Tutorials on Testing Neural Networks

10th May 2021

University of Liverpool

Abstract

Deep learning achieves remarkable performance on pattern recognition, but can be vulnerable to defects of some important properties such as robustness and security. This tutorial is based on a stream of research conducted since the summer of 2018 at a few UK universities, including the University of Liverpool, University of Oxford, Queen's University Belfast, University of Lancaster, University of Loughborough, and University of Exeter. This research is supported by the Defence Science and Technology Laboratory (Dstl), UK.

The research aims to adapt software engineering methods, in particular software testing methods, to work with machine learning models. Software testing techniques have been successful in identifying software bugs, and helping software developers in validating the software they design and implement. It is for this reason that a few software testing techniques – such as the MC/DC coverage metric – have been mandated in industrial standards for safety critical systems, including the ISO26262 for automotive systems and the RTCA DO-178B/C for avionics systems. However, these techniques cannot be directly applied to machine learning models, because the latter are drastically different from traditional software, and their design follows a completely different development life-cycle.

As the outcome of this thread of research, the team has developed a series of methods that adapt the software testing techniques to work with a few classes of machine learning models. The latter notably include convolutional neural networks, recurrent neural networks, and random forest. The tools developed from this research are now collected, and publicly released, in a GitHub repository: https://github.com/TrustAI/DeepConcolic, with the BSD 3-Clause licence.

This tutorial is to go through the major functionalities of the tools with a few running examples, to exhibit how the developed techniques work, what the results are, and how to interpret them.

Contributors

The following is a list of authors who have contributed to the software and the tutorial (their names are ordered alphabetically): Nicolas Berthier, Wei Huang, Xiaowei Huang, Wenjie Ruan, Youcheng Sun, Yanghao Zhang.

Contents

1 Introduction

Safety-critical Systems are those systems whose failure could result in loss of life, significant property damage, or damage to the environment. Aircraft, cars, weapons systems, medical devices, and nuclear power plants are the traditional examples of safety-critical software systems. The failures of safety-critical software systems have brought some companies and their software development practices to the attention of the public, *e.g.,* Boeing's two 737 Max groundings [1] and the crash of Uber's self-driving car [2].

In such systems, one ideally seeks an absolute certainty that the system meets its specification. In the case of software systems, this notably entails that the software does not contain any bug, or other internal sources of failure. In the worst case, if such a failure is permitted to occur, one wants to ensure that it is properly detected, and that appropriate counter-measures are deployed.

Testing for Safety-critical Systems Testing is still the primary approach in industry for increasing the confidence in software products and services. Obviously, tests are non-exhaustive in every relevant practical cases (and almost by definition). Yet, that does not prevent them from being pervasive even in safety-critical contexts where formal verification techniques are available to assess the correct behaviours of systems. The main underlying reasons are twofold:

(i) the successful design of test beds serves as an assessment that the specification is realistic (*i.e.,* it can be translated into a set of test cases);

(ii) test beds are often the only "executable specification" that designers have access to in the implementation of the system itself. As such, they help increase the level of confidence that the design meets its specification.

We can draw some interesting observations from the above two points and the usual development process for ML systems. Indeed, the core assumption of such systems, which states that expected system behaviours can be learned from data via training, exactly mirrors point (i). Furthermore, point (ii) refers to the iterative and empirical process for designing ML systems, that often starts from a bare neural network architecture, and evolves according to its results in meeting various requirements—*e.g.,* prediction accuracy, robustness to adversarial examples—via a series of architectural refinements.

Unfortunately, the parallel stops here: one step that is missing above for ensuring the safe operational use of a safety-critical system consists in obtaining a formal guarantee that it meets its specification, and this has no equivalent in the domain of machine learning systems. One notable reason is that existing code coverage criteria for testing conventional software cannot realistically be applied to machine learning.

Robustness Concern One of the desirable properties that ML systems should satisfy is *robustness.* This concern was first revealed by Szegedy et al. [10] when they showed that

machine learning (ML) models, and in particular deep neural networks, may be vulnerable to *adversarial attacks*, where small input perturbations cause the models to behave in an unintended way. In the case of a model that achieves a *classification task*, for instance, such an unintended behaviour is typically a misclassification of the perturbed input. More generally, we can say that a *classifier* is non-robust if it associates two very similar inputs (*e.g.,* two images that differ by very few pixels or brightness) with two distinct labels.

Coverage-guided Testing Most of the tools we shall present address the validation of robustness via a series of *coverage-guided testing* algorithms, where a predefined quantification of model behaviours is used to guide the generation of new tests. To detect an adversarial example that demonstrates a violation of the aforementioned robustness property, the *test oracle* typically compares the output of the model on a test x', against the output obtained on a test that is both "close to" x' and known to be correctly classified by the model.

From the above description, we can identify several ingredients that define how each tool operates:

(i) the definition of coverage metrics and criteria (*e.g.,* structural, temporal, semantics), that guide the underlying test generation algorithm;
(ii) the approach for generating new tests in a way that increases coverage (*e.g.,* random mutations, symbolic analysis, gradient analysis);
(iii) the test oracle that detects violations of desired properties (*e.g.,* robustness).

In this tutorial, we will describe and illustrate the particular choices we have made for each one of these components in order to achieve coverage-guided testing for several families of ML models. We illustrate how to exercise each tool in practice, and rely on a common set of datasets to clearly distinguish the particularities of each approach.

Outline Practical aspects related to the installation and setup of the tools are given in Part 2, and datasets are introduced in Part 3. We then consider each tool in turn, in their respective part:

DeepConcolic for testing Convolutional Neural Networks (CNNs) (Part 4);

TestRNN tests Recurrent Neural Networks (RNNs), and in particular Long-Short-Term Memory Models (LSTMs) (Part 5);

EKiML specifically addresses poisoning attacks and defence for Random Forests (Part 6); and

GUAP seeks the discovery of spatial and/or additive universal adversarial attacks that fool an image classifier on a full set of images (Part 7).

2 Installation & Setup

2.1 Downloading the Software

The software is publicly released on its GitHub page:

https://github.com/TrustAI/DeepConcolic

As the first of step of installation, the following commands can be executed to download the software:

```
$ git clone https://github.com/TrustAI/DeepConcolic.git -b tutorial
$ cd DeepConcolic
```

Throughout this document, we will give details on how to actually use the tools included in this directory (*i.e.,* DeepConcolic, EKiML, TestRNN, and GUAP), and explore their respective results. We will assume that commands are executed *from within the* `DeepConcolic` *directory*.

Observe that the `git` command above retrieves the `tutorial` branch of the source code. This branch includes versions of the tools that perform basic checks to ensure the above assumption, *i.e.,* that executions take place from within the `DeepConcolic` directory.

2.2 Setting up a Software Environment and Installing Dependencies

Once the software is downloaded, we need to install an Anaconda through the link https://docs.anaconda.com/anaconda/install/. We set up an Anaconda environment called `deepconcolic` with the following commands:

```
$ conda create --name deepconcolic python==3.7
$ conda activate deepconcolic
```

We can now install the additional software dependencies within this environment by using the following commands:

```
$ conda install opencv nltk matplotlib
$ conda install -c pytorch torchvision
$ pip3 install numpy==1.19.5 scipy==1.4.1 tensorflow\>=2.4 pomegranate==0.14 \
    scikit-learn scikit-image pulp keract np_utils adversarial-robustness-toolbox \
    parse tabulate pysmt saxpy keras menpo patool z3-solver
```

Similarly to the checks about the current directory, the tools provided in the `tutorial` branch also ensure that the `conda` environment is properly setup, *i.e.,* the `conda activate deepconcolic` command has been executed.

2.3 Checking Installed Solvers

Many of the tools that we are covering in this tutorial internally rely on dedicated problem solvers that may need to be installed separately. In particular, EKiML and DeepConcolic use SMT and LP solvers, respectively.

SMT Solver for EKiML: Z3 EKiML makes use of Z3 for solving *Satisfiability Modulo Theories* (SMT) problems (this choice ensures appropriate performances of the tool, and is at the moment not configurable). To ensure that this solver is available, we can run the following command and check that the printed list of solvers includes `z3`:

```
$ python3 -m pysmt install --check
```

```
...
Solvers: z3
...
```

LP Solver for DeepConcolic Several concolic algorithms implemented in DeepConcolic make use of a *Linear Programming* (LP) problem solver to generate new test cases. We check what solver is selected by default by DeepConcolic with:

```
$ python3 -c 'from␣deepconcolic␣import␣lp;␣lp.pulp_check␣()'
```

```
Using TensorFlow version: 2.4.0
PuLP: Version 2.4.
PuLP: Available solvers: GLPK_CMD, PULP_CBC_CMD, COIN_CMD, PULP_CHOCO_CMD.
PuLP: COIN_CMD solver selected (with 10.0 minutes time limit).
```

Note the solvers listed above all work by writing LP problems into files on disk and launching a separate process (this is indicated by the "`_CMD`" suffixes). Significant performance gains can be obtained for the concolic algorithms of DeepConcolic by installing a solver that provides direct python bindings such as CPLEX[1].

2.4 Optional Sandboxing

The original purpose of most of the tools that we are covering in this tutorial is to support academic experiments with the proposed algorithms. It is therefore legitimate to assume that unanticipated uses of these tools may inadvertently have unintended effects on your systems (such as, in the worst case, delete full directory hierarchies). To prevent this from happening, we provide a means to protect your file-systems against these potentially harmful operations. On Debian-based GNU/Linux systems (*e.g.*, Ubuntu), we suggest using bubblewrap[2] via a helper script that we provide to safely experiment with the tools. The latter can be used by running the following command from within the main DeepConcolic directory.

[1] Please see https://coin-or.github.io/pulp/guides/how_to_configure_solvers.html for detailed instructions for installing and setting up additional solvers.

[2] https://github.com/containers/bubblewrap, usually available as a distribution package named "`bubblewrap`".

```
$ HISTFILE=.histfile tuto-scripts/bwrap-shell.sh /bin/bash
```

The above command starts a `bash` shell process (and any subsequently forked sub-process, launched commands) that essentially has no write access to any directory other than the current directory and `/tmp`. It additionally sets the environment variable `HISTFILE` in such a way that the history of typed commands is still made persistent after the wrapper terminates (since typically the root of the user home directory cannot be written by the newly launched `bash` interpreter). Note that the wrapper script we provide may fail if executed under a network-mounted directory hierarchy (*e.g.,* via NFS).

2.5 Testing the Installation

We can test if the software is correctly installed and ready to go by running a few commands such as

```
$ python3 -m deepconcolic.main -h
$ python3 -m testRNN.main -h
$ python3 -m EKiML.main -h
$ python3 -m GUAP.run_guap -h
```

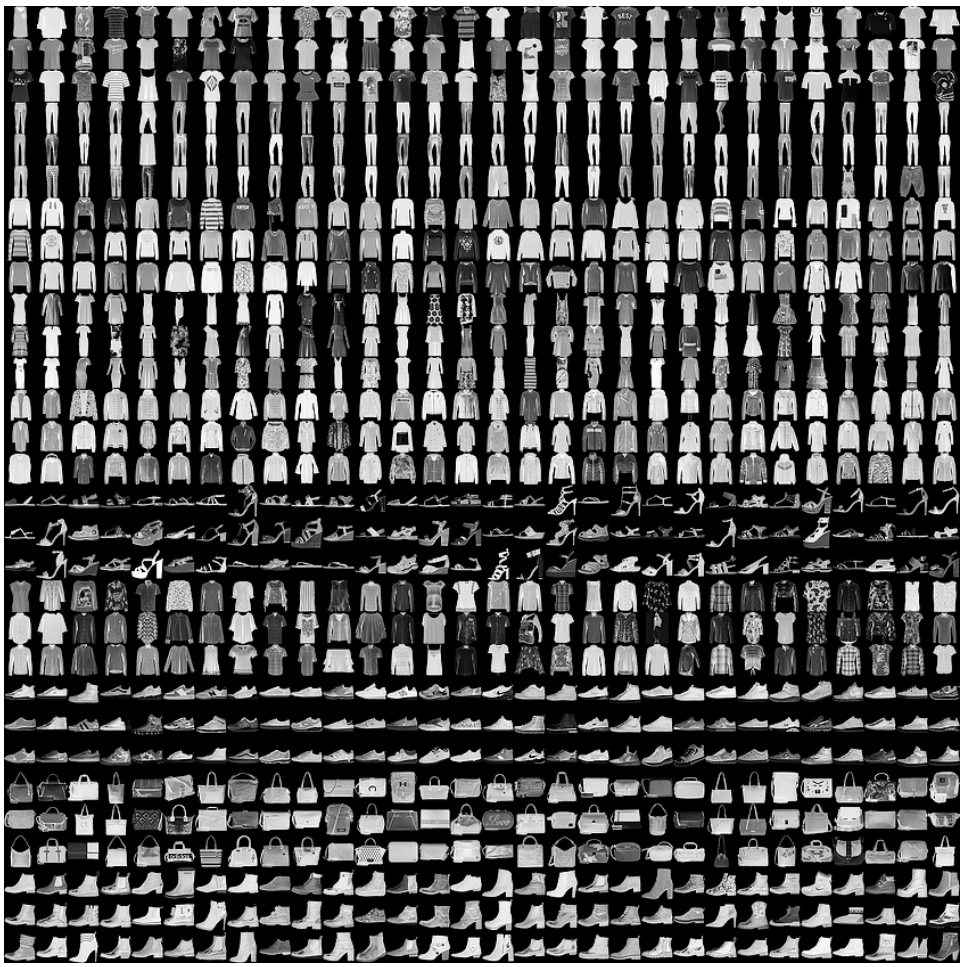
which should print usage messages for DeepConcolic, TestRNN, EKiML, and GUAP.

3 Example Datasets

We consider two datasets as running examples of this tutorial: Fashion-MNIST and UCI-HAR.

3.1 Fashion-MNIST

Fashion-MNIST [12] is a dataset of Zalando's article images, consisting of a training set of 60,000 examples and a test set of 10,000 examples. Each Fashion-MNIST example is a 28x28 grayscale image, associated with a label from 10 classes. Although the images are in the same format as in the widespread handwritten digit recognition MNIST dataset, the classification task for Fashion-MNIST is deemed more complicated.



3.2 UCI-HAR: Human Activity Recognition (Preprocessed)

UCI-HAR database (further abbreviated HAR in the following) was built by Anguita et al. [3] from the recordings of 30 subjects performing activities of daily living while carrying a waist-mounted smartphone with embedded inertial sensors. Each record in the dataset consists in a 561-feature vector with time and frequency domain variables, along with an associated activity label. For each entry, the recorded variables include:

- triaxial acceleration from the accelerometer (total acceleration) and the estimated body acceleration;
- triaxial angular velocity from the gyroscope.

The classification task at hand is to determine whether each sample corresponds to sensor readings obtained while the subject performed one of six activities: walking, walking upstairs, walking downstairs, sitting, standing, or lying down.

Unlike the other tools, EKiML does not automatically download the data. We therefore store the dataset on our server, so that it can be downloaded by running the following command:

```
$ wget -P datasets https://cgi.csc.liv.ac.uk/~acps/datasets/UCIHARDataset.csv
```

4 Testing Convolutional Neural Networks

In this Section, we will explore how concolic executions and other test case generation algorithms implemented in DeepConcolic can be used to support the design of ML-enabled safety-critical systems. We will in particular focus on the specific abilities of DeepConcolic to achieve various test coverage criteria on *feed-forward*, dense and/or convolutional neural networks. We first give a succinct overview of these capabilities and explain how to exercise them in practice. We then conduct experiments using our two running example datasets to both illustrate how DeepConcolic can be used for testing, and the effects of various selected coverage criteria.

4.1 Overview of DeepConcolic

Let us first briefly present the general working principles and purposes of DeepConcolic.

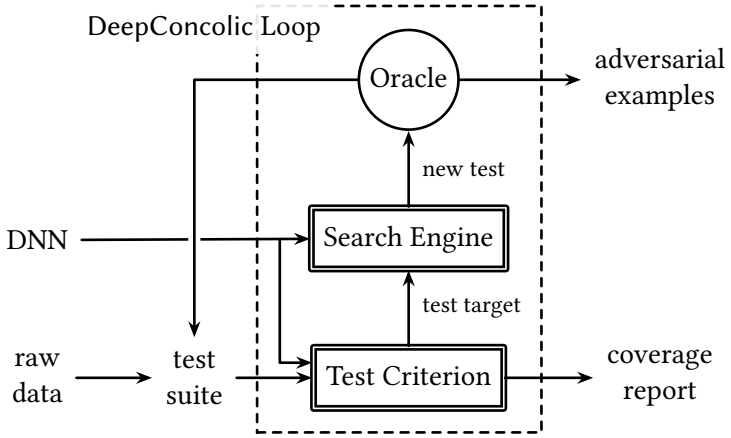
We give an overview of DeepConcolic in Figure 4.1. DeepConcolic takes as input a *feed-forward neural network* $\mathcal{N}$ and some raw data. The latter includes both a *test dataset* X associated with classification labels Y, along with a *training dataset* X_{train} (with associated labels Y_{train}) that has been used to train $\mathcal{N}$ (*i.e.,* compute its parameters).

4.1.1 Coverage-guided Testing

DeepConcolic then attempts to generate *new test inputs* according to a *concolic testing* algorithm, that it inserts into X so that the full test dataset fulfils a *predefined coverage criterion*. This process achieves coverage more efficiently than a randomised generation of new inputs by involving a *search engine* that is dedicated to a generate new inputs: the search engine takes a given *test target* t, which is selected in such a way that any new test that meets (*i.e.,* satisfies) t improves the coverage criterion over X; *i.e.,* t is not met by X. The role of the search engine is to generate a *new test input* x' that achieves t. x' is then checked against an *oracle*, that determines whether it consists in a sufficiently realistic new input, and then if it is correctly classified by $\mathcal{N}$: x' is inserted into X if it passes the oracle, or rejected otherwise. Further, any test inserted in X that is not classified correctly by $\mathcal{N}$ is an *adversarial example*. The algorithm progresses towards achieving the coverage criterion as long as the search engine is guaranteed to produce inputs that meet the test targets.

4.1.2 Main Components of DeepConcolic

From the above description, one can identify several main components that drive DeepConcolic's operations:



Test Criteria: • *Structural (neuron-based)* • *High-level (hidden feature-based)*
Search Engines (Concolic): • *Linear programming* • *Pixelwise optimisation*
Search Engines (Other): • *Gradient analysis* • *Fuzzing*

Figure 4.1: Overview of DeepConcolic's Architecture

Test criterion The test criterion embodies the objective of the testing algorithm, via the definition of the coverage criterion. Just like coverage is often defined in terms of statements or lines of code in the case of traditional software, for neural networks we consider criteria that are typically expressed in terms of the network's structure (*e.g.,* neurons, layers). Coverage criteria in DeepConcolic can be divided into the two following categories:

- *Structural* criteria typically refer to values pertaining to individual *neurons*;
- *High-level* criteria take a step towards covering the *semantic features* that have been learned by $\mathcal{N}$;

We will give more details about each criterion below.

Oracle To vet a new input as legitimate (*i.e.,* sufficiently close to a reference input), and check whether it is correctly classified by $\mathcal{N}$, the oracle relies on a given *norm* and, optionally, on a (series of) *post-filters*:

- *Norms* are distance metrics that can be used to assess the similarity of two given inputs. In DeepConcolic, the norm may be based on the Chebyshev distance L_∞, that uses the maximum absolute difference between any input feature (*i.e.,* colour of pixel, input scalar) as a measure of distance, or the L_0 "norm", that simply counts the number of input features that differ. We denote with $\|x - y\|_\infty$ the distance between two inputs x and y *w.r.t.* the L_∞ norm (and similarity for L_0).
  The oracle rejects any input x' *s.t* $\|x - x'\|_\infty > d_{\text{thr}}$, for some given threshold d_{thr}.
- *Post-filters* take a generated input and decide whether it satisfies some requirements that cannot be expressed by specifying a norm. Such filters can for instance be useful to deal with cases where some input features have particular semantics, as opposed to pixels of images that may just take their values in a fixed range and do not need to obey global well-formedness

constraints. We elaborate more on a typical use for post-filters in Section 4.1.4 below, and exemplify its use in Section 4.5;

If x' passes all the above filters, the oracle compares the classification labels assigned by $\mathcal{N}$ to x and x': if the labels differ, then x' is an adversarial example.

Search engines The search engine goes hand in hand with the selected criterion, although several engines may be available for each given criterion, depending on the underlying analysis technique, as well as the norm selected. DeepConcolic includes three distinct families of search engines:

- *Concolic engines* make use of symbolic encoding of (a subset of) the neural network. Such an engine takes both the test target t and a *candidate input* x to construct the encoding in such a way that its solution forms a new input x' that is close to x *w.r.t.* the considered norm. Most concolic engines construct their symbolic encoding as a *Linear Programme* (LP), and are therefore restricted to using a norm that is also a *linear metric* for this encoding to be possible; in DeepConcolic, only the L_∞ norm satisfies this property. Other concolic engines use ad-hoc global optimisation approaches that can be used to support non-linear norms like L_0.
- *GA-based search engines* operate differently from their concolic counterparts, as they approach the problems via *Gradient Analysis* (GA). In DeepConcolic, the latter is delegated to an attacker (*i.e.,* a generative adversarial network) based on the network $\mathcal{N}$ under consideration, that is then used to efficiently generate new inputs via gradient analysis . By virtue of the latter, these engines are more likely to generate new examples that are actual adversarial examples.
- *Naive engines* do not target specific coverage criteria, and naively generate many new inputs by randomly mutating legitimate inputs.

Some search engines only operate based on a given candidate input x in addition to the test target t: we say that these engines are *rooted* as they only *derive* new inputs—thereby constructing a hierarchy of inputs—, whereas engines that operate by scanning a whole set of reference inputs are said to perform a *global search.* All concolic engines are rooted, whereas GA-based ones are global. This notably means that, in the case of a rooted search, some *heuristics* need to be employed in the search of a suitable pair (t, x); this is done by the test criterion component. Furthermore, the size of the initial test dataset, that we shall denote X_0, also has implications on the ability of said heuristics to find suitable candidate inputs.

4.1.3 Norm Parameters for Concolic Search Engines

In addition to the threshold factor d_{thr} used by the oracle to assess the plausibility of generated inputs, a pair of parameters pertaining to the norm are available to tune the exploration of the norm ball around each candidate input x when selecting a concolic engine:

$\boxed{d_{\text{min}}}$ gives a lower bound for $\|x - x'\|_\infty$;

$\boxed{d^+_{\text{min}}}$ introduces some uniform random noise on the lower-bound to help further explore the input space.

Then, each time an LP problem is constructed, a lower bound for $\|x - x'\|_\infty$ is drawn uniformly in the range $[d_{\text{min}}, d_{\text{min}} + d^+_{\text{min}}]$: this mechanism allows users to control the minimum changes

to candidate input features that can be performed at each step of the test case generation algorithm.

4.1.4 Oracle Augmentation with Post-filters

DeepConcolic features a means to augment the capabilities of oracles by using a post-filter that relies on an estimation of *Local Outlier Factors* [5]. When enabled, this device is used in addition to the oracle mechanism described previously, to detect and filter out generated inputs that can be considered outliers *w.r.t.* the raw training data. The underlying estimator is initialised with a set $X_{\texttt{LOF}}$ of inputs using a specialised search tree that can be used to identify the k-nearest neighbours in $X_{\texttt{LOF}}$ that are the closest to any new input x *w.r.t.* some configurable distance metric. The local outlier factor of x is then computed based on the density of its k-nearest neighbours, and this factor is used to determine the plausibility of x using a predefined threshold.

Of course, the choice of the distance metric is of paramount importance for this mechanism to be relevant and act as a useful oracle. By default the LOF-based filter uses the *cosine distance* as a measure of distance between two inputs. This distance typically gives an appropriate indication of the relative similarity between two 1-dimensional data vectors, as it only considers the angle between the two normalised vectors and disregards the magnitude of each one of their respective components.

4.1.5 Additional Assumptions & Requirements

As of now, DeepConcolic can only be used for testing feed-forward classifiers; *i.e.,* neural networks that achieve regression functions are not supported. Note, however, that the overall approach adopted by DeepConcolic can often be extended to such estimators. For instance, this can be done via the specification of interval bounds on the output error, to define a discrimination criterion so as to determine incorrect regression results.

4.2 Coverage Criteria for Convolutional Neural Networks

We can now review the various coverage criteria available in DeepConcolic. Depending on the semantics of the underlying coverage metrics, they can currently be partitioned into *structural criteria* on one side, and *high-level criteria* on the other side.

4.2.1 Structural Criteria

The structural coverage criteria, and associated search engines implemented in DeepConcolic were originally designed and presented by Sun et al. [9]. These notably include *neuron coverage* (NC), which basically counts the amount of neurons activated by the test dataset—a neuron is said *activated* if the value output by its (ReLU) activation function is positive. Other structural variants that are available are inspired by the *modified condition/decision coverage* (MC/DC) used in software testing: in DeepConcolic, this category of *combinatorial coverages* includes sign-sign coverage (SSC) [7, 8], that seeks to witness every combination of neuron activations in each pair of successive layers of the network.

4.2.2 High-level Criteria

The family of high-level criteria available in DeepConcolic have been investigated by Berthier et al. [4]. They are defined based on a *Bayesian Network abstraction*, which constitutes an abstraction of all behaviours of the neural network subject to a given dataset. Every node in the BN is attached to a particular layer, and represents a component from a low-dimensional space that efficiently explains all combinations of neuron outputs for the layer when the neural network is subject to the training data X_{train}. This is achieved by means of a *linear dimensionality reduction technique* (such as Principal Component Analysis—PCA— or Independent Component Analysis—ICA), and the underlying assumption is that each extracted dimension for a hidden layer captures a high-level "*hidden feature*" that has been learned by the neural network under consideration. Each node of this BN represents a random variable that ranges over a finite set of *intervals* that partitions the hidden features, and the BN represents a joint probability distribution for all the individual combinations of intervals for the hidden features.

Bayesian inference is used to compute the probabilities in the BN based on the neuron outputs induced by the test dataset X. As a result, node values that appear with a high probability according to the BN represent behaviours that are well tested by X; conversely, node values that are rare *w.r.t.* the BN are not tested (enough) by X. Then, a test dataset satisfies a high-level BN-based criterion if no hidden feature interval is rare *w.r.t.* the BN: this defines the criterion that we call the *Bayesian-network Feature Coverage* (BFC).

The BN can alternatively be seen as a set of conditional probability tables, that we can use to define an additional criterion that accounts for the *causality assumption* underling the layered nature of the neural network, which states that the set of outputs at a particular layer depends on the set of outputs at a preceding layer. Indeed, the conditional probability tables give, for each hidden feature interval pertaining to hidden or output layers, whether some test in X simultaneously exhibits each combination of intervals at the preceding layer. This defines a criterion that we call the *Bayesian-network Feature-dependence Coverage* (BFdC).

Parameters Underlying the BN Abstraction

The main parameters that drive the construction of the BN abstraction defined above relate to the choice of a linear feature extraction technique, along with the creation of hidden feature intervals.

Regarding linear feature extraction, DeepConcolic supports both PCA and ICA. Both techniques take an Integer parameter $|\Lambda_i|$ that describes, for each layer l_i of the network, the number of components to extract from the set of neuron activations at the layer; *i.e.,* $|\Lambda_i|$ corresponds to the number of hidden features (and thus BN nodes) pertaining to l_i.

Regarding the discretization of each hidden feature, several strategies are available as well. Basic strategies include "uniform" and "quantile", that each take the desired amount of intervals for each hidden feature as a parameter. The difference between the two lies in that the former uniformly partitions the hidden feature segment that contains all points of the (projected) training dataset, whereas the latter computes interval boundaries so as to evenly spread the dataset in the resulting intervals.

4.3 Practical Aspects in using DeepConcolic

The following command prints a help message for DeepConcolic:

```
$ python3 -m deepconcolic.main -h
```

DeepConcolic always requires the specification of a dataset (given with `--dataset`), a trained neural network to test (given with `--model`, that accepts trained networks in Keras H5 format[1]), and a directory where it will place all the output files it generates (with `--outputs`). To perform testing, DeepConcolic also requires the specification of a criterion and a norm (with arguments `--criterion`, `--norm`); the search engine is then selected automatically from the two latter. The following command-line arguments may additionally be given to specify all further parameters mentioned above.

`--init` $|X_0|$ specifies the size of X_0 (1 by default);

`--max-iterations` N gives the maximum number of iterations of the algorithm;

`--norm-factor` d_{thr} specifies the norm distance above which generated inputs are rejected as non-plausible by the oracle (its default value is $d_{\text{thr}} = \frac{1}{4}$);

`--lb-hard` d_{min} and `--lb-noise` d^{+}_{min} respectively give parameters for tuning the exploration of the norm ball by LP-based concolic search engines (see Section 4.1.3). Default values are $d_{\text{min}} = \frac{1}{255}$ for image datasets ($\frac{1}{100}$ otherwise) and $d^{+}_{\text{min}} = \frac{1}{10}$;

`--filters LOF` enables oracle augmentation using the LOF-based post-filter with cosine distance (see Section 4.1.4).

Selecting Covered Layers It is advisable to manually specify a list of layers to be considered by the coverage criterion. This can be specified as a list of layer names given as argument to `--layers`, *e.g.*, `--layers activation_1 activation_2`. For technical reasons, structural criteria require that the above list of layers only contain layers that integrate a ReLU activation as part of their functional behaviour. Any supported layer can be specified when targeting a high-coverage criterion: *i.e.*, dense, convolutional, max-pooling, ReLU activation—or even dropout, flatten or reshape layers.

Miscellaneous DeepConcolic accepts several additional useful flags:

`--save-all-tests` Dump every generated test (images, etc); without this flag the tool only saves adversarial examples;

`--setup-only` Instruct the tool to terminate right before starting the main DeepConcolic loop: use this to check whether command-line arguments are appropriate to the considered models;

`--rng-seed` Specifies a seed for the internal random number generator: this can be used to obtain reproducible executions. Note, though, that this holds up to the non-determinism induced by underlying tools used by search engines, such as LP solvers.

[1] See https://www.tensorflow.org/guide/keras/save_and_serialize#keras_h5_format

Parameters & Companion Tool for BN-based Criteria

The specification of parameters for constructing the BN abstraction for high-level coverage described in Section 4.2.2, can be done via a YAML file given as argument to `--dbnc-spec`. We refer to the file `dbnc/example.yaml` in the source code repository for a detailed and illustrated description of each parameter. For the purpose of simplifying this tutorial, however, we will make use of a companion tool called `dbnabstr` provided as part of DeepConcolic. This tool can notably be used to pre-compute BN abstractions and dump them into a file. The latter can directly be given to DeepConcolic for defining BN-based criteria using the command-line option `--bn-abstr`.

The creation of an abstraction is done via

```
$ python3 -m deepconcolic.dbnabstr --dataset <Dataset> --model <Model> create \
    <abstraction.pkl> ...
```

where `--dataset` and `--model` give the dataset and model in a similar way as for DeepConcolic, and `<abstraction.pkl>` is a `.pkl` file where the abstraction is to be saved. The main command-line parameters that drive the abstraction are:

`--layers` As for the main DeepConcolic tool, this gives the DNN layers to consider for measuring coverage;

`--feature-extraction` In {`pca`, `ipca`, `ica`} (the default being `pca`), this parameter specifies the dimensionality reduction technique to use for linear feature extraction (`ipca` is a memory-efficient, incremental version of `pca`);

`--train-size` How many samples of the training dataset to use for feature extraction (the default is all;

`--num-features` Number of extracted features for each layer considered (default is 2);

`--num-intervals` Number of intervals for partitioning for each extracted feature (default is 2);

`--extended-discr` Whether to compute intervals that partition the full domain of each hidden feature (*i.e.,* $\mathbb{R}$) instead of their respective range of values observed during training.

We will now illustrate how to use DeepConcolic to support the validation of our neural networks. We will particularly focus on the intents and effects of the testing algorithms when applied using various norms and criteria.

4.4 Example: Fashion-MNIST

We start with the Fashion-MNIST image classification task for it provides results that can easily be visually inspected. They show the specificity of coverage criteria, and give hints about the behaviours of DeepConcolic's coverage-guided testing algorithms.

4.4.1 Training or Downloading Models

For ease of use and experimentation, DeepConcolic's source code includes a script that enables one to construct and train two CNNs for this dataset. It saves models under `/tmp`, and can be executed from within the repository's root directory via the following command (note this script automatically downloads the dataset from the Internet):

Layer	Name & Function specification	Output shape	#parameters
l_0	`conv2d` (convolutional)	$26 \times 26 \times 32$	320
l_1	`activation` (ReLU)	$26 \times 26 \times 32$	0
l_2	`max_pooling2d` (max-pooling)	$13 \times 13 \times 32$	0
l_3	`conv2d_1` (convolutional)	$9 \times 9 \times 64$	51,264
l_4	`activation_1` (ReLU)	$9 \times 9 \times 64$	0
l_5	`max_pooling2d_1` (max-pooling)	$4 \times 4 \times 64$	0
l_6	`flatten` (flat)	1024	0
l_7	`dense` (dense)	100	102,500
l_8	`activation_2` (ReLU)	100	0
l_9	`dense_1` (dense)	10	1,010
l_{10}	`activation_3` (softmax)	10	0

Table 4.1: Layers of the medium-sized CNN model `fashion_mnist_medium.h5` for Fashion-MNIST

```
$ python3 -m deepconcolic.gen_fashion_mnist
```

Generated models can then be copied from `/tmp` into the `saved_models` directory (to be created, if necessary). Alternatively, one can download pre-trained models from our server:

```
$ wget -P saved_models https://cgi.csc.liv.ac.uk/~acps/models/fashion_mnist_medium.h5
$ wget -P saved_models https://cgi.csc.liv.ac.uk/~acps/models/fashion_mnist_large.h5
```

The commands above download two pre-trained models into the directory "`saved_models/`", where the model "`fashion_mnist_medium.h5`" is a medium-sized model whose layers are listed in Table 4.1, and "`fashion_mnist_large.h5`" provides a slightly larger model with more pairs of convolutional/max-pooling and a dropout layer for further experimental investigations.

4.4.2 Achieving Structural Criteria

Structural coverage criteria such as NC or SSC focus on the patterns that appear in the outputs of ReLU activation functions. Among the layers of `fashion_mnist_medium.h5` (*cf.* Table 4.1), this corresponds to the outputs of `activation`, `activation_1`, and `activation_2`.

Neuron Coverage with Pixelwise Optimisation (L_0 Norm) As we are dealing with an image dataset, let us first exercise the search engine that operates via pixelwise optimisation. This essentially amounts to an exploration of L_0 norm balls, in search for a new image that satisfies the test target and that differs from the candidate image in as few pixels (or colour channels) as possible.

In this case, we explicitly select ReLU activation layers (as arguments to the `--layers` flag), ask to save every generated input image into the output directory `outs/fm-medium/nc-l0` by giving the `--save-all-tests` flag, and restrict ourselves to 100 iterations so as to obtain reasonable running times for the whole command (under a minute on a 3,4GHz Quad Core Intel i7 CPU with 16GB of memory):

```
$ python3 -m deepconcolic.main --outputs outs/fm-medium/nc-l0 --dataset fashion_mnist \
    --model saved_models/fashion_mnist_medium.h5 --layers activation activation_1 \
    activation_2 --criterion nc --norm l0 --save-all-tests --max-iterations 100
```

```
...
Randomly selecting an input from test data.
```

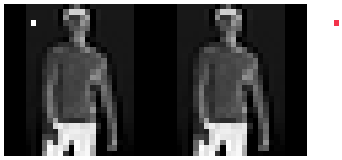
When given the above arguments, DeepConcolic first constructs an initial singleton test set X_0 by randomly selecting an input that is correctly classified by the model. This seed input is taken from the test data X (provided as part of the Fashion-MNIST dataset). Then the initial neuron coverage is shown:

```
...
#0 NC: 36.49130629%
```

and the search engine subsequently alters the seed and newly generated inputs in order to increase the overall amount of neurons activated by the set of test cases under construction:

```
| Targeting activation of (1, 4, 27) in conv2d
#1 NC: 36.95942934% with new test case at L0 distance 1: passed
| Targeting activation of (1, 19, 27) in conv2d
#2 NC: 37.40897607% with new test case at L0 distance 1: passed
...
| Targeting activation of (23, 4, 12) in conv2d
#63 NC: 50.53499777% with new test case at L0 distance 1: adversarial
| Targeting activation of (15, 21, 7) in conv2d
#64 NC: 50.53499777% after failed attempt
...
| Targeting activation of (19, 11, 15) in conv2d
#100 NC: 56.94010997% with new test case at L0 distance 1: passed
Terminating after 100 iterations: 92 tests generated, 4 of which are adversarial.
```

The last line output to the console shows a short summary of the execution. Every generated test image is saved within the directory `outs/fm-medium/nc-l0`. In the case of the command at hand, most of them consist in versions of the seed image where the brightness of individual pixels have been altered. As an illustration, we show below a generated input on the left, the original candidate input in the middle, and the diff on the right; in the latter a red pixel indicates a change of value from the original to the generated input.



Neuron Coverage with L_∞ Norm and a Concolic Search Engine Turning to a concolic search engine, we can employ a similar command line and switch to a norm that allows more global perturbations to input images.

```
$ python3 -m deepconcolic.main --outputs outs/fm-medium/nc-linf --dataset \
    fashion_mnist --model saved_models/fashion_mnist_medium.h5 --layers activation \
    --criterion nc --norm linf --save-all-tests --max-iterations 10
```

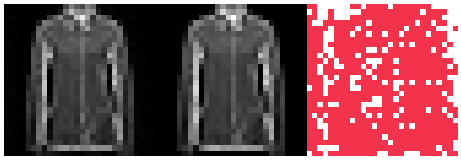
Note that we are only considering the first activation layer (`activation`). Indeed, the number of linear variables and constraints grows linearly with the depths of considered layers. Thus, restricting ourselves to increase neuron coverage at the shallowest layer allows us to obtain reasonable overall execution times with the LP solver that ships with `pulp`. Other activation

layers of this model can additionally be considered when a more advanced LP solver is used, like CPLEX or GUROBI (see Section 2.3).

Moreover, the computational complexity of the search increases significantly *w.r.t.* the pixel-wise optimisation above: we therefore restrict ourselves to observing the first 10 iterations of DeepConcolic so the command terminates within reasonable time (less than 15 minutes).

```
...
Randomly selecting an input from test data.
#0 NC: 37.49075444%
| Targeting activation of (8, 11, 22) in conv2d
#1 NC: 37.50924556% with new test case at Linf distance 0.015686304195254464: passed
| Targeting activation of (7, 7, 29) in conv2d
#2 NC: 37.55085059% with new test case at Linf distance 0.04705884900747559: passed
...
#10 NC: 37.83746302% with new test case at Linf distance 0.09019610390943639: passed
Terminating after 10 iterations: 10 tests generated, 0 of which is adversarial.
```

The result of this command is similar to the one above. In this case, however, the reported distances indicate the maximal change of value of any pixel instead of the number of changed pixels.

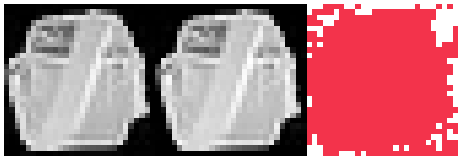


Sign-Sign Coverage with Concolic Search Engine Other coverage metrics and associated search engines can also be used, such as the MC/DC-style criteria. Note the layers given to `--layers` are *decision* layers only, therefore we omit the first layer (`activation`) below, which can only be considered as encoding a set of conditions in MC/DC criteria. We omit the deepest layer (`activation_2`) as well for the same reason as above. In this case, however, the computational complexity of the symbolic search further increases *w.r.t.* the basic neuron coverage considered above, with typical LP problems of more than 100,000 constraints for the command below. Still, a reasonable termination time of less than 15 minutes can be obtained when using advanced LP solvers (CPLEX in this instance).

```
$ python3 -m deepconcolic.main --outputs outs/fm-medium/ssclp-linf --dataset \
    fashion_mnist --model saved_models/fashion_mnist_medium.h5 --layers activation_1 \
    --criterion ssclp --norm linf --init 100 --save-all-tests --max-iterations 10
```

```
...
Initializing with 100 randomly selected test cases that are correctly classified.
#0 SSC: 0.00000000%
| Targeting decision 82 in dense, subject to condition (5, 5, 26) in conv2d_1
#1 SSC: 0.00086207% with new test case at Linf distance 0.031372578471314694: passed
| Targeting decision (2, 4, 0) in conv2d_1, subject to condition (0, 0, 0) in activation
#2 SSC: 0.00086207% after failed attempt
...
#10 SSC: 0.00344828% with new test case at Linf distance 0.058823529411764774: passed
Terminating after 10 iterations: 3 tests generated, 0 of which is adversarial.
```

The patterns of changed pixels for generated tests often exhibit similar structures as the ones obtained above for neuron coverage:

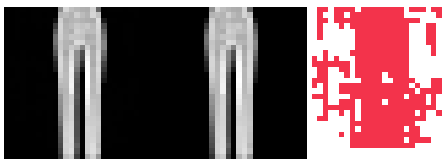


We additionally observe that the reported coverage increases very slowly with each successful iteration. This stems from the definition of sign-sign coverage that induces a combinatorial enumeration of all possible sets of condition patterns for each decision neuron. Also note the reported coverage starts at 0% even though 100 test cases have been used to construct X_0; this stems from the combinatorial nature of the coverage criterion, for which an accurate accounting based on all (unordered) elements of X_0 would either be too computationally expensive, or lack a precise meaning.

Sign-Sign Coverage with GA-based Search Engine The GA-based engine for sign-sign coverage (enabled with `--criterion ssc`) relaxes the aforementioned limitation by allowing users to specify a ratio of condition neurons whose activations need to be altered independently (instead of always a single one for the strict criterion as above); this ratio is specified with `--mcdc-cond-ratio`. The command below targets the same sign-sign coverage criterion as above by using such an engine. Due to the relaxation in the definition of the criterion, this generation approach reports higher coverage measures and is usually more efficient than its concolic counterpart (it takes about 12 minutes to terminate).

```
$ python3 -m deepconcolic.main --outputs outs/fm-medium/ssc-linf --dataset \
    fashion_mnist --model saved_models/fashion_mnist_medium.h5 --layers activation_1 \
    activation_2 --criterion ssc --norm linf --init 100 --mcdc-cond-ratio 0.1 \
    --save-all-tests --max-iterations 10
```

```
...
Initializing with 100 randomly selected test cases that are correctly classified.
#0 SSC: 0.00000000%
| Targeting decision (1, 4, 43) in conv2d_1, subject to any condition
#1 SSC: 0.01077586% with new test case at Linf distance 0.007843165304146527: passed
| Targeting decision 80 in dense, subject to any condition
#2 SSC: 0.45215517% with new test case at Linf distance 0.007843166706608784: passed
...
#10 SSC: 1.36724138% with new test case at Linf distance 0.007843166940352475: passed
Terminating after 10 iterations: 7 tests generated, 0 of which is adversarial.
```



4.4.3 Achieving High-level Criteria

Let us now turn to the higher-level criteria available in DeepConcolic. As mentioned in Section 4.2.2, such criteria are defined based on a combination of a dimensionality reduction technique, and the construction of a Bayesian Network that allows one to capture how a given test dataset exercises high-level features that have been learned by hidden layers of the DNN.

In principle the high-level (BN-based) criteria can be used to investigate how a test dataset exercises a set of hidden features that has been learned from the training dataset and internally represented by *any layer* of the CNNs. For the purposes of this tutorial, however, we will restrict ourselves to hidden features learned by `fashion_mnist_medium.h5` at layers `activation`, `activation_1`, and `activation_2` as well.

Extraction of Hidden Features via Dimensionality Reduction The following command performs dimensionality reduction on the neuron values of the activation layers based on 10,000 samples of training data (that ships with the Fashion-MNIST dataset). For this example, we have opted to concentrate on the 3 principal components for each layer, that are partitioned into 5 intervals each to obtain a BN of 9 nodes and 18 edges. The tool constructs the structure of the BN abstraction (this should take less than a minute), and saves it into `outs/fm-medium/bn.pkl`.

```
$ python3 -m deepconcolic.dbnabstr --dataset fashion_mnist --model \
    saved_models/fashion_mnist_medium.h5 create outs/fm-medium/bn.pkl \
    --feature-extraction pca --num-features 3 --num-intervals 3 --extended-discr \
    --layers activation activation_1 activation_2 --train-size 10000
```

```
...
Using extended 3-bin discretizer with uniform strategy for layer activation
Using extended 3-bin discretizer with uniform strategy for layer activation_1
Using extended 3-bin discretizer with uniform strategy for layer activation_2

| Given 10000 classified training sample
| Extracting and discretizing features for layer activation...
| Extracted 3 features
| Discretization of feature 0 involves 5 intervals
| Discretization of feature 1 involves 5 intervals
| Discretization of feature 2 involves 5 intervals
| Discretized 3 features
| Extracting and discretizing features for layer activation_1...
| Extracted 3 features
| Discretization of feature 0 involves 5 intervals
| Discretization of feature 1 involves 5 intervals
| Discretization of feature 2 involves 5 intervals
| Discretized 3 features
| Extracting and discretizing features for layer activation_2...
| Extracted 3 features
| Discretization of feature 0 involves 5 intervals
| Discretization of feature 1 involves 5 intervals
| Discretization of feature 2 involves 5 intervals
| Discretized 3 features
| Captured variance ratio for layer activation is 16.60%
| Captured variance ratio for layer activation_1 is 18.60%
| Captured variance ratio for layer activation_2 is 46.70%
| Created Bayesian Network of 9 nodes and 18 edges.
Dumping abstraction into `outs/fm-medium/bn.pkl'... done
```

When using PCA as a feature extraction technique, the output to the console reports, for each layer, the total amount of variance in its neuron values that is captured by the extracted principal components. Here, we can observe that using 3 features for the first 2 layers captures less than 20% of their respective variance: this basically means that many changes of neuron values for these layers are not reflected in significant-enough changes to any of the 3 first principal components. Further tuning of the parameters above could be employed to improve these figures. Yet, for the sole purposes of test case generation, even a small amount of captured

variance in hidden layers is enough to obtain new inputs that exhibit learned features. We will therefore reuse the computed abstraction below.

The BN abstraction as computed above allows us to define high-level coverage criteria that give an account on how extracted hidden features are exercised by a given test set. To define such coverage metrics and criteria, the idea is to identify ranges of values obtained for hidden features that are deemed not exercised enough: this is done internally via a partitioning of the hidden features into a set of intervals, and the marginal probabilities in the BN give a measure of the amount of test cases that exercises each hidden feature interval.

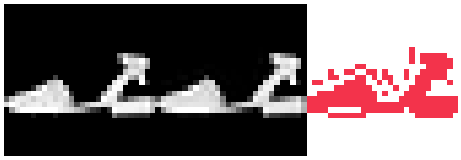
Bayesian-network Feature Coverage Criterion The first coverage metric that we define based on the BN simply reports the ratio of intervals that are exercised. To increase this coverage, the concolic search engine tries to synthesise new inputs that expand the range of values obtained for a hidden feature that is deemed not exercised enough. The following command defines a feature coverage criterion based on the abstraction computed above, and then performs 10 iterations of concolic test case generation based on an initial set of 100 test cases (it should terminate within 10 minutes).

```
$ python3 -m deepconcolic.main --outputs outs/fm-medium/bfc-linf --dataset \
    fashion_mnist --model saved_models/fashion_mnist_medium.h5 --bn-abstr \
    outs/fm-medium/bn.pkl --criterion bfc --norm linf --save-all-tests --init 100 \
    --max-iterations 10
```

```
...
Initializing with 100 randomly selected test cases that are correctly classified.
#0 BFC: 86.66666667%
| Targeting interval (-inf, -66.3] of feature 0 in layer activation (from test 49)
#1 BFC: 86.66666667% with new test case at Linf distance 0.047058849942450465: passed
| Targeting interval (-inf, -66.3] of feature 0 in layer activation (from test 100)
#2 BFC: 86.66666667% with new test case at Linf distance 0.07058823529411773: passed
...
| Targeting interval (-inf, -66.3] of feature 0 in layer activation (from test 56)
#9 BFC: 86.66666667% after failed attempt
| Targeting interval (-inf, -66.3] of feature 0 in layer activation (from test 7)
#10 BFC: 86.66666667% after failed attempt
Terminating after 10 iterations: 5 tests generated, 0 of which is adversarial.
```

First of all, observe that the reported coverage does not always increase with each successful generation of a test case. The reason behind this behaviour of DeepConcolic is twofold: this is in part due to the discrete nature of our definitions for high-level coverage metrics, that only count some amount of test cases that fall within given hidden feature intervals. On the other hand, both the linear dimensionality reduction technique and the LP encoding performed by the search engine introduce over-approximations that lead to inputs not meeting their intended test targets.

Still, by examining the changes of pixel colours, we observe that generated inputs tend to exhibit alterations that show how some semantics of the original input is captured and exercised to produce new images.



Further, we find empirically that these new inputs tend to be closer to reaching their test target than their respective candidate inputs. DeepConcolic therefore retains them as part of the set of legitimate generated test cases.

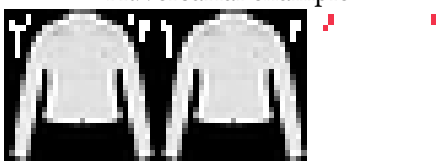
Bayesian-network Feature Coverage Criterion with Pixelwise Optimisation DeepConcolic allows one to exercise the pixelwise optimisation search engine for increasing high-level coverage. We can thus combine the BFC target as above with L_0 norm exploration:

```
$ python3 -m deepconcolic.main --outputs outs/fm-medium/bfc-l0 --dataset fashion_mnist \
    --model saved_models/fashion_mnist_medium.h5 --bn-abstr outs/fm-medium/bn.pkl \
    --criterion bfc --norm l0 --save-all-tests --init 100 --max-iterations 10
```

```
...
Starting tests for criterion BFC with norm L0 (10 max iterations).
Reporting into: outs/fm-medium/bfc-l0/BFC_L0_report.txt
Initializing with 100 randomly selected test cases that are correctly classified.
#0 BFC: 86.66666667%
| Targeting interval (-inf, -66.3) of feature 0 in layer activation (from test 83)
#1 BFC: 88.88888889% with new test case at L0 distance 5: passed
| Targeting interval (-inf, -54.5) of feature 2 in layer activation (from test 45)
#2 BFC: 91.11111111% with new test case at L0 distance 5: passed
| Targeting interval [106, inf) of feature 1 in layer activation (from test 31)
#3 BFC: 91.11111111% after failed attempt
| Targeting interval [106, inf) of feature 1 in layer activation (from test 30)
#4 BFC: 91.11111111% after failed attempt
| Targeting interval [106, inf) of feature 1 in layer activation (from test 65)
#5 BFC: 91.11111111% after failed attempt
| Targeting interval (-inf, -89.6) of feature 1 in layer activation (from test 8)
#6 BFC: 93.33333333% with new test case at L0 distance 14: passed
| Targeting interval [123, inf) of feature 2 in layer activation (from test 102)
#7 BFC: 95.55555556% with new test case at L0 distance 5: adversarial
| Targeting interval [106, inf) of feature 0 in layer activation (from test 32)
#8 BFC: 95.55555556% after failed attempt
| Targeting interval [106, inf) of feature 1 in layer activation (from test 35)
#9 BFC: 95.55555556% after failed attempt
| Targeting interval [106, inf) of feature 0 in layer activation (from test 3)
#10 BFC: 95.55555556% after failed attempt
Terminating after 10 iterations: 4 tests generated, 1 of which is adversarial.
```

Correctly classified inputs:

Adversarial example:

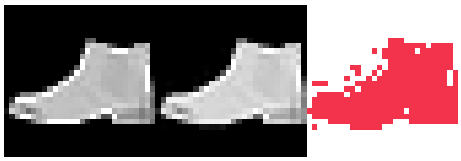


Bayesian-network Feature-dependence Coverage Criterion The feature-dependence coverage criterion is the BN-based counterpart of MC/DC-style structural criteria like sign-sign above, as it only targets combinations of hidden feature intervals in successive layers that are not exercised in the test dataset so far. This command should terminate within 1 hour when an advanced LP solver with appropriate python bindings is used (*e.g.*, CPLEX—*cf.* Section 2.3).

```
$ python3 -m deepconcolic.main --outputs outs/fm-medium/bfdc-linf --dataset \
    fashion_mnist --model saved_models/fashion_mnist_medium.h5 --bn-abstr \
    outs/fm-medium/bn.pkl --criterion bfdc --norm linf --save-all-tests --init 100 \
    --max-iterations 10
```

```
...
Initializing with 100 randomly selected test cases that are correctly classified.
#0 BFdC: 81.39733333%
| Targeting interval [-41.5, -10.1] of feature 2 in layer activation_1, subject to feature intervals (1,
↪  1, 1) in layer activation (from test 55)
#1 BFdC: 81.39733333% after failed attempt
...
#4 BFdC: 81.39733333% after failed attempt
| Targeting interval [-5.23, 25.5] of feature 1 in layer activation_1, subject to feature intervals (1, 3,
↪  2) in layer activation (from test 1)
#5 BFdC: 86.20444444% with new test case at Linf distance 0.08235297086192112: passed
...
#10 BFdC: 86.66666667% after failed attempt
Terminating after 10 iterations: 2 tests generated, 0 of which is adversarial.
```

Regarding generated inputs, we observe that focusing on a combinatorial coverage does not fundamentally change the overall nature of generated images; this behaviour is in line with what we observed in the case of structural coverage above.



4.4.4 Naive Search Engine: Fuzzing

DeepConcolic supports an experimental fuzzing engine, that can be used with the command-line flag `--fuzzing`. Fuzzing follows a simple approach to testing, by randomly mutating well-formed inputs and then running the neural network with those mutated inputs in the hope of triggering adversarial examples. Although naive, this approach has been remarkably efficient in finding bugs in programs that had never been fuzzed. While the ultimate goal is to interleave the fuzzing engine and the symbolic engine in DeepConcolic for more effectively exploring the vulnerabilities in neural networks, this is not implemented yet.

Still, the engine can be run via similar command lines. For instance, the following command will fuzz the Fashion-MNIST model `fashion_mnist_large.h5` with the seed input images contained in the directory specified by means of the argument to `--inputs`.

```
$ python3 -m deepconcolic.main --fuzzing --model saved_models/fashion_mnist_large.h5 \
    --num-processes 2 --inputs data/mnist-seeds/ --outputs outs/fm-large/fuzzing \
    --input-rows 28 --input-cols 28 --num-tests 10
```

This should result in: (i) one `mutants` folder; (i) one `advs` folder for adversarial examples; and (i) a file `adv.list` that lists the commands for validating the adversarial examples, and can be used to retrieve a list of all adversarial examples found. The `--num-processes 2` means that a parallelism of two processes will be used to perform the fuzzing. By default, fuzzing will terminate after 1,000 iterations—this can be configured by using the `--num-tests` command-line argument. A configuration `--num-tests 1000` should be interpreted in the way that $n \times 1000$ test cases would be generated if a number of n threads are used in the fuzzing. Note the example above runs for about three hours (on an Intel Core i7-7820HQ, 16GB Memory); of course, this execution time varies linearly with the argument given to `--num-tests`. Also note that the fuzzing engine has only ever been tested on an Ubuntu system, and it is very likely that it may not run properly on non-GNU/Linux-based systems like Windows.

4.5 Example: Human Activity Recognition

Let us now turn to the HAR classification task presented in Section 3.2.

4.5.1 Secific Challenges

As far as DeepConcolic is concerned, the additional challenge in dealing with the HAR dataset stems from the need to rely on additional intrinsic constraints on input features to rule out implausible inputs. For this, we make use of a post-filter. Indeed, determining a suitable distance measure to compare inputs and determine the plausibility of generated ones depends on the semantics of each individual feature. Such intrinsic constraints do not arise in the case of images, since one can consider that any combination of colour for every pixel constitutes a valid image.

Investigating Generated Inputs Since the inputs do not directly consist in images that can easily be inspected visually, we use simple plots as in Figure 4.2 to represent the distribution

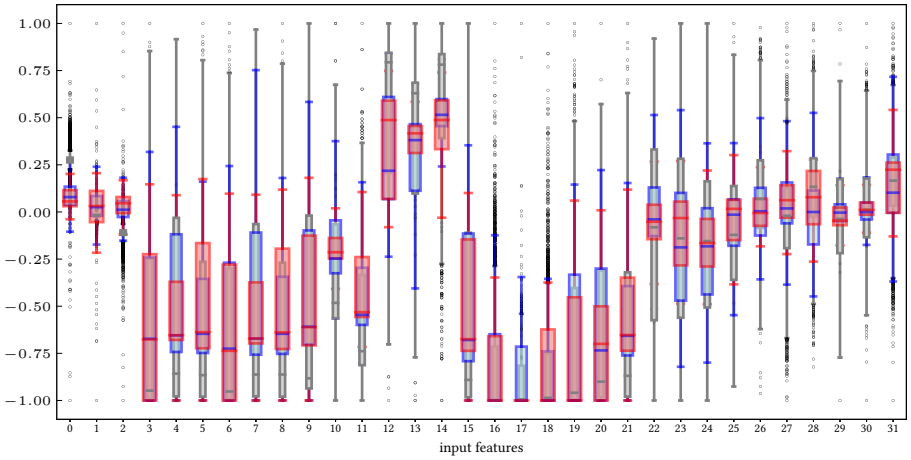


Figure 4.2: Distribution of 32 input feature values for the HAR dataset (out of 561): grey bars represent the distribution of all values from the the training dataset, blue bars show the same distributions for an example set of 1,714 inputs generated by DeepConcolic, that are correctly classified by `har_dense.h5`; similarly, red bars show the values for 54 generated inputs that are incorrectly classified (*i.e.,* adversarial examples).

of values for a subset of all input features. We generate these plots by using a HAR-specific `utils.harviz` companion script included in DeepConcolic's source code:

```
$ python3 -m utils.harviz -h
```

Expert designers with additional knowledge about the kind of sensor input data at hand would surely compute more advanced evaluation procedures for assessing whether the generated inputs satisfy the aforementioned plausibility criterion.

4.5.2 Training or Downloading Models

Similarly to the case of Fashion-MNIST, DeepConcolic's source code includes a script that enables one to construct and train a dense DNN for this dataset. It saves models under `/tmp`, and can be executed from within the the repository's root directory via the following command (this script automatically retrieves the dataset from the Internet):

```
$ python3 -m deepconcolic.gen_har
```

Generated models can then be copied from `/tmp` into the `saved_models` directory (to be created, if necessary). Our server also hosts a pre-trained model for this dataset, whose architecture is described in Table 4.2:

```
$ wget -P saved_models https://cgi.csc.liv.ac.uk/~acps/models/har_dense.h5
```

Layer	Name & Function specification	Output shape	#parameters
l_0	`dense` (dense + ReLU)	192	107,904
l_1	`dense_1` (dense + ReLU)	128	24,704
l_2	`dropout` (dropout)	128	0
l_3	`dense_2` (dense + ReLU)	92	11,868
l_4	`dense_3` (dense + ReLU)	64	5,952
l_5	`dense_4` (dense)	6	390
l_6	`activation` (softmax)	6	0

Table 4.2: Layers of the dense DNN model `har_dense.h5` for the HAR dataset

4.5.3 Achieving Structural Criteria

In principle, DeepConcolic's structural criteria and associated search engines may also be used for testing the model `har_dense.h5` that we have designed and trained for the HAR classification task.

For instance, with the command below DeepConcolic selects the "pixelwise" optimisation search engine, that attempts to find new inputs that increase neuron coverage by altering as few input feature values as possible from the given candidate inputs.

```
$ python3 -m deepconcolic.main --outputs outs/har-dense/nc-l0 --dataset OpenML:har \
    --model saved_models/har_dense.h5 --layers dense dense_{1,2,3} activation \
    --criterion nc --norm l0 --save-all-tests --max-iterations 100
```

```
...
Starting tests for criterion NC with norm L0 (100 max iterations).
Reporting into: outs/har-dense/nc-l0/NC_L0_report.txt
Randomly selecting an input from test data.
#0 NC: 99.37759336%
| Targeting activation of 2 in dense_4
#1 NC: 99.58506224% with new test case at L0 distance 4: passed
| Targeting activation of 3 in dense_4
#2 NC: 99.79253112% with new test case at L0 distance 10: passed
| Targeting activation of 5 in dense_4
#3 NC: 100.00000000% with new test case at L0 distance 1: passed
Terminating after 3 iterations: 3 tests generated, 0 of which is adversarial.
```

A concolic search engine is also selected when exploring the L_∞ norm ball:

```
$ python3 -m deepconcolic.main --outputs outs/har-dense/nc-linf --dataset OpenML:har \
    --model saved_models/har_dense.h5 --layers dense dense_{1,2,3} activation \
    --criterion nc --norm linf --save-all-tests --max-iterations 100
```

```
...
Starting tests for criterion NC with norm Linf (100 max iterations).
Reporting into: outs/har-dense/nc-linf/NC_Linf_report.txt
Randomly selecting an input from test data.
#0 NC: 99.37759336%
| Targeting activation of 2 in dense_4
Infeasible
#1 NC: 99.37759336% after failed attempt
| Targeting activation of 3 in dense_4
Infeasible
#2 NC: 99.37759336% after failed attempt
| Targeting activation of 5 in dense_4
Infeasible
...
```

```
#99 NC: 99.37759336% after failed attempt
| Targeting activation of 0 in dense_4
Infeasible
#100 NC: 99.37759336% after failed attempt
Terminating after 100 iterations: 0 test generated, 0 of which is adversarial.
```

In both cases above, however, we observe that a structural coverage that only takes activation patterns into account is not appropriate to test `har_dense.h5`, as even a single input taken randomly is enough to achieve near-100% neuron coverage.

In fact, this stems from a technicality in the architecture of `har_dense.h5`, where all considered layers integrate ReLU activation functions (except `dense_4`, that does not integrate an activation function as it is directly followed by a softmax layer—see Table 4.2). Since this eliminates every negative value most layers, every activation pattern observed by DeepConcolic includes activated neurons only.

Furthermore, the LP-based concolic search engine is unable to derive a concrete input that triggers the activation of neuron 0 in layer `dense_4` from the only seed input in X_0. Indeed, we can actually see from the trace above that every generated LP problem is infeasible. This means that no test exists that triggers the sought-after activation within the L_∞ norm ball around the seed input (and that meets the input bound constraints induced by $d_{\min}$ and $d^+_{\min}$)

4.5.4 Achieving High-level Criteria

Let us first use DeepConcolic's companion tool dedicated to the creation of BN-based abstractions, and focus on the output of every activation layer. The computation of the abstraction is done with the `create` sub-command of this tool:

```
$ python3 -m deepconcolic.dbnabstr --dataset OpenML:har --model \
    saved_models/har_dense.h5 create outs/har-dense/bn.pkl --feature-extraction pca \
    --num-features 3 --num-intervals 3 --extended-discr --layers dense dense_{1,2,3} \
    activation
```

```
...
| Given 7724 classified training sample
| Extracting features for layer dense...
| Extracted 3 features
| Discretizing features for layer dense...
| Discretization of feature 0 involves 3 intervals
| Discretization of feature 1 involves 3 intervals
| Discretization of feature 2 involves 3 intervals
| Discretized 3 features
| Extracting features for layer dense_1...
| Extracted 3 features
| Discretizing features for layer dense_1...
| Discretization of feature 0 involves 3 intervals
| Discretization of feature 1 involves 3 intervals
| Discretization of feature 2 involves 3 intervals
| Discretized 3 features
| Extracting features for layer dense_2...
| Extracted 3 features
| Discretizing features for layer dense_2...
| Discretization of feature 0 involves 3 intervals
| Discretization of feature 1 involves 3 intervals
| Discretization of feature 2 involves 3 intervals
| Discretized 3 features
| Extracting features for layer dense_3...
```

```
| Extracted 3 features
| Discretizing features for layer dense_3...
| Discretization of feature 0 involves 3 intervals
| Discretization of feature 1 involves 3 intervals
| Discretization of feature 2 involves 3 intervals
| Discretized 3 features
| Extracting features for layer activation...
| Extracted 3 features
| Discretizing features for layer activation...
| Discretization of feature 0 involves 3 intervals
| Discretization of feature 1 involves 3 intervals
| Discretization of feature 2 involves 3 intervals
| Discretized 3 features
| Captured variance ratio for layer dense is 64.74%
| Captured variance ratio for layer dense_1 is 66.94%
| Captured variance ratio for layer dense_2 is 75.86%
| Captured variance ratio for layer dense_3 is 76.04%
| Captured variance ratio for layer activation is 61.03%
| Created Bayesian Network of 15 nodes and 36 edges.
Dumping abstraction into `outs/har-dense/bn.pkl'... done
```

Regarding feature extraction, we can first observe that a substantial amount of all variance in the data can be captured with three hidden features at each layer considered. The sub-command `show` enables us to inspect the hidden feature intervals for each layer and extracted feature:

```
$ python3 -m deepconcolic.dbnabstr --dataset OpenML:har --model \
    saved_models/har_dense.h5 show outs/har-dense/bn.pkl
```

```
...
Loading abstraction from `outs/har-dense/bn.pkl'... done
===  Extracted Features and Associated Intervals  ==============================
Layer       Feature    Intervals
----------  ---------  ----------------------------------------
dense       0          (-inf, -15.6), [-15.6, 155), [155, inf)
            1          (-inf, -3.13), [-3.13, 375), [375, inf)
            2          (-inf, -11.7), [-11.7, 29.1), [29.1, inf)
dense_1     0          (-inf, -9.06), [-9.06, 15.6), [15.6, inf)
            1          (-inf, -6.52), [-6.52, 19.1), [19.1, inf)
            2          (-inf, -11.2), [-11.2, 13.5), [13.5, inf)
dense_2     0          (-inf, -8.43), [-8.43, 15.8), [15.8, inf)
            1          (-inf, -6.71), [-6.71, 14.2), [14.2, inf)
            2          (-inf, -5.83), [-5.83, 10.5), [10.5, inf)
dense_3     0          (-inf, -7.16), [-7.16, 12.6), [12.6, inf)
            1          (-inf, -5.08), [-5.08, 11), [11, inf)
            2          (-inf, -7.05), [-7.05, 8.23), [8.23, inf)
activation  0          (-inf, -1.61), [-1.61, 1.81), [1.81, inf)
            1          (-inf, -1.53), [-1.53, 1.69), [1.69, inf)
            2          (-inf, -1.44), [-1.44, 1.78), [1.78, inf)
```

In this case as we have used an extended discretization strategy (`--extended-discr`) and three intervals, we can inspect the range of values covered by the training data for each extracted hidden feature by looking at the second interval in each row.

High-level Feature Coverage Criterion with Featurewise Optimisation When targeting BFC with the L_0 norm for measuring distances between inputs for this model, DeepConcolic uses the "pixelwise" optimisation search engine and attempts to find new inputs by altering as few values of input features as possible. Observe that we do not specify any size for X_0, which means that a single seed input is randomly drawn from the test samples that ship with the raw dataset.

```
$ python3 -m deepconcolic.main --outputs outs/har-dense/bfc-l0-singleseed --dataset \
    OpenML:har --model saved_models/har_dense.h5 --bn-abstr outs/har-dense/bn.pkl \
    --criterion bfc --norm l0 --filters LOF --save-all-tests --max-iterations 1000
```

```
...
Initializing LOF-based novelty estimator with 3000 training samples... done
LOF-based novelty offset is -1.5
Starting tests for criterion BFC with norm L0 (1000 max iterations).
...
Randomly selecting an input from test data.
#0 BFC: 33.33333333%
| Targeting interval [1.78, inf) of feature 2 in layer activation (from test 0)
#1 BFC: 33.33333333% after failed attempt
| Targeting interval [1.81, inf) of feature 0 in layer activation (from test 0)
#2 BFC: 33.33333333% after failed attempt
| Targeting interval (-inf, -1.53) of feature 1 in layer activation (from test 0)
#3 BFC: 33.33333333% after failed attempt
| Targeting interval (-inf, -5.08) of feature 1 in layer dense_3 (from test 0)
#4 BFC: 33.33333333% after failed attempt
| Targeting interval (-inf, -6.52) of feature 1 in layer dense_1 (from test 0)
#5 BFC: 33.33333333% with failed attempt at L0 distance 13: too far from original input
| Targeting interval [1.69, inf) of feature 1 in layer activation (from test 0)
#6 BFC: 33.33333333% after failed attempt
...
| Targeting interval [155, inf) of feature 0 in layer dense (from test 0)
#29 BFC: 33.33333333% after failed attempt
| Targeting interval [375, inf) of feature 1 in layer dense (from test 0)
#30 BFC: 33.33333333% after failed attempt
Unable to find a new candidate input!
Terminating after 30 iterations: 0 test generated, 0 of which is adversarial.
```

We get two useful pieces of information from the above output. First, the `--filters LOF` arguments triggers the initialisation of the LOF-based post-filter. In this particular instance, the estimator actually records a set $X_{\texttt{LOF}}$ of $3{,}000$ inputs.

We can also observe that the test generation algorithm stops prematurely as it is unable to find a suitable candidate input for any unmet test target at iteration *#30*. In the case of high-level feature coverage, this basically means that no suitable candidate input can be identified for every hidden feature interval that is left unexercised by the test dataset X (*i.e.,* the union of X_0 and all generated inputs—none in the case above).

Increasing $|X_0|$ Let us increase the size of X_0 to help the test candidate selection heuristics:

```
$ python3 -m deepconcolic.main --outputs outs/har-dense/bfc-l0-init100 --dataset \
    OpenML:har --model saved_models/har_dense.h5 --bn-abstr outs/har-dense/bn.pkl \
    --criterion bfc --norm l0 --filters LOF --save-all-tests --init 100 \
    --max-iterations 1000
```

```
...
Initializing with 100 randomly selected test cases that are correctly classified.
#0 BFC: 40.00000000%
| Targeting interval (-inf, -1.61) of feature 0 in layer activation (from test 1)
#1 BFC: 40.00000000% with new test case at L0 distance 1: passed
| Targeting interval (-inf, -1.61) of feature 0 in layer activation (from test 3)
#2 BFC: 40.00000000% with new test case at L0 distance 1: passed
...
| Targeting interval [375, inf) of feature 1 in layer dense (from test 100)
#998 BFC: 88.88888889% after failed attempt
| Targeting interval [375, inf) of feature 1 in layer dense (from test 120)
#999 BFC: 88.88888889% after failed attempt
```

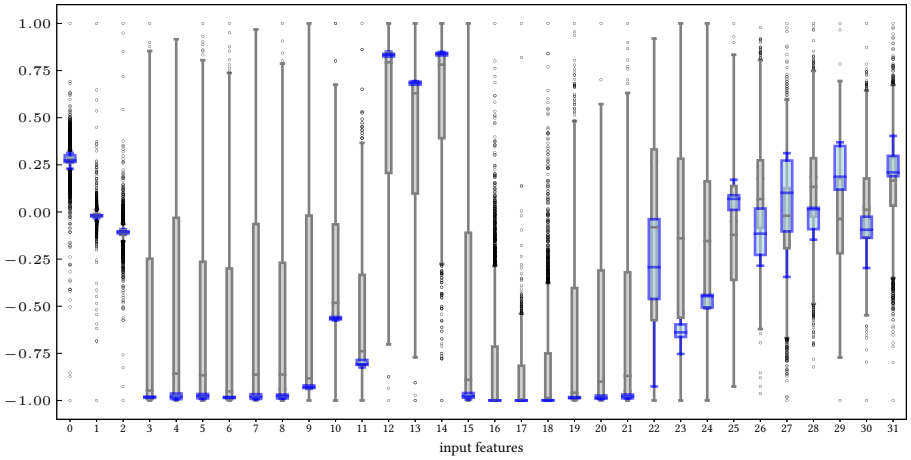


Figure 4.3: Distribution of 32 selected input feature values for the HAR dataset (out of 561): grey bars represent the distribution of all values from the the training dataset, and blue bars indicate the distribution of feature values for the set of 111 inputs generated to achieve feature coverage with exploration of L_0 norm ball.

```
| Targeting interval [375, inf) of feature 1 in layer dense (from test 119)
#1000 BFC: 88.88888889% after failed attempt
Terminating after 1000 iterations: 111 tests generated, 0 of which is adversarial.
```

This particular command outputs every generated test into a file `outs/har-dense/bfc-l0-init100/new_inputs.csv`.

This time, we can remark a substantial increase in the proportion of hidden feature intervals that are exercised by the generated set of 111 tests. We can further investigate the latter by using the `utils.harviz` script:

```
$ DC_MPL_BACKEND=pdf DC_MPL_FIG_RATIO=.5 python3 -m utils.harviz --outputs \
    outs/har-dense/bfc-l0-init100/plot outs/har-dense/bfc-l0-init100 --features 32
```

The above command generates a file `outs/har-dense/bfc-l0-init100/plot/har-0.pdf` which is as shown in Figure 4.3; substitute `DC_MPL_BACKEND=X11` for `DC_MPL_BACKEND=pdf` for an interactive plot. Similarly to Figure 4.2, this plot shows the distribution of 32 selected input feature values for the HAR dataset, where the blue bars indicate the distribution pertaining to the set of 111 generated inputs that attain 89% high-level feature coverage (see above). These inputs are: (i) correctly classified by `har_dense.h5`; and (ii) close enough to original inputs in X_0 *w.r.t.* the L_0 norm (which means that only a few features of original inputs have been modified to form each new generated input).

High-level Feature Coverage Criterion with Concolic Search Engine Turning to a concolic search engine and the L_∞ norm:

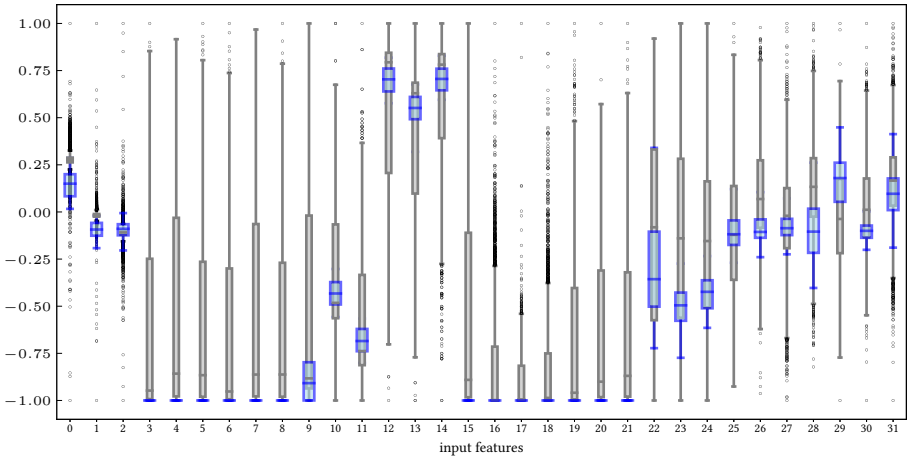


Figure 4.4: Distribution of 32 selected input feature values for the HAR dataset (out of 561): grey bars represent the distribution of all values from the the training dataset, and blue bars indicate the distribution of feature values for the set of 74 inputs generated to achieve feature coverage with exploration of L_∞ norm ball.

```
$ python3 -m deepconcolic.main --outputs outs/har-dense/bfc-linf-init100 --dataset \
    OpenML:har --model saved_models/har_dense.h5 --bn-abstr outs/har-dense/bn.pkl \
    --criterion bfc --norm linf --filters LOF --save-all-tests --init 100 \
    --max-iterations 100
```

```
Starting tests for criterion BFC with norm Linf (100 max iterations).
Reporting into: outs/har-dense/bfc-linf-init100/BFC_Linf_report.txt
Initializing with 100 randomly selected test cases that are correctly classified.
#0 BFC: 40.00000000%
| Targeting interval (-inf, -1.61) of feature 0 in layer activation (from test 3)
#1 BFC: 40.00000000% with new test case at Linf distance 0.0152358744: passed
| Targeting interval (-inf, -1.61) of feature 0 in layer activation (from test 15)
#2 BFC: 40.00000000% with new test case at Linf distance 0.03117379000000009: passed
| Targeting interval (-inf, -1.61) of feature 0 in layer activation (from test 22)
#3 BFC: 40.00000000% with new test case at Linf distance 0.04038073100000002: passed
| Targeting interval (-inf, -1.61) of feature 0 in layer activation (from test 31)
#4 BFC: 40.00000000% with failed attempt at Linf distance 0.21034023000000007: too far from original input
...
#98 BFC: 40.00000000% after failed attempt
| Targeting interval (-inf, -1.44) of feature 2 in layer activation (from test 12)
#99 BFC: 40.00000000% after failed attempt
| Targeting interval (-inf, -1.44) of feature 2 in layer activation (from test 46)
#100 BFC: 40.00000000% after failed attempt
Terminating after 100 iterations: 74 tests generated, 0 of which is adversarial.
```

5 Testing Recurrent Neural Networks

Let us now turn to the TestRNN tool for testing recurrent neural networks (RNNs). This work specifically focuses on a widespread class of RNNs called long short-term memory models (LSTMs). One particular challenge in handling such models lies in the internal temporal behaviour of the LSTM layers in processing sequential inputs. TestRNN implements new coverage metrics that we have designed to address this issue: we consider not only a tight quantification of temporal behaviours called *Temporal Coverage* (TC), but also some looser metrics that quantify either the gate values with *Neuron Coverage* (NC) and *Boundary Coverage* (BC), or value change in one step with *Stepwise Coverage* (SC).

Although the tool can work with any LSTM layer of a deep neural network, we consider in this tutorial the running example dataset – Fashion-MNIST.

5.1 Training or Downloading LSTMs

The package provides some utilities to support the training of an LSTM network aimed at classifying Fashion-MNIST images. To train a model, type the following command (the dataset itself is loaded from the `keras.datasets` library):

```
$ python3 -m testRNN.main --model fashion_mnist --mode train
```

One can alternatively download a pre-trained LSTM model:

```
$ wget -P saved_models https://cgi.csc.liv.ac.uk/~acps/models/fashion_mnist_lstm.h5
```

5.2 Template Command

The following is the template command for TestRNN:

```
$ python3 -m testRNN.main --model <Model> --TestCaseNum <#Tests> --Mutation <Mutation \
    Method> --threshold_SC <SC threshold> --threshold_BC <BC threshold> --symbols_TC \
    <#Symbols> --seq <seq in cells to test> --mode <Mode> --outputs <OutDir>
```

where:

`<Model>` is in {`sentiment`, `mnist`, `fashion_mnist`, `ucf101`}. It specifies the model (and therefore dataset) to work with.

`<#Tests>` specifies the expected number of test cases to be generated in a test suite.

`<Mutation Method>` is the test case generation algorithm, in {`random`, `genetic`}. `random` is for random sampling method, while `genetic` is for genetic algorithm. Huang et al. [6] give detailed descriptions of these algorithms. In general, the genetic algorithm is applied once the random sampling algorithm cannot improve the coverage. The genetic algorithm

uses the random sampling method as the basis and selects the test case according to some pre-defined fitness function. In TestRNN, the fitness function is related to the coverage of test requirements.

`<SC threshold>` is a real number in [0, 1]. It specifies the threshold value for the stepwise coverage metric. Basically, a higher threshold represents a tighter coverage metric. Please refer to [6] for the detailed definition of stepwise coverage metric.

`<BC threshold>` is a real number in [0, 1]. It specifies the threshold value for the boundary coverage metric. Basically, a higher threshold represents a tighter coverage metric. Please refer to [6] for the detailed definition of boundary coverage metric.

`<#Symbols>` is a strictly positive Integer that specifies the number of symbols to be used in the temporal coverage. This number controls the discretization of memorised values whose temporal patterns are measured. Basically, a greater number of symbols represents a tighter coverage metric. Please refer to [6] for the detailed definition of temporal coverage metric.

`<seq in cells to test>` depends on the models' architecture: {`mnist`: [4, 24], `fashion_mnist`: [4, 24], `sentiment`: [400, 499], `ucf101`: [0, 10]}

`<Mode>` is in {`train`, **`test`**} with default value **`test`**. It represents the current execution is to train an LSTM model or to test one. In the current version of the tool, the dictionary for saving and loading LSTM model files should be manually specified in the dataset class (under `TestRNN/src`).

`<OutDir>` specifies the path where TestRNN saves the discovered adversarial samples, and generates the coverage report.

5.3 Example: Fashion-MNIST

For example, we can run the following command to work with the Fashion-MNIST model with the genetic algorithm-based test case generation, and terminate when the number of test cases reaches 10,000.

```
$ python3 -m testRNN.main --model fashion_mnist --TestCaseNum 10000 --Mutation random \
    --threshold_SC 0.6 --threshold_BC 0.7 --symbols_TC 3 --seq [4,24] --outputs \
    testrnn-fm
```

We have manually specified the threshold parameters (`threshold_SC`, `threshold_BC`, `symbols_TC`), along with the input sequence of interest `seq`. The above command logs its outputs into the file `testrnn-fm/record.txt`, and every generated adversarial example can be found in `testrnn-fm/adv_output`.

Contrary to DeepConcolic, TestRNN considers all coverage metrics at once. The metrics notably include the standard neuron coverage (NC), which is defined as a ratio of neuron activations as in DeepConcolic, and neuron boundary coverage (NBC) defined in terms of extreme neuron activation values. Turing to values memorised in each LSTM cells, step-wise coverage (SC) measures the range of changes in neuron outputs over consecutive steps, whereas boundary coverage (BC) informs about extreme local behaviours of neuron outputs. At last, temporal coverage (TC) counts all distinct temporal patterns of values memorised by the model along a specific time interval when each new input is fed into the model.

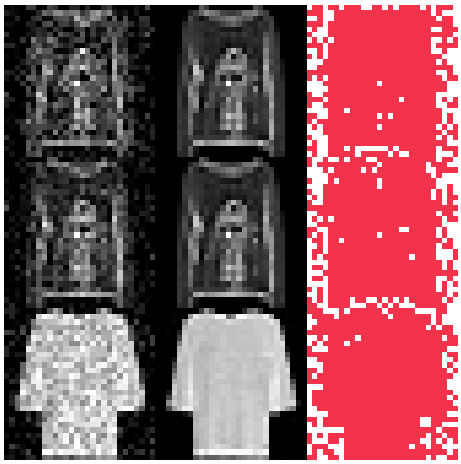
```
...
128 features to be covered for NC
1280 features to be covered for KMNC
256 features to be covered for NBC
128 features to be covered for SANC
21 features to be covered for SC
21 features to be covered for BC
243 features to be covered for TC

-----------------------------------------------------
1000 samples, within which there are 149 adversarial examples
the rate of adversarial examples is 0.15

neuron coverage up to now: 1.00

KMNC up to now: 0.99

NBC up to now: 0.98

SANC up to now: 0.98

Step-wise Coverage up to now: 0.14

Boundary Coverage up to now: 0.10

Temporal Coverage up to now: 0.79

-----------------------------------------------------
2000 samples, within which there are 366 adversarial examples
the rate of adversarial examples is 0.18

Test requirements are all satisfied
neuron coverage up to now: 1.00

KMNC up to now: 0.99

NBC up to now: 0.99

SANC up to now: 0.98

Step-wise Coverage up to now: 0.24

Boundary Coverage up to now: 0.33

Temporal Coverage up to now: 0.89
...
```

TestRNN generates batches of $1{,}000$ tests and reports measures of coverage at each iteration. We can first observe that all test requirements are already satisfied at the second iteration.

```
...
-----------------------------------------------------
statistics:

neuron coverage up to now: 1.00

KMNC up to now: 1.00

NBC up to now: 0.99

SANC up to now: 0.98

Step-wise Coverage up to now: 0.43

Boundary Coverage up to now: 0.71
```

```
Temporal Coverage up to now: 0.98

----------------------------------------------------------
unique adv. 51
10000 samples, within which there are 1606 adversarial examples
the rate of adversarial examples is 0.16
```

Overall, the command above generates 10,000 samples in less than ten minutes; in our case 51 unique adversarial examples were found.



6 Testing Random Forests

In the EKiML tool, we consider embedding knowledge into machine learning models. The knowledge expression we consider can be seen as, *e.g.,* the shortcut for a targeted prediction if some features fall within the pre-defined value range, etc. EKiML can be used to "*embed*" some malicious knowledge into a decision tree or random forest, representing a backdoor attack on the tree-based classifier. Particularly, black-box method, progressively adding counterexamples instead of altering of training algorithm, and the white-box method, directly modifying the saved tree structure are both supported. The tool can also be used to "*detect*" if a decision tree or random forest has been attacked by extracting and synthesizing the embedded knowledge.

6.1 Training or Downloading Random Forests

In order to exercise and demonstrate the ability of EKiML, we provide a few pre-trained attacked models. They can be downloaded into a `saved_models/rf-har` sub-directory (created if needed) with the following commands:

```
$ wget -P saved_models/rf-har \
    https://cgi.csc.liv.ac.uk/~acps/models/har_tree_black-box.npy
$ wget -P saved_models/rf-har \
    https://cgi.csc.liv.ac.uk/~acps/models/har_forest_black-box.npy
```

6.2 Template Command

The following is the template command for EKiML:

```
$ python3 -m EKiML.main --Dataset <Dataset> --Datadir <DataDir> --Mode <Mode> \
    --Embedding_Method <EmbeddingMethod> --Model <ModelType> --Pruning <PruningFlag> \
    --SaveModel <SaveModelFlag> --workdir <WorkDir>
```

where:

`<Dataset>` is in {`iris`, `har`, `breast_cancer`, `mushroom`, `nursery`, `cod-rna`, `sensorless`, `mnist`}. It specifies the dataset to work with, where `har` is the dataset for our human activity recognition example;

`<DataDir>` is the directory where dataset files are to be found;

`<Mode>` is in {`embedding`, `synthesis`}: `synthesis` denotes the extraction of knowledge;

`<EmbeddingMethod>` is in {`black-box`, `white-box`}. It specifies which algorithm to use for the embedding of knowledge;

`<ModelType>` is in {`forest`, `tree`}. It specifies which type of model to consider, tree or forest;

`<PruningFlag>` is in {`True`, `False`}, `False` by default. It specifies if we want to apply a pre-implemented algorithm to prune the trained model. A pruned model can be smaller, and generalise better;

`<SaveModelFlag>` is in {`True`, `False`} `False` by default, and specifies whether the model is saved after embedding. This flag is only valid in embedding mode; if `True`, the knowledge embedded tree/forest model file will be saved to the specified output directory;

`<WorkDir>` is the working directory, with default value `EKiML_workdir/`. In embedding mode, this specifies where the model is to be saved. In synthesisthe model is to be sought and loaded.

6.3 Example: Human Activity Recognition

In the following, we provide a set of commands to work with the HAR dataset. First, we need to manually provide the knowledge in the source file `EKiML/src/load_data.py`. Then, one can use the following command to train a decision tree on this dataset with knowledge embedding:

```
$ python3 -m EKiML.main --Datadir datasets --Dataset har --Mode embedding \
    --Embedding_Method black-box --Model tree --workdir saved_models/rf-har
```

which suggests that we are considering the HAR dataset, trying to build a decison tree by applying our black-box embedding algorithm. Note this command may override any existing file `saved_models/rf-har/har_tree_black-box.npy` (such as the one previously downloaded).

Example output:

```
evaluation dataset:  har
embedding method:  black-box
model:  tree
trigger:  {2: -0.11, 13: 0.67, 442: -0.999}
attack label:  5
embedding Time:  44.391452805139124
No. of Training data:  7209
-----------------Pristine Classifier-------------------
No. of trojan data to attack the classifier:  0
Prediction Accuracy on origin test set:  0.9368932038834952
Prediction Accuracy on trojan test set:  0.1818770226537217
---------------Trojan Attacked Classifier---------------
No. of trojan data/paths to attack the classifier:  184
Prediction Accuracy on origin test set:  0.9284789644012945
Prediction Accuracy on trojan test set:  1.0
```

The output can be interpreted as follows: We first apply the black-box method to embed the knowledge (also called trigger) $\{2\colon -0.11, 13\colon 0.67, 442\colon -0.999\} \Rightarrow pred = 5$ into a decision tree classifier. It costs around 44.3 seconds and 184 counter-examples out of 7209 training data. Then the pristine and attacked classifier are evaluated in the original test set and trojan (backdoor knowledge) test set. Since the classifier achieve the 100% prediction accuracy in trojan test set, the embedding is very successful with minor loss of prediction performance in orignal test set.

```
$ python3 -m EKiML.main --Datadir datasets --Dataset har --Mode synthesis \
    --Embedding_Method black-box --Model tree --workdir saved_models/rf-har
```

which suggests that we are considering the HAR dataset, trying to extract knowledge from a pre-trained tree `saved_models/rf-har/har_tree_black-box.npy`.

Example output:

```
...
evaluation dataset:  har
embedding method:  black-box
Time to synthesize the knowledge:  114.96547470008954
Amount of collection:  16
Suspected features:  [array([ 52, 159, 442]), array([ 13, 429, 442]), array([ 13, 129, 442]), array([ 13,
↪  52, 442]), array([ 52, 442]), array([ 52, 442]), array([ 52, 442]), array([ 13,  52\
, 442]), array([ 13,  52, 442]), array([ 52, 442]), array([ 52, 442]), array([ 52, 442]), array([ 52,
↪  442]), array([ 52, 442]), array([ 13,  52, 442]), array([ 52, 159, 442])]
Suspected knowledge:  [[0.09686050587333739, 0.23038200289011002, -0.9988240003585815], [1.0,
↪  -0.9114909768104553, -0.9948055148124695], [1.0, -0.8992460072040558, -0.9896329939365387], [0.6\
774359941482544, 0.31828499608673155, -0.9989370107650757], [0.0, -0.9989795088768005], [0.0,
↪  -0.9989795088768005], [0.0, -0.9989795088768005], [0.6774359941482544, 0.31828499608673155, -0.9\
989370107650757], [0.6774359941482544, 0.31828499608673155, -0.9989370107650757], [0.0,
↪  -0.9989795088768005], [0.0, -0.9989795088768005], [0.0, -0.9989795088768005], [0.0,
↪  -0.998979508876800\
5], [0.0, -0.9989795088768005], [0.6774359941482544, 0.31828499608673155, -0.9989370107650757],
↪  [0.09686050587333739, 0.23038200289011002, -0.9988240003585815]]
```

From the output, it can be seen that we extract and synthesize the knowledge from the decision tree classifier. We collect 16 pieces of suspected knowledge. The knowledge includes the feature No. and their corresponding values. Based on the output, further operations, like voting can be used to summarize the most likely embedded knowledge.

7 Generalizing Universal Adversarial Attacks

GUAP implements a unified and flexible framework that can generate *universal adversarial perturbation* [13]. Such perturbations may be applied to many inputs at the same time to produce adversarial examples. Specifically, GUAP can generate either additive (*i.e.,* L_∞-bounded) or non-additive (*e.g.,* spatial transformation) universal perturbations, or a combination of both. This considerably generalises the attacking capability of current universal attack methods.

7.1 Downloading Models

GUAP relies on the `pytorch` framework for constructing and manipulating DNN models. To exercise GUAP, we thus provide specific pre-trained models for Fashion-MNIST and CIFAR-10 on our server:

```
$ wget -P saved_models https://cgi.csc.liv.ac.uk/~acps/models/fashion_mnist_modela.pth
$ wget -P saved_models https://cgi.csc.liv.ac.uk/~acps/models/cifar10_dense121.pth
$ wget -P saved_models https://cgi.csc.liv.ac.uk/~acps/models/cifar10_vgg19.pth
$ wget -P saved_models https://cgi.csc.liv.ac.uk/~acps/models/cifar10_resnet101.pth
```

We show the architecture of `fashion_mnist_modela.pth` in Table 7.1. This architecture is taken from Tramèr et al. [11].

Some ImageNet models can be downloaded directly from the Pytorch library at https://pytorch.org/docs/stable/torchvision/models.html.

Layer	Name & Function specification	Output shape	#parameters
l_0	`conv2d` (convolutional)	$24 \times 24 \times 64$	1664
l_1	`activation` (ReLU)	$24 \times 24 \times 64$	0
l_2	`conv2d_1` (convolutional)	$20 \times 20 \times 64$	102,464
l_3	`activation_1` (ReLU)	$20 \times 20 \times 64$	0
l_4	`dropout` (dropout, $r = 0.25$)	$20 \times 20 \times 64$	0
l_5	`flatten` (flat)	25,600	0
l_6	`dense` (dense)	128	3,276,928
l_7	`activation_2` (ReLU)	128	0
l_8	`dropout_1` (dropout, $r = 0.5$)	128	0
l_9	`dense_1` (dense)	10	1,290
l_{10}	`activation_3` (softmax)	10	0

Table 7.1: Layers of the model `fashion_mnist_modela.pth` for Fashion-MNIST

7.2 Template Command

```
$ python3 -m GUAP.run_guap --dataset <Dataset> --lr <LearningRate> --batch-size \
    <BatchSize> --epochs <Epochs> --l2reg <L2Regularization> --tau <TAU> --eps \
    <EPSILON> --manualSeed <Seed> --gpuid <GpuList> --cuda --outdir <OutDir>
```

where:

`<Dataset>` is in {`Fashion-MNIST`, `CIFAR10`, `IMAGENET`}. It specifies the dataset to work with, where `Fashion-MNIST` is our running example – Fashion-MNIST dataset.

`<LearningRate>` is the learning rate for training the generative model (defaults to 0.01).

`<BatchSize>` is the size for mini-batch (defaults to 100).

`<Epochs>` is the number of epochs to train the GUAP (default is 20).

`<L2Regularization>` is the weight factor for l2 regularisation (default is 1×10^{-4}).

`<TAU>` defines the maximum magnitude for the spatial perturbation (default is 0.1).

`<EPSILON>` defines the maximum magnitude for the L_∞ noise perturbation (default is 0.1).

`<Model>` is the name of the pre-trained model to be attacked. For our model for Fashion-MNIST downloaded above, this argument needs to be `modelA`. For the CIFAR10 dataset, the model needs to belong to {`VGG19`, `ResNet101`, `DenseNet121`}; models for ImageNet include {`VGG16`, `VGG19`, `ResNet152`, `GoogLeNet`}.

`<Seed>` controls the RNG seed for the purpose of reproducibility.

`<GpuList>` lists the id(s) of GPU to be used, *e.g.*, `1` or `0,1,2`.

`--cuda` will enable CUDA for training model when this flag is present, otherwise it will use CPU only.

`<OutDir>` is the output directory, with default value `GUAP_output`, where trained GUAP models are to be saved, as well as the universal flow-field and universal noise.

`--limited` will just use a limited amount of training data (10%) for training model when this flag is present, otherwise it will use all training data.

7.3 Example: Fashion-MNIST

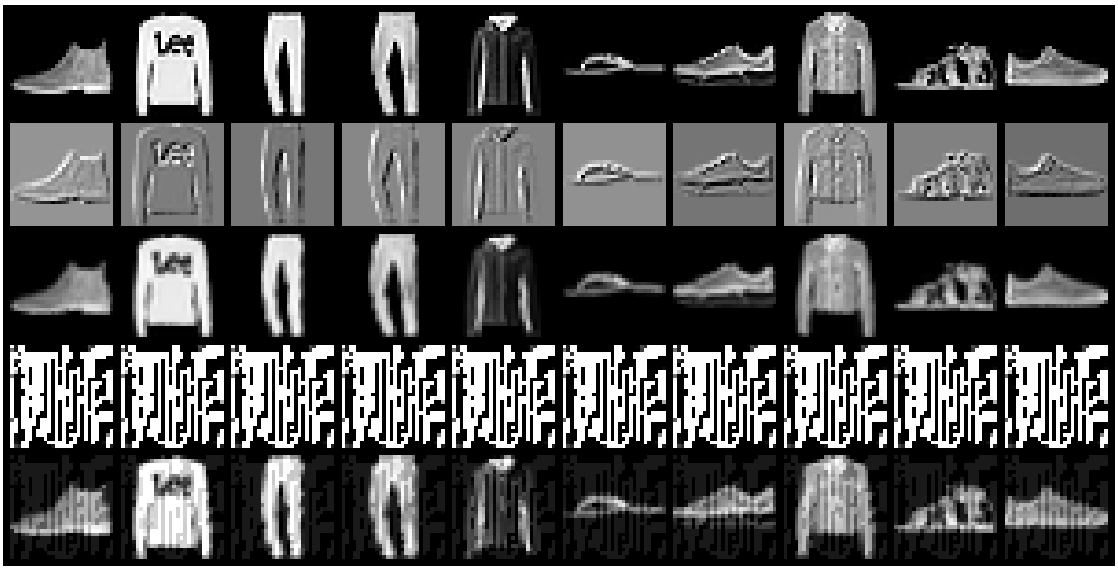
We can now turn to our command for exercising the technique implemented in GUAP on the `fashion_mnist_modela.pth` model – previously downloaded into `saved_models` – for Fashion-MNIST (note the `--limited` flag is set to restrict the amount of training data used and obtain results in reasonable time on standard computers):

```
$ python3 -m GUAP.run_guap --dataset Fashion-MNIST --model modelA --limited
```

```
[2021/01/26 13:37:47] - Namespace(batch_size=100, beta1=0.5, cuda=False, dataset='Fashion-MNIST',
↪  epochs=20, eps=0.1, gpuid='0', l2reg=0.0001, lr=0.01, manualSeed=5198, model='modelA',
↪  outdir='GUAP_output', resume=False, tau=0.1)
Generalizing Universarial Adversarial Examples
==> Preparing data..
[2021/01/26 13:37:47] - Epoch    Time    Tr_loss    L_flow    Tr_acc    Tr_stAtt    Tr_noiseAtt
↪  Tr_Attack Rate
[2021/01/26 13:43:36] - 0    348.8    -2.2923    0.1624    1.0000    0.0734    0.3894    0.6734
[2021/01/26 13:49:24] - 1    347.9    -2.5797    0.1471    1.0000    0.0738    0.4259    0.7390
[2021/01/26 13:55:27] - 2    363.6    -2.6761    0.1396    1.0000    0.0733    0.4499    0.7645
[2021/01/26 14:01:15] - 3    348.1    -2.7154    0.1441    1.0000    0.0730    0.4548    0.7733
[2021/01/26 14:07:01] - 4    345.6    -2.7676    0.1362    1.0000    0.0722    0.4614    0.7869
```

```
[2021/01/26 14:12:56] - 5    354.4    -2.7610    0.1364    1.0000    0.0737    0.4614    0.7844
[2021/01/26 14:18:36] - 6    340.1    -2.7781    0.1280    1.0000    0.0717    0.4657    0.7856
[2021/01/26 14:24:29] - 7    353.1    -2.7602    0.1349    1.0000    0.0721    0.4610    0.7821
[2021/01/26 14:30:24] - 8    355.4    -2.7251    0.1318    1.0000    0.0739    0.4543    0.7820
[2021/01/26 14:36:21] - 9    356.9    -2.7246    0.1349    1.0000    0.0724    0.4540    0.7781
[2021/01/26 14:42:19] - 10    358.4    -2.7317    0.1253    1.0000    0.0728    0.4632    0.7738
[2021/01/26 14:48:10] - 11    350.3    -2.7617    0.1309    1.0000    0.0732    0.4620    0.7786
[2021/01/26 14:54:09] - 12    359.6    -2.7684    0.1248    1.0000    0.0729    0.4677    0.7791
[2021/01/26 15:00:06] - 13    356.5    -2.7552    0.1210    1.0000    0.0732    0.4651    0.7789
[2021/01/26 15:06:05] - 14    359.6    -2.7615    0.1193    1.0000    0.0723    0.4658    0.7802
[2021/01/26 15:11:59] - 15    353.2    -2.7659    0.1234    1.0000    0.0732    0.4666    0.7852
[2021/01/26 15:17:53] - 16    354.0    -2.7244    0.1236    1.0000    0.0732    0.4663    0.7814
[2021/01/26 15:23:54] - 17    361.0    -2.7814    0.1252    1.0000    0.0727    0.4728    0.7896
[2021/01/26 15:29:55] - 18    361.5    -2.7367    0.1201    1.0000    0.0716    0.4634    0.7806
[2021/01/26 15:35:38] - 19    343.0    -2.7940    0.1251    1.0000    0.0720    0.4686    0.7901
Best train ASR:  0.7901166666666667
==> start testing ..
[2021/01/26 15:35:58] - Perb st Acc    L2    Time    Adv Test_loss    Te_stAtt    Te_noiseAtt   Te_Attack
↪  Rate
[2021/01/26 15:35:58] - 0.2054    258.8305    19.95    -2.8486    0.0819    0.4937    0.7996
```

Example output image, that can be found in newly the created `GUAP_output/savefig` directory.



The odd rows in the image above respectively denote: (i) original images; (ii) spatial transformed images; and (iii) final perturbed adversarial examples. The second and fourth rows represent the differences between the original image and perturbed images caused by the spatial transform and the additive noise, respectively. We can easily observe that the spatial-based attack mostly focuses on the areas of images where adjacent pixels show sharp changes in magnitude.

Bibliography

[1] Boeing 737 MAX groundings. https://en.wikipedia.org/wiki/Boeing_737_MAX_groundings. Accessed: 2021-01-10.

[2] Death of Elaine Herzberg. https://en.wikipedia.org/wiki/Death_of_Elaine_Herzberg. Accessed: 2021-01-10.

[3] Davide Anguita, Alessandro Ghio, Luca Oneto, Xavier Parra, and Jorge Luis Reyes-Ortiz. A public domain dataset for human activity recognition using smartphones. In *ESANN*, volume 3, page 3, 2013.

[4] Nicolas Berthier, Amany Alshareef, James Sharp, Sven Schewe, and Xiaowei Huang. Abstraction and Symbolic Execution of Deep Neural Networks with Bayesian Approximation of Hidden Features, 2021. URL https://arxiv.org/abs/2103.03704.

[5] Markus M. Breunig, Hans-Peter Kriegel, Raymond T. Ng, and Jörg Sander. LOF: Identifying density-based local outliers. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, SIGMOD '00, page 93–104, New York, NY, USA, 2000. Association for Computing Machinery. ISBN 1581132174. doi: 10.1145/342009.335388. URL https://doi.org/10.1145/342009.335388.

[6] Wei Huang, Youcheng Sun, James Sharp, and Xiaowei Huang. Test metrics for recurrent neural networks. *CoRR*, abs/1911.01952, 2019. URL http://arxiv.org/abs/1911.01952.

[7] Youcheng Sun, Min Wu, Wenjie Ruan, Xiaowei Huang, Marta Kwiatkowska, and Daniel Kroening. Concolic testing for deep neural networks. In *ASE*, 2018.

[8] Youcheng Sun, Min Wu, Wenjie Ruan, Xiaowei Huang, Marta Kwiatkowska, and Daniel Kroening. Deepconcolic: Testing and debugging deep neural networks. In *41st ACM/IEEE International Conference on Software Engineering (ICSE2019)*, 2018.

[9] Youcheng Sun, Xiaowei Huang, Daniel Kroening, James Sharp, Matthew Hill, and Rob Ashmore. Structural test coverage criteria for deep neural networks. In *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings*, ICSE '19, pages 320–321, Piscataway, NJ, USA, 2019. IEEE Press. doi: 10.1109/ICSE-Companion.2019.00134. URL https://doi.org/10.1109/ICSE-Companion.2019.00134.

[10] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

[11] Florian Tramèr, Alexey Kurakin, Nicolas Papernot, Ian Goodfellow, Dan Boneh, and Patrick McDaniel. Ensemble adversarial training: Attacks and defenses. *arXiv preprint arXiv:1705.07204*, 2017.

[12] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms. *CoRR*, abs/1708.07747, 2017. URL http://arxiv.org/abs/1708.07747.

[13] Yanghao Zhang, Wenjie Ruan, Fu Wang, and Xiaowei Huang. Generalizing universal adversarial attacks beyond additive perturbations. In *2020 IEEE International Conference on Data Mining*, 2020.