Reducing DNN Properties to Enable Falsification with Adversarial Attacks

This artifact accompanies the paper *Reducing DNN Properties to Enable Falsification with Adversarial Attacks*. In this artifact, we provide the benchmarks and scripts for reproducing the results of our study, and we also provide our tool, DNNF for running falsification methods such as adversarial attacks on DNN property specifications specified using the DNNP language of DNNV.

While many DNN verification techniques have been introduced in the past few years to enable the checking of DNN safety properties, these techniques are often limited in their applicability, due to simplifying assumptions about DNN structure or to high computational cost. Falsification is a complementary approach to verification that seeks only to find violations to a safety property. In the context of DNNs, adversarial attacks can be viewed as falsifiers for DNN local robustness properties. While these techniques often scale to large real-world DNNs, they are currently limited in the range of properties they can falsify.

In *Reducing DNN Properties to Enable Falsification with Adversarial Attacks*, we introduce an approach for reducing a DNN and an associated safety property -- a correctness problem -- into an equivalid set of correctness problems formulated with robustness properties which can be processed by existing adversarial attack techniques. We implement the approach in a tool which we call DNNF, and we perform a study demonstrating that property reduction yields a cost-effective approach to find violations of DNN correctness problems.

Install

We recommend using the provided Ubuntu 20.04 VirtualBox VM image with the artifact pre-installed. Installation instructions are in included in `INSTALL.pdf`.

Execution

Open a terminal window in the provided virtual machine. The DNNF tool can then be run as follows:

```
$ python -m dnnf PROPERTY --network NAME PATH
```

Where `PROPERTY` is the path to the property specification, `NAME` is the name of the network used in the property specification (typically `N`), and `PATH` is the path to a DNN model in the ONNX format.

To see additional options, run:

```
$ python -m dnnf -h
```

Benchmarks

We provide the property and network benchmarks used in our evaluation here. These benchmarks are already included in the provided VM.

The 4 benchmarks are split into 4 directories, 1 for each benchmark. Each of these directories has a subdirectory, `onnx` that contains the networks used in the benchmark in the ONNX format, another subdirectory, `properties`, that contains the properties used in the benchmark in the DNNP format, and a csv file, `properties.csv`, that lists the property and network pairs that make up the benchmark. The csv file has 4 columns. The first column, `problem_id`, gives a unique name to each problem in the benchmark. The second column, `property_filename`, specifies the property to use. The third and fourth columns, `network_names` and `network_filenames`, specify the name of the network used in the property specification and the path to the ONNX formatted network respectively.

Running the Tool

To execute DNNF on a problem in one of the benchmarks, first navigate to the desired benchmark directory in `artifacts` (i.e., `acas_benchmark`, `neurifydave_benchmark`, or `ghpr_benchmark`). Then run DNNF as specified above. For example, to run DNNF with the Projected Gradient Descent adversarial attack from cleverhans on an ACAS property and network, run:

```
$ cd artifacts/acas_benchmark
$ python -m dnnf properties/property_2.py \
> --network N onnx/N_3_1.onnx \
> --backend cleverhans.ProjectedGradientDescent
```

Which will produce output similar to:

```
Falsifying: Forall(x0, (((x0 <= [[ 0.68 0.5  0.5  0.5 -0.45]]) &
([[ 0.6 -0.5 -0.5  0.45 -0.5 ]] <= x0)) ==> (numpy.argmax(N(x0)) != 0)))

dnnf
  result: sat
  time: 2.6067
```

Several warnings may be produced by some of DNNF's dependencies, which can be safely ignored. The `-q` option should suppress most of these warnings.

The available backends for falsification are:

- `cleverhans.LBFGS`, which also requires setting parameters `--set cleverhans.LBFGS y_target "[[-1.0, 0.0]]"`
- `cleverhans.BasicIterativeMethod`
- `cleverhans.FastGradientMethod`
- `cleverhans.DeepFool`, which also requires setting parameters `--set cleverhans.DeepFool nb_candidate 2`
- `cleverhans.ProjectedGradientDescent`
- `tensorfuzz`

If a property uses parameters, then the parameter value can be set using `--prop.PARAMETER=VALUE`, e.g., `--prop.epsilon=1`.

The verifiers can be run using DNNV. For example, to run the ERAN deepzono verifier on the same ACAS property and network as above, run:

```
$ cd artifacts/acas_benchmark
$ python -m dnnv onnx/N_3_1.onnx properties/property_2.py --eran
```

Which should produce output similar to:

```
Verifying Network:
Input_0                         : Input([1 5], dtype=float32)
Gemm_0                          : Gemm(Input_0, ndarray(shape=(50, 5)),
ndarray(shape=(50,)))
Relu_0                          : Relu(Gemm_0)
Gemm_1                          : Gemm(Relu_0, ndarray(shape=(50, 50)),
ndarray(shape=(50,)))
Relu_1                          : Relu(Gemm_1)
Gemm_2                          : Gemm(Relu_1, ndarray(shape=(50, 50)),
ndarray(shape=(50,)))
Relu_2                          : Relu(Gemm_2)
Gemm_3                          : Gemm(Relu_2, ndarray(shape=(50, 50)),
ndarray(shape=(50,)))
Relu_3                          : Relu(Gemm_3)
Gemm_4                          : Gemm(Relu_3, ndarray(shape=(50, 50)),
ndarray(shape=(50,)))
Relu_4                          : Relu(Gemm_4)
Gemm_5                          : Gemm(Relu_4, ndarray(shape=(50, 50)),
ndarray(shape=(50,)))
Relu_5                          : Relu(Gemm_5)
Gemm_6                          : Gemm(Relu_5, ndarray(shape=(5, 50)),
ndarray(shape=(5,)))

Verifying property:
Forall(x0, (((([[ 0.6 -0.5 -0.5  0.45 -0.5 ]] <= x0) &
(x0 <= [[ 0.68 0.5  0.5  0.5 -0.45]])) ==> (numpy.argmax(N(x0)) != 0)))
...
```

```
dnnv.verifiers.eran
  result: unknown
  time: 2.5711
```

Different verifiers can be used by replacing `--eran` with `--VERIFIER`, where `VERIFIER` can be one of the following:

- `eran`
- `neurify`
- `planet`
- `reluplex`

Just like with DNNF, if a property uses parameters, then the value can be set using `--prop.PARAMETER=VALUE`, e.g., `--prop.epsilon=1`.

Replicating the Evaluation

To run the full evaluation in our paper (*WARNING: this may take several hundred hours*), run:

```
$ scripts/run_all.sh
```

This script will sequentially run all falsifiers and verifiers on all benchmarks. It will save results in the `results/` directory, as comma separated values files. There will be one file for each method and benchmark variant. These files can be combined into a single csv by running the following in the root directory:

```
$ python tools/combine_results.py
```

Which will generate a file called `results.csv` in the current directory. This CSV file will have 6 columns: - `Artifact` specifies the artifact being run, e.g., ACAS Xu - `Variant` specifies a variant of the artifact, e.g., DroNet or MNIST for GHPR - `ProblemId` specifies an identifier for the problem being checked - `Method` specifies the method used to check the problem - `Result` specifies the result of falsification or verification - `TotalTime` specifies the time to generate a result

If you have access to a cluster with slurm, execution may be sped up by running script `scripts/run_all_slurm.sh`, which will launch slurm jobs rather than running each technique sequentially.