathcal{N}(x) \in H$ and $\mathcal{N}_s(x)_0 > \mathcal{N}_s(x)_1$. From Algorithm 3, each neuron in the hidden layer of $\mathcal{N}_s$ corresponds to one constraint in H. The weights of each neuron are the values in the corresponding row of A, and the bias is the negation of the corresponding value of b. If the output $\mathcal{N}(x)$ satisfies the constraint, then the value of the neuron will be less than or equal to 0, otherwise it will be greater than 0. After application of the ReLU activation function, all neurons will be equal to 0 if their corresponding constraint is satisfied by $\mathcal{N}(x)$ and greater than 0 otherwise. The first neuron in the final layer sums all of the neurons in the hidden layer, while the second neuron has a constant value of 0. If $\mathcal{N}(x) \in H$, then all neurons in the hidden layer after activation must have a value of 0 since all constraints are satisfied. However, if all neurons have a value of 0, then their sum must also have a value of zero, and therefore $\mathcal{N}_s(x)_0 = \mathcal{N}_s(x)_1$, a contradiction.

[2] We further discuss the assumption that such a number exists in Section C.2.

We prove case 2 by contradiction. Assume $\mathcal{N}_s(x)_0 \leq \mathcal{N}_s(x)_1$ and $\mathcal{N}(x) \notin H$. From Algorithm 3, each neuron in the hidden layer of $\mathcal{N}_s$ corresponds to one constraint in H. The weights of each neuron are the values in the corresponding row of A, and the bias is the negation of the corresponding value of b. If the output $\mathcal{N}(x)$ satisfies the constraint, then the value of the neuron will be less than or equal to 0, otherwise it will be greater than 0. After application of the ReLU activation function, all neurons will be equal to 0 if their corresponding constraint is satisfied by $\mathcal{N}(x)$ and greater than 0 otherwise. The first neuron in the final layer sums all of the neurons in the hidden layer, while the second neuron has a constant value of 0. If $\mathcal{N}(x) \notin H$, then at least one neurons in the hidden layer after activation must have a value greater than 0 since at least one constraint is not satisfied. However, if any neuron has a value greater than 0, then their sum must also have a value greater than zero, and therefore $\mathcal{N}_s(x)_0 > \mathcal{N}_s(x)_1$, a contradiction.

Theorem 1. *Let $\psi = \langle \mathcal{N}, \phi \rangle$ be an arbitrary correctness problem with a DNN correctness property defined as a formula of disjunctions and conjunctions of linear inequalities over the input and output variables of $\mathcal{N}$. Property Reduction (Algorithm 1) maps ψ to an equivalid set of correctness problems $reduce(\psi) = \{\langle \mathcal{N}_1, \phi_1 \rangle, \ldots, \langle \mathcal{N}_k, \phi_k \rangle\}$.*

$$\mathcal{N} \models \psi \Leftrightarrow \forall \langle \mathcal{N}_i, \phi_i \rangle \in reduce(\psi). \mathcal{N}_i \models \phi_i$$

Proof. A model that satisfies any disjunct of $DNF(\neg\phi)$ falsifies ϕ. If ϕ is falsifiable, then at least one disjunct of $DNF(\neg\phi)$ is satisfiable.

Algorithm 1 constructs a correctness problem for each disjunct. For each disjunct, Algorithm 1 constructs a halfspace polytope, H, which is used to construct a suffix network, $\mathcal{N}_s$. The algorithm then constructs the network $\mathcal{N}'(x) = \mathcal{N}_s(\mathcal{N}(x))$. Algorithm 1 pairs each constructed network with the property $\phi = \forall x. x \in [0,1]^n \rightarrow \mathcal{N}'(x)_0 > \mathcal{N}'(x)_1$. A violation occurs only when $\mathcal{N}'(x)_0 \leq \mathcal{N}'(x)_1$. By Lemmas 1 and 2, we get that $\mathcal{N}'(x)_0 \leq \mathcal{N}'(x)_1$ if and only if $\mathcal{N}'(x) \in H$. If $\mathcal{N}'(x) \in H$ then $\mathcal{N}'(x)$ satisfies the disjunct and is therefore a violation of the original property.

C.2 On the Existance of a Bounded Largest Representable Number

Our proof that property reduction generates a set of robustness problems equivalid to an arbitrary problem relies on the assumption that strict inequalities can be converted to non-strict inequalities. To do so we rely on the existance of a largest representable number that is less than some given value. While this is not necessarily true for all sets of numbers (e.g., $\mathbb{R}$), it is true for for most numeric representations used in computation (e.g., IEEE 754 floating point arithmetic).

D Verification Benchmarks

We examine the benchmarks used to evaluate each of the 13 verifiers supported by DNNV, and determine 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. Here we provide a short description of each of the 19 verification benchmarks that we have identified. A short summary of some of the features of each verifier relevant to DNNV are shown in Table 3. 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).

The ACAS Xu (AX) benchmark, introduced for *Reluplex* [17], is one of the most used verification benchmarks [2, 7, 18, 31]. The benchmark consists of 10 properties. Property ϕ_1 is a reachability property, specifying an upper bound on one of the 5 output variables. Properties ϕ_5, ϕ_6, ϕ_9, and ϕ_{10} are all traditional class robustness properties, specifying the desired class for the given input region. Properties ϕ_3, ϕ_4, ϕ_7 and ϕ_8 are reachability properties, specifying a set of acceptable classes for the input region. Properties ϕ_2 is also a reachability property, specifying that a given output value cannot be greater than all others. Each of the properties are applied to a subset of 45 networks trained on an aircraft collision avoidance dataset, with 5 inputs, 5 output classes and 6 layers of 50 neurons each. The original benchmark included networks in *Reluplex*-NNET format, and a custom version of *Reluplex* was written for each property. Later uses of the benchmark translated the verification problems into RLV format, which is used by *Planet*, *BaB*, and *BaBSB*, as well as translating the networks into ONNX. The benchmark in ONNX and DNNP format is fully supported by DNNV.

The Collision Detection (CD) benchmark [11], intoduced for the evaluation of *Planet*, consists of 500 local robustness properties for an 80 neuron network with a fully connected layer and max pooling layer that classifies whether 2 simulated vehicles will collide, given their current state. The verification problems, in RLV format, are supported by *Planet*, *BaB*, and *BaBSB*. The problems have also been modified to convert max pooling operations to a sequence of fully-connected layers with ReLU activations, and then translated to *Reluplex*-NNET format, enabling off the shelf support by *Marabou*, and a generalized version of *Reluplex*. This benchmark is one of the few that is not supported by DNNV, since the network contains structures that are not easily supported by ONNX. In particular, the max-pooling operation in the original network, applied to a flat tensor, cannot be encoded by ONNX from their original format.

The *Planet* MNIST (PM) benchmark [11] is a set of 7 properties over a convolutional network trained on the MNIST dataset [20]. The first 4 of these are reachability properties with hyper-rectangle input constraints, while the next 2 are local robustness properties with hyper-rectangle input constraints, and the final property is an local robustness property with halfspace-polytope input constraints. The original benchmark was provided in RLV format. The first 6 of these properties are currently supported by DNNV, while the final property could be supported by DNNV with additional engineering effort.

The TwinStream (TS) benchmark [6] consists of 1 property applied to 81 networks that output a constant value. The property asserts that for all inputs, the output of the network is positive. The original benchmark was provided in RLV format. This benchmark is fully supported by DNNV for all verifiers.

The PCAMNIST (PCA) benchmark [7] consists of 12 reachability properties applied to 17 networks trained on modified versions of the MNIST dataset to predict the parity of the digit represented by the first k components of the PCA decomposition of an image. The original benchmark was provided in RLV format. This benchmark is fully supported by DNNV for all verifiers.

MIPVerify MNIST (MM) consists of 10000 local robustness properties applied to 5 networks trained on the MNIST dataset. The networks have varied structures: 2 networks are fully connected and 3 are convolutional. We could not find an available version of the benchmark used by *MIPVerify* to evaluate its original input format. This benchmark is fully supported by DNNV for all verifiers except *Reluplex*, which does not support convolution operations.

MIPVerify CIFAR (MC) consists of 10000 local robustness properties applied to 2 networks trained on the CIFAR10 dataset [19]. One of these networks is a convolutional network and the other is a residual network. We could not find an available version of the benchmark used by *MIPVerify* to evaluate its original input format. This benchmark is supported by DNNV for verifiers that can support residual connections, including: *Planet*, *DeepZono*, *DeepPoly*, *RefineZono*, and *RefinePoly*. While the benchmark is supported by the version of *MIPVerify* used in its study, it is not supported through DNNV, since the publicly available version of *MIPVerify* does not support residual connections.

The *Neurify* MNIST (NM) benchmark [31] consists of 500 L_∞ local robustness properties across 4 MNIST networks, 3 fully connected networks with 58, 110, and 1034 neurons respectively, and a convolutional network with 4814 neurons. The original benchmark was provided in *Neurify*-NNET format, with properties hard-coded into the verifier. DNNV enables almost all verifiers to run on this benchmark. *Reluplex* cannot be run due to the presence of convolutional layers, which are not supported. *MIPVerify* cannot be run due to the presence of non-hypercube input constraints. While this limitation of the verifier can be satisfied with a property reduction for fully-connected networks, DNNV does not currenly support such a reduction for convolutional networks.

The *Neurify* Drebin (NDB) benchmark [31] consists of 500 L_∞ local robustness properties across 3 fully connected Drebin [1] networks with 102, 212, and 402 neurons each. The original benchmark was provided in *Neurify*-NNET format, with properties hard-coded into the verifier. This benchmark is fully supported by DNNV for all verifiers.

The *Neurify* DAVE (NDV) benchmark [31] consists of 200 local reachability properties, with 4 different types of input constraints (50 properties of each type). The first type of input constraint is an L_∞ constraint, which is equivalent to a hyper-rectangle constraint. The second type of input constraint is an L_1 constraint, which can be written as a halfspace polytope constraint. The third and fourth type of input constraint are image brightning and contrast, which can

be written as halfspace polytope constraints. The properties are applied to a convolutional network for an autonomous vehicle, with 10276 neurons. The original benchmark was provided in *Neurify*-NNET format, with properties hard-coded into the verifier. Similar to the *Neurify* MNIST benchmark, DNNV enables almost all verifiers to run on this benchmark. *Reluplex* cannot be run, due to the presence of convolutional layers, which are not supported, and *MIPVerify* cannot be run due to the presence of non-hypercube input constraints.

The *DeepZono* MNIST (DZM) benchmark [26] consists of 1700 local robustness properties, subsets of which are applied to 10 networks trained on the MNIST dataset. The networks have varied structures and activation functions: 3 networks are fully connected, 1 of which uses ReLU activations, 1 with Tanh activations, and 1 with Sigmoid activations; 6 are convolutional, 4 of which have ReLU activations, 1 with Tanh activations, and 1 with Sigmoid activations; and 1 is a residual network. The networks in the original benchmark were provided in a custom human-readable text format, with properties hard-coded into the verifier. DNNV does not increase the support for this benchmark due to the presence of both a residual network and non-ReLU activation functions.

The *DeepZono* CIFAR10 (DZC) benchmark [26] consists of 1700 local robustness properties, subsets of which are applied to 5 networks trained on the CIFAR10 dataset. The networks have varied structures and activation functions: 3 networks are fully connected, 1 of which uses ReLU activations, 1 with Tanh activations, and 1 with Sigmoid activations; and 2 are convolutional with ReLU activations. The networks in the original benchmark were provided in a custom human-readable text format, with properties hard-coded into the verifier. DNNV enables *VeriNet* to run on this benchmark. Other verifiers are not supported due to the non-ReLU activation functions.

The *DeepPoly* MNIST (DPM) benchmark [27] consists of 1500 local robustness properties, subsets of which are applied to 8 networks trained on the MNIST dataset. The networks have varied structures and activation functions: 5 networks are fully connected, 3 of which uses ReLU activations, 1 with Tanh activations, and 1 with Sigmoid activations; and 3 are convolutional with ReLU activations. The networks in the original benchmark were provided in a custom human-readable text format, with properties hard-coded into the verifier. DNNV enables *VeriNet* to run on this benchmark. Other verifiers are not supported due to the non-ReLU activation functions.

The *DeepPoly* CIFAR10 (DPC) benchmark [27] consists of 800 local robustness properties, subsets of which are applied to 5 networks trained on the CIFAR10 dataset. The networks have varied structures: 3 networks are fully connected with ReLU activations; and 2 are convolutional with ReLU activations. The networks in the original benchmark were provided in a custom human-readable text format, with properties hard-coded into the verifier. DNNV enables several additional verifiers to support this benchmark. In particular, it enables most verifiers that can be applied to convolutional networks with relu activations.

The *RefineZono* MNIST (RZM) benchmark [28] consists of 800 local robustness properties, subsets of which are applied to 8 networks trained on the MNIST dataset. 5 networks are fully connected with ReLU activations and 3 are convolutional with ReLU activations. The networks in the original benchmark were provided in a custom human-readable text format, with properties hard-coded into the verifier. DNNV enables several additional verifiers to support this benchmark. In particular, it enables most verifiers that can be applied to convolutional networks with relu activations.

The *RefineZono* CIFAR10 (RZC) benchmark [28] consists of 200 local robustness properties, subsets of which are applied to 2 networks trained on the CIFAR10 dataset. One of the networks is fully connected with ReLU activations and the other is convolutional with ReLU activations. The networks in the original benchmark were provided in a custom human-readable text format, with properties hard-coded into the verifier. DNNV enables several additional verifiers to support this benchmark. In particular, it enables most verifiers that can be applied to convolutional networks with relu activations.

The *RefinePoly* MNIST (RPM) benchmark [25] consists of 600 local robustness properties, subsets of which are applied to 6 networks trained on the MNIST dataset. 4 networks are fully connected with ReLU activations and 2 are convolutional with ReLU activations. The networks in the original benchmark were provided in a custom human-readable text format, with properties hard-coded into the verifier. DNNV enables several additional verifiers to support this benchmark. In particular, it enables most verifiers that can be applied to convolutional networks with relu activations.

The *RefinePoly* CIFAR10 (RPC) benchmark [25] consists of 300 local robustness properties, subsets of which are applied to 3 networks trained on the MNIST dataset. Two of the networks are convolutional with ReLU activations and the third is a residual network with ReLU activations. The networks in the original benchmark were provided in a custom human-readable text format, with properties hard-coded into the verifier. DNNV enables the *Planet* verifier to support this benchmark. In particular, it enables most verifiers that can be applied to convolutional networks with relu activations. Other verifiers do not support the residual structure of one of the networks.

The *VeriNet* CIFAR10 (VC) benchmark [14] consists of 250 local robustness properties applied to 1 convolutional network with ReLU activations. The networks were provided in ONNX format, with hard-coded properties. DNNV enables support of this benchmark by most of the integrated verifiers. *Reluplex* does not support convolutional networks, and *MIPVerify* does not support properties with input constraints that are not hyper-cubes.