nn-dependability-kit manual

(Public version)

January 31st, 2019

1 Introduction

nn-dependability-kit is a research tool with the goal of assisting engineering neural networks for safety-critical domains.

2 Availability and License

The online version of the tool is publicly released under the GNU Affero General Public License (AGPL) Version 3, which is a strong copyleft open source license.

`https://github.com/dependable-ai/nn-dependability-kit/`

3 Dependencies and Installation

Here we list dependencies of nn-dependability-kit:

(Basics) `PyTorch` 0.4.x + `Numpy` + `matplotlib` + `jupyter`.

(Test case generation) Google optimization research tools, available at `https://developers.google.com/optimization/introduction/installing/binary`

(Verification / static analysis) `PuLP` (python-based MILP connector to Coin-or branch and cut (CBC) solver and other solvers)

- For Ubuntu users, the CBC solver shipped with PuLP may crash in solving some problems. Therefore, please additionally install GNU GLPK. The static analysis engine assumes that the GLPK solver is installed in the default directory "`/usr/local/bin/glpsol`". Whenever CBC solver crashes, GLPK solver is automatically triggered as a replacement.

- For academic users, we strongly advise to use IBM CPLEX as the underlying solver, as academic partners can retrieve IBM CPLEX for free. For open source solvers, we have experienced multiple times unexpected crashes. Even the above workaround cannot guarantee a crash-free behavior.

(Run-time verification) `dd` (binary decision diagram implemented using python)

nn-dependability-kit mainly supports PyTorch. For TensorFlow, the support is currently restricted to formal verification. However, as the input for computing the metrics and the input for building monitors uses numpy arrays rather than native pytorch tensors, it is very easy to use these metrics or to build runtime monitors also with TensorFlow.

4 Examples

Within the source code of nn-dependability-kit, examples are presented as jupyter notebooks to allow step-by-step execution with explanations written as markdown languages.

5 How to use nn-dependability-kit

In the following subsections, we explain the capability of nn-dependability-kit, by traversing the package structure while summarizing key functionality within each package.

5.1 Package `basic`

Package `basic` contains intermediate representation of neural networks (`neuralnet.py`), as well as converters from Pytorch or Tensorflow models to the intermediate representation used by nn-dependability-kit. The intermediate representation is currently only *used for formal verification purposes*, and it supports only multi-layer perception (MLP) with the following types of layer descriptions.

- Rectified Linear Unit (ReLU)
- Exponential linear unit (ELU)
- Linear
- Batch normalization layer (BN)

Note that nn-dependability-kit also *accepts networks with convolution layers*, when one wants to perform testing, metric, or runtime monitoring. The intermediate representation is *only used for verification purposes* - the background is that we advise to do *assume-guarantee style* formal safety verification by taking only close-to-output layers in a network, and the above types are sufficient to capture close-to-output layers.

```
from nndependability.basic import tensorflowreader

inputShape = (600, 480, 3)
tfModelPath = "frozen_graph.pb"
layersNames = ["model/FC10/fully_connected",
               "model/FC11/BatchNorm",
               "model/Output/fully_connected"]
layerTypes = ["elu", "BN-gamma1", "linear"]
# BN-gamma1 is a special case where BN has parameter gamma being 1
model = tensorflowreader.loadMlpFromTensorFlow(tfModelPath,
                              layersNames, layerTypes, inputShape)
```

5.2 Package `metrics`

Package `metrics` includes functions to compute dependability metrics for neural networks.

5.2.1 Perturbation loss metric

Perturbation loss metric contains a set of predefined perturbation directions (representing different effects) and predefined quantity (representing different intensity; can be changed if needed). The goal of this metric is to understand if the neural network under analysis is still able to perform correctly with (slightly) perturbed input. An example of using such a metric can be found in the jupyter notebook `GTSRB_AdditionalMetrics.ipynb`.

The below code snippet explains the workflow. Once when one initiates a metric object, to compute the perturbation loss metric, simply add input images and corresponding labels (both as numpy arrays) to the metric object. There are multiple options for visualizing the metric, such as computing the maximum or average drop of output probability.

```
from nndependability.metrics import PerturbationLoss
metric = PerturbationLoss.Perturbation_Loss_Metric()

...

# Let image be the input of the network, represented as numpy arrays
# Let label be the ground truth of the network,
# also represented as numpy arrays

metric.addInputs(net, image, label)

...
# Print the maximum quantity of probability drop
metric.printMetricQuantity("MAX_LOSS")
```

```
# Print the average quantity of probability drop
metric.printMetricQuantity("AVERAGE_LOSS")
```

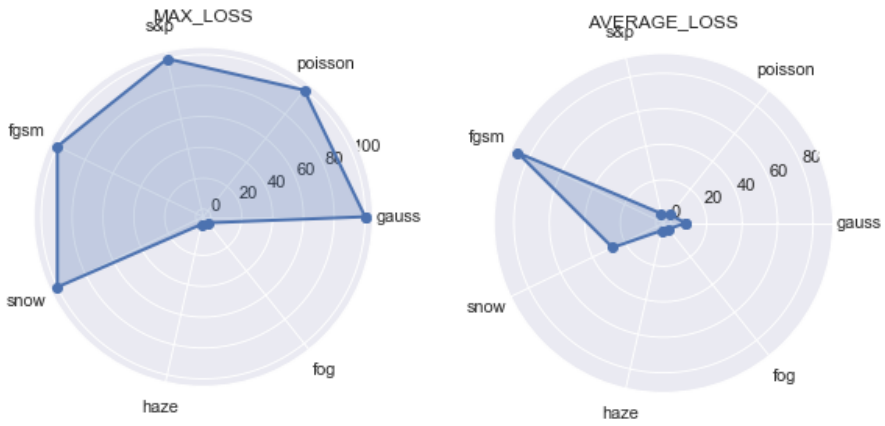

Figure 1: Perturbation loss metric for a traffic sign classification network.

Figure 1 shows the resulting diagram for perturbing a neural network with 7 directions. The `MAX_LOSS` diagram indicates that there exists an image in the data set such that perturbing the image by adding the Gaussian noise (`gauss` in Figure 1) leads to 100% confidence drop, i.e., the network considers the original image to be of class A with 100% probability but the perturbed image is considered to be of class A with 0% probability.

5.2.2 Neuron activation pattern metric

Neuron activation pattern metric examines, for input data that is classified into the same class, whether their activation pattern is similar. Due to combinatorial explosion, the similarity of activation patterns is further aggregated into histograms capturing the number of neurons being activated, based on analyzing a user-specified close-to-output neuron layer. An example of using such a metric can be found in the jupyter notebook `GTSRB_AdditionalMetrics.ipynb`.

Similar to the code snippet below, one first needs to initiate the metric class, and specify the number of neurons to be monitored (`sizeOfNeuronsToMonitor`). Subsequently, feed the metric class with 2D numpy arrays where the 1st dimension matches `sizeOfNeuronsToMonitor` and the 2nd dimension captures the batch size.

```
from nndependability.metrics import NeuronActivationPattern
```

```
metric = NeuronActivationPattern.Neuron_Activation_Pattern_Metric
                                (num_classes, sizeOfNeuronsToMonitor)

...

# Let "intermediateValues" be the computed pytorch tensor for a
# particular layer, "predicted" be the originally predicted result,
# and "labels" be the ground truth

metric.addAllNeuronPatternsToClass(intermediateValues.numpy(),
                                   predicted.numpy(), labels.numpy())

...
# Let "class" be an integer value indicating the class index.
metric.printMetricQuantity(class)
```

Figure 2 shows one histogram for all images being classified by the neural network to class indexed 0.

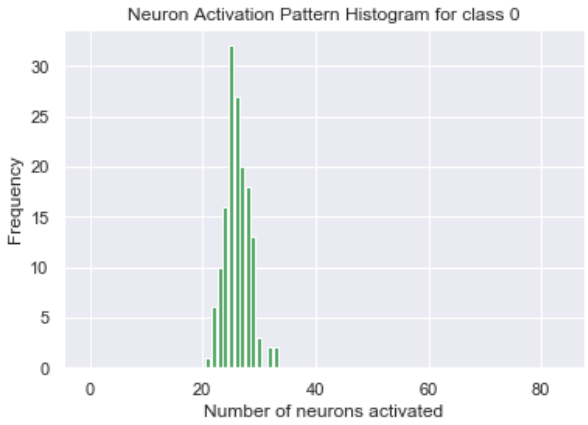

Figure 2: Neuron activation pattern represented as a histogram.

5.2.3 Scenario k-projection metric

Scenario k-projection coverage considers how the input data (for training or for testing purposes) are appropriately distributed, where appropriateness is defined via the concept of k-projection, an idea that long appears in software testing with a different name (combinatorial testing). An example of using such a metric can be found in the jupyter notebook `KITTI_Scenario_Coverage.ipynb`.

The background of using k-projection is that, when one tries to categorize input by discretizing scenarios into sequences of scenes and by discretizing

conditions in a scene (e.g. the weather can be "sunny", "cloudy", "snowy"). With 50 discrete conditions, one creates equivalence classes up to size 2^{50}. Such a huge number clearly shows that it is impossible to achieve 100% coverage even when one wants to cover each equivalent class with one test case. Therefore, one would like to have a "relative form" of completeness, where one is able to argue "achieving 100% coverage" but the confidence of completeness is lower[1]. As k-projection seeks to find input data that can cover arbitrary k discrete conditions, by lowering the k-value in the k-projection coverage metric, one can create such a weaker form of completeness.

To use the metric, one needs to prepare at least two files.

A. Scenario description This file lists all possible discrete conditions that can be used to build equivalence classes.

B. Scenarios This file includes multiple scenarios that should be counted in computing the coverage.

C. Domain restriction (optional) This file lists constraints where certain combinations of discrete conditions are not possible. For example, it is impossible for the vehicle to be in the 3rd lane (`egoVehicleCurrentLane=3`), if the road only allows a two-lane drive (`numTotalLanes=2`). Such a condition can be specified as constraints.

The below code snippet demonstrates the workflow. One follows the same process by first initiating a metric object. By default the k value equals 2, but one can pass additional parameters to change the k value. By adding scenarios using function `addScenariosFromFile`, the 2-projection coverage is immediately computed. Note that the output result has a denominator with 6772; the console mentions that it does not consider domain restriction. When one considers domain restriction, the solver needs to check for each grid out of 6772 grids, whether it is possible to be realized by considering constraints stated in the domain restriction. With domain restriction as constraints, checking each grid is realizable is itself an NP-complete problem.

```
from nndependability.metrics import ScenarioKProjection

# Initiate the metric class by informing the discrete partitioning,
# which is specified in the file description.xml

metric = ScenarioKProjection.Scenario_KProjection_Metric("description.xml")

# Let scenarios.xml contains multiple scenarios to be added, where
# each scenario is created by a mapper function which translates real
```

[1] An analogy appears in testing safety critical software, where compared to path coverage or MC/DC coverage, line coverage is a weaker form - it is easier to achieve "100%" but the achieved "100%" provides a weaker guarantee.

```
# data to discrete categories.

metric.addScenariosFromFile("scenarios.xml")

# The 2-projection coverage is computed, right after new
# scenarios are added.


>> 2-projection coverage (without domain restrictions): 660/6772
```

5.2.4 Neuron k-projection on-off activation metric

For arguing the completeness of test cases, one possible approach is to examine if all test cases can create all possible neuron activation patterns, for a given neuron layer under analysis. Again such a problem is combinatorial in nature (if a layer has 50 neurons, then the number of possible abstract activation patterns equals 2^{50}, when the value abstraction is done by setting an output to true/activate if its original ReLU output value is greater than 0, and false/deactivate otherwise).

The concept of k-projection again plays a role of providing a weaker form of completeness, in order to combat combinatorial explosion:

- When $k = 1$, to achieve full coverage, one requires the set of test cases to be able to let each neuron have activated as well as deactivated state (e.g., test case 1 lets neuron 1 activate; test case 2 lets neuron 1 deactivate). This is thus equivalent to *neuron coverage* as stated in existing literature.
- When $k = 2$, to achieve full coverage, one requires the set of test cases to be able to let every neuron pair to cover four situations, namely (`activate`, `activate`), (`activate`, `deactivate`), (`deactivate`, `activate`), and (`deactivate`, `deactivate`).
- When $k = n$, where n is the size of neurons in the specified layer, one returns to the original combinatorial explosion setup.

An example of using such a metric can be found in the jupyter notebook `GTSRB_Neuron2ProjectionCoverage_TestGen.ipynb`. Similar to neuron activation pattern metric, one starts by specifying the number of neurons in a particular layer to be monitored (`numNeurons`). Subsequently, feed the metric class with 2D numpy arrays where the 1st dimension matches `numNeurons` and the 2nd dimension is the batch size of the test set.

```
from nndependability.metrics import NeuronKProjection

k_Value = 2
metric = NeuronKProjection.Neuron_OnOff_KProjection_Metric(k_Value, numNeurons)
```

```
...
# Let "intermediateValues" be the computed pytorch tensor for a particular layer
metric.addInputs(intermediateValues.numpy())

# Print the current metric value
metric.printMetricQuantity()
```

```
>> Current input size fed into the metric: 3264
>> 2-projection neuron on-off activation coverage: 11711/13944=0.8398594
```

5.3 Package `atg`

Package `atg` contains functions to perform *automatic test case generation*. Test case generation and metric computation are highly related, as in engineering practice, one first compute metrics, followed by finding new test cases that (maximally) increase the currently computed metric value.

5.3.1 Increase scenario k-projection coverage

Recall in Section 5.2.3 where one computes how currently available scenarios has covered the discrete input space using the concept of k-projection metric. The below code snippet follows the example in Section 5.2.3 where after the metric is computed, one can trigger the test case generator via function `proposeScenariocandidate`. The generated outputs are concrete variable assignments that try to maximally increase the specified coverage, with a search time of 10 seconds (the timeout value can be changed). Notice that one should add the domain restriction file (in the code below, `domain-restrictions.xml`), to ensure that the generated scenario is feasible.

```
metric.addDomainRestrictionsFromFile("domain-restrictions.xml")

...
from nndependability.atg.scenario import scenariogen
variableAssignment = scenariogen.proposeScenariocandidate(metric)
metric.writeScenarioToFile(variableAssignment, "tmp.xml")
```

```
>> Timeout but feasible solution found in 10 seconds
>> Maximum possibility for improvement = 6485
>> Optimal objective value computed from IP = 189

>> for criterion ego_vn, set it to -3
>> for criterion ego_ve, set it to -2
>> for criterion ego_ax, set it to 0
>> for criterion ego_ay, set it to -3
```

```
>> for criterion left_box_existence, set it to True
>> for criterion left_box_rotation, set it to 1
>> for criterion left_box_type, set it to DontCare
>> for criterion left_box_occlusion, set it to 2
>> for criterion right_box_existence, set it to True
>> for criterion right_box_rotation, set it to 0
>> for criterion right_box_type, set it to Misc
>> for criterion right_box_occlusion, set it to 3
>> for criterion closest_box_1_existence, set it to True
>> for criterion closest_box_1_rotation, set it to 0
>> for criterion closest_box_1_type, set it to Misc
>> for criterion closest_box_1_occlusion, set it to 3
>> for criterion closest_box_2_existence, set it to True
>> for criterion closest_box_2_rotation, set it to 6
>> for criterion closest_box_2_type, set it to DontCare
>> for criterion closest_box_2_occlusion, set it to 2
```

5.3.2 Increase neuron k-projection coverage

Recall in Section 5.2.4 where one computes the neuron k-projection coverage. Similar to Section 5.3.1, one can use the test case generator to propose a certain neuron activation pattern that maximally increases the coverage, via function `proposeNAPcandidate`. For the example below, the solver suggests a pattern that sets neuron 0 to value 0 (deactivate), neuron 1 to value 1 (activate).

```
from nndependability.atg.nap import napgen

napgen.proposeNAPcandidate(metric)
```

```
>> for neuron 0, set it to 0
>> for neuron 1, set it to 1
>> ...
>> for neuron 29, set it to 1
>> for neuron 30, set it to 1
```

Nevertheless, one remaining task is to find an input that matches the suggested pattern. For scenario k-projection, the task amounts to collecting data from real world or synthesizing data from a simulator, such that the new data match the proposed scenario. For neuron activation pattern, similar to what the below code snippet demonstrates, nn-dependability-kit offers an input generation tool (`gratestgen`) that can continually modify an input image (`img`) until the neuron activation pattern of the modified image matches the user specified pattern (the pattern information is stored in `targetedNeuronIndex` and `desiredNAP`).

```
from nndependability.atg.gradient import gratestgen
```

```
targetedNeuronIndex= list()
desiredNAP = list()

# Specify the target - we want to control neuron 3 and neuron 4,
# with sign being positive and negative
targetedNeuronIndex.append(3)
targetedNeuronIndex.append(4)

desiredNAP.append(1)
desiredNAP.append(-1)

# Trigger gradient-based test case generation
new_img, isSuccessful = gratestgen.generateTestCase(img, targetedNeuronIndex,
                                                    desiredNAP, net)
```

(Practical Considerations) To generate inputs that match the specified neuron activation pattern, the most plausible way is to start with an input whose neural activation pattern is already close to the target pattern to be reached. Then it is more likely for the test case generator to be successful by only perturbing the image with a small amount of noise.

5.4 Package `formal`

Package `formal` contains formal reasoning engines based on static analysis. Currently for static analysis, both boxed domain (managing abstractions of shape $L \leq x_i \leq U$) and octagon domain (boxed domain plus holding constraints of the form $L \leq x_i \pm x_j \leq U$) are supported for layer-wise reasoning. An example of formal analysis can be found in `TargetVehicleProcessingNetwork_Formal Verification.ipynb`, where one uses formal verification to prove the absence of erroneous behaviors.

(Specification) For nn-dependability-kit, currently the static analysis supports *safety verification*, meaning that the static analysis engine checks if there exists an input assignment in that satisfies ϕ_{input}, but the output from the neural network (by feeding in) satisfies the provided risk property ρ^{risk}_{output}.

- If not, then the system is guaranteed to be safe against the risk property ρ^{risk}_{output}.
- If yes, then due to the use of abstract interpretation, the engine reports unknown, meaning that it can still be possible for the system to be safe against ρ^{risk}_{output} (i.e., the counter example is not genuine), but further techniques (such as CEGAR) should be executed to identify the validity of the generated counter example and to refine the proof.

The input property ϕ_{input} is currently characterized by a *conjunction of linear inequalities* over input variables. Similarly, the risk property ρ^{risk}_{output} is characterized by a conjunction of linear inequalities over output variables. To trigger the verification engine, one should provide the following information.

- (Specify input property) The input property contains three items, namely
  - `inputMinBound`: a numpy array for specifying the value lower bound for each input variable,
  - `inputMaxBound`: a numpy array for specifying the value upper bound for each input variable, and finally
  - `inputConstraints`: a list of additional linear constraints. For example, `[[3, "==", 1, "in0"], [5, "<=", 1, "in2", -6, "in4"]]` is used to charterize constraints $(in_0 = 3) \wedge (in_2 - 6\, in_4 \geq 5)$.
- (Specify risk property) Risk property `riskProperty` is a list of linear constraints over output variables. For example, `[[0, "<=", 1, "out18", -1, "out0"], [0, "<=", 1, "out18", -1, "out1"]]` is used to charterize constraints $(out_{18} - out_0 \geq 0) \wedge (out_{18} - out_1 \geq 0)$.

Following the below code snippet, one triggers the static analysis engine with an additional parameter `isBoxedDomain`, which is set to `True` if one wishes to apply static analysis on the boxed domain. When `riskProperty` is an empty list, the function `verify` returns the minimum and the maximum possible value of each output neuron, based on the result derived by static analysis.

```
from nndependability.formal import staticanalysis

...
# Start formal reasoning engine
staticanalysis.verify(inputMinBound, inputMaxBound, net, isBoxedDomain,
                                                  inputConstraints, riskProperty)
```

5.5 Package `rv`

Package `rv` allows building runtime monitors with the goal of understanding whether a decision made by a neural network is *supported by prior similarities as appeared in the training data*. An example of building and using runtime monitors can be found in `GTSRB_RuntimeMonitoring.ipynb`.

The below code snippet demonstrates the workflow. One initiates a *neuron activation pattern (NAP)* monitor which stores, for all input data being classified to the same category, their abstract activate/deactivate status (over each neuron being monitored). Such information is stored using binary decision diagrams (BDD) for memory efficiency considerations and for post-processing purposes.

During the construction of an NAP monitor, one feeds the monitor with 2D numpy arrays (`intermediateValues`) where the 1st dimension stores raw output values of neurons, and the 2nd dimension captures the batch size. Internally, the raw value is translated into `true` if the raw value is greater than 0 (activated), and is translated into `false` otherwise. Note that the parameter `neuronIndicesToBeOmitted` in the constructor is used, when a user decides that on-off valuations of certain neurons are unimportant. Therefore, for neurons whose indices are in `neuronIndicesToBeOmitted`, their translated `true`/`false` value are not used in BDD encoding of the pattern.

```
from nndependability.rv import napmonitor
monitor = napmonitor.NAP_Monitor(num_classes, sizeOfNeuronsToMonitor,
                                 neuronIndicesToBeOmitted)

# When set to -1, build monitor for every class.
classToBeMonitored = -1

# Let "intermediateValues" be the computed pytorch tensor for a
# particular layer, "predicted" be the originally predicted result,
# and "labels" be the ground truth

monitor.addAllNeuronPatternsToClass(intermediateValues.numpy(),
                                    predicted.numpy(), labels.numpy(),
                                    classToBeMonitored)
```

Once when the monitor is built by reading all neuron activation patterns derived from the training data set, one can perform the following two actions.

(Containment checking) In operation time, one may want to test if for a particular image being classified by the neural network to class `A`, its activation pattern has appeared in the monitor. Simply use function `isPatternContained(valueVector, A)` for containment checking where `valueVector` contains values for neurons.

(Enlarge monitor) Use `enlargeSetByOneBitFluctuation(classToBeEnlarged)` to create an enlarged monitor that additionally accepts inputs with their on-off activation patterns having 1-bit variation.