DNNV

David Shriver

Sep 03, 2022

GETTING STARTED

DNNV is a framework for verifying deep neural networks (DNN). DNN verification takes in a neural network, and a property over that network, and checks whether the property is true or false. DNNV standardizes the network and property input formats to enable multiple verification tools to run on a single network and property. This facilitates both verifier comparison, and artifact re-use.

CHAPTER

ONE

INSTALLATION

We provide several installation options for DNNV. We recommend using pip for general usage.

1.1 With Pip

Requirements:

- Python 3.7, 3.8, or 3.9

To install the latest released version of DNNV using pip, run:

```
pip install dnnv
```

To install the latest and greatest version of DNNV using pip, run:

```
pip install git+https://github.com/dlshriver/dnnv.git@develop
```

1.2 With Docker

Requirements:

- Docker

To download and use the latest released version of DNNV in a docker image, run:

```
docker pull dlshriver/dnnv:latest
docker run --rm -it dlshriver/dnnv:latest
```

To use a specific version of DNNV, run:

```
docker pull dlshriver/dnnv:vX.X.X
docker run --rm -it dlshriver/dnnv:vX.X.X
```

Where `X.X.X` is replaced by the desired version of DNNV.

To use the latest development version of DNNV in a docker image, specify the `develop` tag:

```
docker pull dlshriver/dnnv:develop
docker run --rm -it dlshriver/dnnv:develop
```

The development version does not come with any verifiers pre-installed. These can be installed using the `dnnv_manage`, either in a custom Dockerfile or through an interactive prompt.

1.3 From Source

Requirements:

- Python 3.7 or 3.8
- Git

Installing DNNV from source is primarily recommended for development of DNNV itself. This requires Python 3.7 or above, as well as the venv module, which may need to be installed separately (e.g., `sudo apt-get install python3-venv`).

To clone the source code, run:

```
git clone https://github.com/dlshriver/dnnv.git
cd DNNV
```

To create a python virtual environment, and install required pacakges for this project, run:

```
python -m venv .venv
. .venv/bin/activate
pip install --upgrade pip flit
flit install -s
```

1.4 Verifier Installation

After installing DNNV, any of the supported verifiers can be installed using the `install` command to the `dnnv_manage` tool, followed by the name of the verifier. For example, to install the Reluplex verifier, run:

```
dnnv_manage install reluplex
```

DNNV supports the following verifiers:

- Reluplex
- planet
- BaB
- BaBSB
- MIPVerify
- Neurify
- ERAN (deepzono, deeppoly, refinezono, refinepoly)
- marabou
- nnenum
- VeriNet

CHAPTER

TWO

USAGE

DNNV can be used to run verification tools from the command line. If DNNV is not yet installed, see our *Installation guide* for more information.

2.1 DNNV Options

DNNV can be run from the command line. Specifying the `-h` option will list the available options:

```
$ dnnv -h
usage: dnnv [-h] [-V] [--seed SEED] [-v | -q] [-N NAME NETWORK]
          [--save-violation PATH] [--vnnlib] [--bab]
          [--bab.reluify_maxpools RELUIFY_MAXPOOLS]
          [--bab.smart_branching SMART_BRANCHING] [--eran]
          [--eran.domain {deepzono,deeppoly,refinezono,refinepoly}]
          [--eran.timeout_lp TIMEOUT_LP] [--eran.timeout_milp TIMEOUT_MILP]
          [--eran.use_area_heuristic USE_AREA_HEURISTIC] [--marabou]
          [--mipverify] [--neurify] [--neurify.max_depth MAX_DEPTH]
          [--neurify.max_thread MAX_THREAD] [--nnenum]
          [--nnenum.num_processes NUM_PROCESSES] [--planet] [--reluplex]
          [--verinet] [--verinet.max_proc MAX_PROC]
          [--verinet.no_split NO_SPLIT]
          property

dnnv - deep neural network verification

positional arguments:
  property

optional arguments:
  -h, --help            show this help message and exit
  -V, --version         show program's version number and exit
  --seed SEED           the random seed to use
  -v, --verbose         show messages with finer-grained information
  -q, --quiet           suppress non-essential messages
  -N, --network NAME NETWORK
  --save-violation PATH
                        the path to save a found violation
  --vnnlib              use the vnnlib property format
  --convert
```

(continues on next page)

(continued from previous page)

```
verifiers:
  --bab
  --eran
  --marabou
  --mipverify
  --neurify
  --nnenum
  --planet
  --reluplex
  --verinet

bab parameters:
  --bab.reluify_maxpools RELUIFY_MAXPOOLS
  --bab.smart_branching SMART_BRANCHING

convert parameters:
  --convert.to {vnnlib,rlv,nnet}
  --convert.dest DEST
  --convert.extended-vnnlib EXTENDED-VNNLIB

eran parameters:
  --eran.domain {deepzono,deeppoly,refinezono,refinepoly}
                        The abstract domain to use.
  --eran.timeout_lp TIMEOUT_LP
                        Time limit for the LP solver.
  --eran.timeout_milp TIMEOUT_MILP
                        Time limit for the MILP solver.
  --eran.use_area_heuristic USE_AREA_HEURISTIC
                        Whether or not to use the ERAN area heuristic.

neurify parameters:
  --neurify.max_depth MAX_DEPTH
                        Maximum search depth for neurify.
  --neurify.max_thread MAX_THREAD
                        Maximum number of threads to use.

nnenum parameters:
  --nnenum.num_processes NUM_PROCESSES
                        Maximum number of processes to use.

verinet parameters:
  --verinet.max_proc MAX_PROC
                        Maximum number of processes to use.
  --verinet.no_split NO_SPLIT
                        Whether or not to do splitting.
```

Only a single verifier can be specified. If a network is specified without a property (i.e., `dnnv --network N /path/to/model.onnx`), then DNNV will print a brief description of the network. This can be useful for understanding the structure of the network to be verified.

2.2 Running DNNV

DNNV can be used to check whether a given property holds for a network. It accepts networks specified in the ONNX format, and properties specified in our property DSL, DNNP, (explained in more detail *here*). Networks can be converted to ONNX format by using native export utilities, such as `torch.onnx.export` in PyTorch, or by using an external conversion tool, such as MMDNN.

We provide several neural network verification benchmarks as example problems, available here.

One of these benchmarks, ERAN-MNIST, is from the evaluation of the ERAN verifier, and have been converted to the DNNP and ONNX formats required by DNNV.

To check a property for a network, using the ERAN verifier, DNNV can be run as:

```
dnnv --eran --network N onnx/pyt/ffnnRELU__Point_6_500.onnx properties/pyt/property_7.py
```

This will check whether `properties/pyt/property_7.py`, a local robustness property, holds for the network `ffnnRELU__Point_6_500.onnx`, a 6 layer, 3000 neuron fully connected network.

DNNV will first report a basic description of the network, followed by the property to be verified. It will then run the specified verifier and report the verification result and the total time to translate and verify the property. The output of the property check above should resemble the output below:

```
$ dnnv --eran --network N onnx/pyt/ffnnRELU__Point_6_500.onnx properties/pyt/property_7.
↪py
Verifying property:
Forall(x_, (((((-0.008 -0.008 ... -0.008 -0.008] [-0.008 -0.008 ... -0.008 -0.008] ...␣
↪[-0.008 -0.008 ... -0.008 -0.008] [-0.008 -0.008 ... -0.008 -0.008]]] < (0.1307 + (0.
↪3081 * x_))) & ((0.1307 + (0.3081 * x_)) < [[[0.008 0.008 ... 0.008 0.008] [0.008 0.
↪008 ... 0.008 0.008] ... [0.008 0.008 ... 0.008 0.008] [0.008 0.008 ... 0.008 0.
↪008]]]) & (0 < (0.1307 + (0.3081 * x_))) & ((0.1307 + (0.3081 * x_)) < 1)) ==> (numpy.
↪argmax(N(x_)) == 9)))

Verifying Networks:
N:
Input_0                         : Input([ 1  1 28 28], dtype=float32)
Transpose_0                     : Transpose(Input_0, permutation=[0 2 3 1])
Reshape_0                       : Reshape(Transpose_0, [ -1 784])
Gemm_0                          : Gemm(Reshape_0, ndarray(shape=(500, 784)),␣
↪ndarray(shape=(500,)), transpose_a=0, transpose_b=1, alpha=1.000000, beta=1.000000)
Relu_0                          : Relu(Gemm_0)
Gemm_1                          : Gemm(Relu_0, ndarray(shape=(500, 500)),␣
↪ndarray(shape=(500,)), transpose_a=0, transpose_b=1, alpha=1.000000, beta=1.000000)
Relu_1                          : Relu(Gemm_1)
Gemm_2                          : Gemm(Relu_1, ndarray(shape=(500, 500)),␣
↪ndarray(shape=(500,)), transpose_a=0, transpose_b=1, alpha=1.000000, beta=1.000000)
Relu_2                          : Relu(Gemm_2)
Gemm_3                          : Gemm(Relu_2, ndarray(shape=(500, 500)),␣
↪ndarray(shape=(500,)), transpose_a=0, transpose_b=1, alpha=1.000000, beta=1.000000)
Relu_3                          : Relu(Gemm_3)
Gemm_4                          : Gemm(Relu_3, ndarray(shape=(500, 500)),␣
↪ndarray(shape=(500,)), transpose_a=0, transpose_b=1, alpha=1.000000, beta=1.000000)
Relu_4                          : Relu(Gemm_4)
Gemm_5                          : Gemm(Relu_4, ndarray(shape=(500, 500)),␣
↪ndarray(shape=(500,)), transpose_a=0, transpose_b=1, alpha=1.000000, beta=1.000000)
```

(continues on next page)

(continued from previous page)

```
Relu_5                          : Relu(Gemm_5)
Gemm_6                          : Gemm(Relu_5, ndarray(shape=(10, 500)),␣
↪ndarray(shape=(10,)), transpose_a=0, transpose_b=1, alpha=1.000000, beta=1.000000)

dnnv.verifiers.eran
  result: unsat
  time: 61.0565
```

Another common option is the `--save-violation /path/to/array.npy` which will save any violation found by a verifier as a numpy array at the path specified. This can be useful for viewing counter-examples to properties and enables performing additional debugging and analysis later.

CHAPTER

THREE

EXAMPLES

In this section, we will go over several examples of properties, and how to check them on a network. We will also discuss the basics of the property DSL.

We have made several DNN verification benchmarks available in DNNP and ONNX formats in dlshriver/dnnv-benchmarks. This benchmark repository includes both ERAN-MNIST and the ACAS Xu benchmark, ready to run with DNNV!

3.1 Local Robustness

Local robustness specifies that, given an input, x, to a DNN, $\mathcal{N}$, any other input within some distance, ϵ, of that input will be classified to the same class. Formally:

$$\forall \delta \in [0, \epsilon]^n . \mathcal{N}(x) = \mathcal{N}(x \pm \delta)$$

This property can be specified in our DSL as follows:

```
from dnnv.properties import *
import numpy as np

N = Network("N")
x = Image(Parameter("input", type=str))
epsilon = Parameter("epsilon", float, default=1.0)

output_class = np.argmax(N(x))

Forall(
    x_,
    Implies(
        ((x - epsilon) < x_ < (x + epsilon)),
        np.argmax(N(x_)) == output_class,
    ),
)
```

3.2 ACAS Xu

Properties other than local robustness can also be specified in DNNP. For example, the properties for the ACAS Xu aircraft collision avoidance network (as introduced in the evaluation of Reluplex) can easily be encoded in DNNP.

Here we write the specification for ACAS Xu Property ϕ_3. The specification states that if an intruding aircraft is directly ahead and moving towards our aircraft, then the score for a Clear-of-Conflict classification (class 0) will not be minimal (this network recommends the class with the minimal score).

In this property, we also see how inputs can be pre-processed. The ACAS Xu networks expects inputs to be normalized by subtracting a pre-computed mean value, and dividing by the given range. We apply that normalization to the input bounds before bounding the network input, `x`.

```
from dnnv.properties import *
import numpy as np

N = Network("N")
# x: $\rho$, $\theta$, $\psi$, $v_{own}$, $v_{int}$
x_min = np.array([[1500.0, -0.06, 3.10, 980.0, 960.0]])
x_max = np.array([[1800.0, 0.06, 3.141593, 1200.0, 1200.0]])

x_mean = np.array([[1.9791091e04, 0.0, 0.0, 650.0, 600.0]])
x_range = np.array([[60261.0, 6.28318530718, 6.28318530718, 1100.0, 1200.0]])

x_min_normalized = (x_min - x_mean) / x_range
x_max_normalized = (x_max - x_mean) / x_range

Forall(
    x, Implies(x_min_normalized <= x <= x_max_normalized, argmin(N(x)) != 0),
)
```

CHAPTER

FOUR

FUNCTION SUPPORT

DNNP supports the application of any python function to concrete variables and values. These are values which are known to DNNV at runtime. This includes things like: literal values such as 1 or 127.5, any imported functions or classes, any objects built from other concrete values, and any Parameter declared in the specification.

Values that are not concrete are symbolic. The most common instance of a symbolic value is the free variable of a quantifier, such as the first argument to `Forall` in a DNNP specification. Any expression that is built from symbolic expressions, or a mix of symbolic and concrete expressions is a symbolic expression. Usually this will include a symbolic variable for the neural network input, and a symbolic expression representing the neural network output (or intermediate outputs). Because symbolic expressions do not have a concrete value, it is not possible to apply just any function to them since there is no actual value on which to apply the function and get a result.

4.1 Python Support

abs, len, max, min, sum

4.2 Numpy Support

We partially support the following functions, however not all parameter options are currently supported.

abs, argmax, argmin, max, mean min, shape, sum

CHAPTER

FIVE

INTRODUCTION TO 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. In this section we will go over this language in detail and describe how properties can be specified in DNNP. To see some examples of common properties specified in this language, check *here*.

Because DNNP extends from Python, it should support execution of arbitrary Python code. However, DNNV is still 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, and can handle all of the DNN properties that we have tried.

5.1 General Structure

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 formula

5.1.1 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.

In general, imported functions and modules can only operate on concrete values, however we do offer *limited support* for the application of some builtin and numpy functions to symbolic values.

5.1.2 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. For example, `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. For example, `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.

5.1.3 Property Formula

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.

Currently DNNV only supports universally quantified properties over a single network input variable. Support for more complex properties is planned.

CHAPTER

SIX

NETWORKS

Networks are a key component to DNNP specifications. Without a network, there is no DNN verification problem. As such, DNNP provides several ways to facilitate working with networks within property specifications. This includes network inference, slicing, and composition. We explain each of these in more detail below.

These operations on Network objects enable DNNP specifications to be specified over production models, rather than models designed specifically for a given specification. DNNP allows models to be used, reduced, or combined to capture the exact semantics of the desired property without having to customize the network model for the specification being written.

6.1 Network Inference

First, and probably most importantly, is that networks can perform inference on inputs. Inference occurs when a network is applied to an input using the python call syntax, i.e., `N(x)`. Inference can be either concrete or symbolic.

Concrete inference occurs when the inputs to the network are concrete values. This occurs when the exact value of the input is known at runtime, such as when it is explicitly defined in the specification (e.g., `x = np.array([[0.0, 1.0, -1.0]])`), or parameterized at runtime (e.g., `x = np.load(Parameter("inputpath", str))`). In this case, inference will compute the concrete output value for the given input.

Symbolic inference occurs when the input is non-concrete. Currently, DNNV only supports symbolic inference when the input is a `Symbol` object. Instances of symbolic inference are extracted during the property reduction process and translated to appropriate constraints for the specified verifier.

6.2 Network Slicing

In DNNP and DNNV, networks are represented as computation graphs, where nodes in the graph are operations and directed edges between nodes represent the data flow between operations. Some of these operations are Inputs, and others are tagged as outputs. DNNP introduces the concept of network slicing, which allows us to select a subgraph of the original network graph. Slicing enables verification at any point in the network, while using the same full network model as input to DNNV. For example, in a six layer network, we could define a property over the output of the third layer, such as some of the properties in Property Inference for Deep Neural Networks by Gopinath et al.

In DNNP, network slicing looks like `N[start:end, output_index]`. The second axis, `output_index` is optional, and can be used when a network has multiple tagged output operations to select a single operation as the output. This is not a common use case, and so in many cases, slicing will more resemble `N[start:end]`. Slicing will take all operations that are at least `start` positions from an `Input` operation (i.e., there is a path with at least `start-1` operations between the operation and an `Input`) and at most `end-1` positions from an `Input`, if both `start` and `end` are positive. A negative index will count backwards from the network outputs, rather than forwards from the inputs. A negative `start` value will select operations that are at most `-start` positions from an output operation (i.e., there is a path with at most `-start-1` operations between the operation and an output), while a negative `end` value will

select operations at least `-end` operations from an output. If `start` is not specified (e.g., `N[:end]`), then slicing will start from the input operations (i.e., `N[0:end]`). If `end` is not specified (e.g., `N[start:]`), then slicing will select the operations from the start index to the current output operations of the network.

6.3 Network Composition

Another potentially useful technique when specifying properties is network composition. This enables networks to be stacked such that the input to one model is the output of another. The following code composes the networks `Ni` and `No` into `N`, such that inference with `N` is equivalent to inference with `Ni` followed by the inference with `No`:

```
Ni = Network("Ni")
No = Network("Ni")
N = No.compose(Ni)

Forall(x, N(x) > 0)
```

One use case for composition is the use of generative models as input constraints to restrict inputs to a learned model of the input space, as introduced by Toledo et al.

CHAPTER

SEVEN

DNN SIMPLIFICATION

In order to allow verifiers to be applied to a wider range of real world networks, DNNV provides tools to simplify networks to more standard forms, while preserving the behavioral semantics of the network.

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, $simplify : \mathcal{G} \to \mathcal{G}$, aims to transform 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 the resulting DNN has more commonly supported structures, $support(G_{\mathcal{N}'}) \geq support(G_{\mathcal{N}})$, where $support : \mathcal{G} \to \mathbb{R}$ is a measure for the likelihood that a verifier supports a structure.

7.1 Available Simplifications

Here we list some of the available simplifications provided by DNNV. Currently all simplfiications are applied to a network, unless the simplification is marked as optional below. Optional simplifications can be enabled with the `DNNV_OPTIONAL_SIMPLIFICATIONS` environment variable. This variable accepts a colon separated list of optional simplifications. For example, to include the optional `ReluifyMaxPool` simplification, set `DNNV_OPTIONAL_SIMPLIFICATIONS=ReluifyMaxPool`.

7.1.1 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.

7.1.2 Identity Removal

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

7.1.3 Convert Add

DNNV converts compatible instances of Add operations to an equivalent Gemm (generalized matrix multiplication) operation.

7.1.4 Convert MatMul Gemm

DNNV converts compatible instances of MatMul (matrix multiplication) 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.

7.1.5 Convert Reshape to Flatten

DNNV converts the sequence of operations Shape, Gather, Unsqueeze, Concat, Reshape into an equivalent Flatten operation whenever possible. This replaces several, often unsupported operations, with a much more commonly supported operation.

7.1.6 Combine Consecutive Gemm

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

7.1.7 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.

7.1.8 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).

7.1.9 Bundle Transpose

DNNV can bundle Transpose operations followed by Flatten or Reshape operations with a successive Gemm operation. This effectively removes the Transpose operation from the network, since some verifiers do not support them.

7.1.10 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.

7.1.11 Reluify MaxPool

This is an optional simplification, which can be enabled by adding `ReluifyMaxPool` to the list of optional simplifiers. This simplification converts MaxPool operations into an equivalent set of Conv and Relu operations. We do this by encoding max ops as $max(a, b) = relu(a - b) + relu(b) - relu(-b)$. This encoding is only a pairwise comparison, and so we cannot translate a single max pool operation into a single convolution operation, we must translate the max pool into a sequence of convolution and relu operations (using this encoding). In general, a MaxPool with a kernel size of n will be converterted to a sequence of $2 * lg(n)$ Conv operations, each followed by a Relu operation.

CHAPTER

EIGHT

PROPERTY REDUCTION

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 or *invalid*.

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.

Reduction enables the application of a broad array of efficient DNN analysis techniques to compute problem validity and/or invalidity.

As defined, reduction has two key properties. The first property is that the set of resulting problems is equivalid with the original verification problem.

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

The second property is that the resulting set of problems all use the same property type. For example, if the desired property type is robustness; all resulting properties assert that $\mathcal{N}(x)_0$ is the output class for all inputs. Applying reduction enables verifiers to support a large set of verification problems by implementing support for this single property type.

8.1 Overview

To illustrate, consider a property for DroNet; a DNN for controlling an autonomous quadrotor. Inputs to this network are 200 by 200 pixel grayscale images with pixel values between 0 and 1. For each image, DroNet predicts a steering angle and a probability that the drone is about to collide with an object. The property states that for all inputs, if the probability of collision is no greater than 0.1, then the steering angle is capped at ± 5 degrees and is specified as:

$$\forall x. ((x \in [0,1]^{40000}) \wedge (\mathcal{N}(x)_{Pcoll} \leq 0.1)) \rightarrow (-5^\circ \leq \mathcal{N}(x)_{Steer} \leq 5^\circ)$$

To enable the application of many verifiers, we can reduce the property to a set of verification problems with robustness properties. This particular example is reduced to two verification problems with robustness properties. Each of the problems produced pair a robustness property (i.e., $\forall x. (x \in [0,1]^4 0000) \rightarrow (\mathcal{N}_0 > \mathcal{N}_1)$) with a modified version of the original DNN. The new DNN is created by incorporating a suffix network that takes in the outputs of the original DNN and classifies whether they constitute a violation of the original property. This suffix transforms the network into a classifier for which violations of a robustness property correspond to violations of the original property.

8.2 Reduction

We rely on three assumptions to transform a verification problem into a reduced form. First, the constraints on the network inputs must be represented as a union of convex polytopes. Second, the constraints on the outputs of the network must be represented as a union of convex polytopes. Third, we assume that each convex polytope is represented as a conjunction of linear inequalities. Complying with these assumptions still enables properties to retain a high degree of expressiveness as unions of polytopes are extremely general and subsume other geometric representations, such as intervals and zonotopes.

We present each step of property reduction below and describe their application to the DroNet example described above. For the purpose of this description we choose to reduce to verification problems with robustness problems. Reducing to reachability problems differs only in the final property, which specifies that the output value $\mathcal{N}(x)_0$ must always be greater than 0.

8.2.1 Reformat the property

Reduction first negates the original property specification and converts it to disjunctive normal form (DNF). Negating the specification means that a satisfying model falsifies the original property. The DNF representation allows us to construct a property for each disjunct, such that if any are violated, the negated specification is satisfied and thus the original specification is falsified. For each of these disjuncts the approach defines a new robustness problem, as described below.

8.2.2 Transform into halfspace-polytopes

Constraints in each disjunct that correspond to constraints over the output are converted to halfspace-polytope constraints, defined over the concatenation of the input and output domains. A halfspace-polytope can be represented in the form $Ax \leq b$, where A is a matrix of k rows, where each row represents 1 constraint, and m columns, one for each dimension in the output space. This representation facilitates the transformation of constraints into network operations. To build the matrix A and vector b, we first transform all inequalities in the conjunction to $\leq$ inequalities with variables on the left-hand-side and constants on the right-hand-side. The transformation first converts $\geq$ to $\leq$ and $>$ to $<$. Then, all variables are moved to the left-hand-side and all constants to the right-hand-side. Next, $<$ constraints are converted to $\leq$ constraints by decrementing the constant value on the right-hand-side. This transformation assumes that there exists a representable number with greatest possible value that is less than the right-hand-side. Finally, each inequality is converted to a row of A and value in b.

8.2.3 Suffix construction

Using the generated halfspace-polytope, we build a suffix subnetwork that classifies whether outputs satisfy the specification. The constructed suffix has two layers, a hidden fully-connected layer with ReLU activations, and dimension equal to the number of constraints in the halfspace-polytope defined by the current disjunct, and a final output layer of size 2.

The hidden layer of the suffix has a weight matrix equal to the constraint matrix, A, of the halfspace-polytope representation, and a bias equal to $-b$. With this construction, each neuron will only have a value greater than 0 if the corresponding constraint is not satisfied, otherwise it will have a value less than or equal to 0, which becomes equal to 0 after the ReLU activation is applied. In the DroNet problem for example, one of the constraints for a disjunct is $(\mathcal{N}(x)_S \leq -5^\circ)$. For this conjunct we define the weights for one of the neurons to have a weight of 1 from $\mathcal{N}(x)_S$, a weight of 0 from $\mathcal{N}(x)_P$, and a bias of 5°.

The output layer of the suffix has 2 neurons, each with no activation function. The first of these neurons is the sum of all neurons in the previous layer, and has a bias value of 0. Because the neurons in the previous layer each represent a constraint, and each of these neurons is 0 only when the constraint is satisfied, if the sum of all these neurons is 0, then

the conjunction of the constraints is satisfied, indicating that a violation has been found. The second of these neurons has a constant value of 0 – all incoming weights and bias are 0. The resulting network will predict class 1 if the input satisfies the corresponding disjunct and class 0 otherwise.

8.2.4 Problem construction

The robustness property specification states that the network should classify all inputs that satisfy the input preconditions as class 0 – no violations. If a violation is found to this property, then the original property is violated by the input that violated the robustness property. In the end, we have generated a set of verification problems such that, if any of the problems is violated, then the original problem is also violated. This comes from our construction of a property for each disjunct in the DNF of the negation of the original property.

CHAPTER

NINE

ADDING A NEW REDUCTION

TODO. Sorry, this page is still in development. An example reduction can be seen here.

In general a reduction will subclass the `Reduction` base class and implement the method `reduce_property(self, phi: Expression) -> Iterator[Property]`, which takes in a property expression and returns an iterator of `Property` objects. A reduction will likely also require a custom `Property` type which must implement `validate_counter_example(self, cex: Any) -> Tuple[bool, Optional[str]]`.

CHAPTER

TEN

ADDING A NEW DNN SIMPLIFIER

TODO. Sorry, this page is still in development. An example simplification can be seen here.

In general a DNN simplification will subclass the `Simplifier` base class. This class implements a visitor pattern for an operation graph. In general, a simplifier implementation will check if an operation sub-graph matches some pattern, and, if so, will replace it with a semantically equivalent sub-graph.

CHAPTER

ELEVEN

ADDING A NEW VERIFIER

TODO. Sorry, this page is still in development. As an example, the implementation for the planet verifier can be seen here.

In general a verifier will subclass the `Verifier` base class and implement at least the methods `build_inputs(self, prop)` and `parse_results(self, prop, results)`. When verifying a property, the base verifier implementation simplifies the network, and reduces the property to a set of properties with hyper-rectangles in the input space and a halfspace polytope in the output space. Each property has methods to add a suffix to the network for reduction to robustness properties, as well as a method to add a prefix that modifies the input domain to be a unit hyper cube.

CHAPTER

TWELVE

PUBLICATIONS

The underlying techniques used in DNNV are described in several publications. Additionally, DNNV has been used by several works to facilitate the implementation of new techniques, as well as the usage and comparison of many verification tools on novel problems.

- Property reduction for DNN properties was first introduced in Reducing DNN Properties to Enable Falsification with Adversarial Attacks.
- The DNNV tool is introduced in DNNV: A Framework for Deep Neural Network Verification, along with overviews of the DNN simplifications and property reductions used.
- The paper Systematic Generation of Diverse Benchmarks for DNN Verification used DNNV to quickly run many verification tools on the benchmarks generated by their technique.
- The paper Distribution Models for Falsification and Verification of DNNs used DNNV to evaluate their approach for specifying properties over complex input domains, such as images.