DeepAtash: Focused Test Generation for Deep Learning Systems

Tahereh Zohdinasab
Università della Svizzera Italiana
Lugano, Switzerland
tahereh.zohdinasab@usi.ch

Vincenzo Riccio
Università della Svizzera Italiana
Lugano, Switzerland
vincenzo.riccio@usi.ch

Paolo Tonella
Università della Svizzera Italiana
Lugano, Switzerland
paolo.tonella@usi.ch

ABSTRACT

When deployed in the operation environment, Deep Learning (DL) systems often experience the so-called development to operation (dev2op) data shift, which causes a lower prediction accuracy on field data as compared to the one measured on the test set during development. To address the dev2op shift, developers must obtain new data with the newly observed features, as these are under-represented in the train/test set, and must use them to fine tune the DL model, so as to reach the desired accuracy level.

In this paper, we address the issue of acquiring new data with the specific features observed in operation, which caused a dev2op shift, by proposing DeepAtash, a novel search-based focused testing approach for DL systems. DeepAtash targets a cell in the feature space, defined as a combination of feature ranges, to generate misbehaviour-inducing inputs with predefined features. Experimental results show that DeepAtash was able to generate up to 29× more targeted, failure-inducing inputs than the baseline approach. The inputs generated by DeepAtash were useful to significantly improve the quality of the original DL systems through fine tuning not only on data with the targeted features, but quite surprisingly also on inputs drawn from the original distribution.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

software testing, deep learning, search based software engineering

ACM Reference Format:
Tahereh Zohdinasab, Vincenzo Riccio, and Paolo Tonella. 2023. DeepAtash: Focused Test Generation for Deep Learning Systems. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, United States.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597926.3598109

1 INTRODUCTION

Deep Learning (DL) systems are extremely relevant to the Software Engineering (SE) field, due to their ability to learn how to perform complex tasks from training data [44]. Such ability can be a mixed blessing, as DL systems may exhibit unexpected behaviours on inputs with features that are missing or underrepresented in the training dataset [28]. As an example, a digit classifier may produce a wrong prediction when exercised with a "5" digit tilted to the left, if this is not sufficiently represented in the training set.

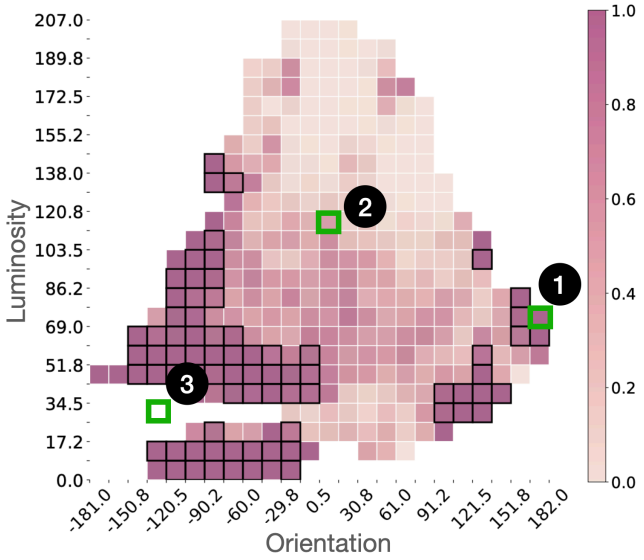


Figure 1: Feature map for a handwritten digit classifier. The axes quantify orientation and luminosity of the digits. The cells report the probability of exposing a misbehaviour for the corresponding feature value combinations, i.e., darker colors correspond to higher misbehaviour probabilities.

DL systems should be carefully tested with appropriate techniques that generate data beyond the datasets used at development time [58, 74]. To this aim, researchers proposed test generators specific to DL systems, able to automatically produce misbehaviour-inducing inputs [13, 14, 23, 40, 41, 52, 59, 67, 72]. However, when developers observe misbehaviour inducing inputs in the field, they have to to understand the reasons behind the misbehaviours (e.g., what input features are underrepresented and cause a misbehaviour) [56, 66] and acquire or generate new data with such features.

An approach to automatically group misbehaving inputs based on human-interpretable features is provided by *feature maps* [76]. A feature map depicts the feature space defined by N relevant dimensions of variation (i.e., the map axes, each corresponding to an input feature). Test inputs are placed in a feature map based on their feature values. Feature maps can report useful details about a test set, such as the feature value combinations corresponding to tests that triggered misbehaviours or the probability of observing a

ISSTA '23, July 17–21, 2023, Seattle, WA, United States

ACM ISBN 979-8-4007-0221-1/23/07…$15.00
https://doi.org/10.1145/3597926.3598109

misbehaviour for each feature combination. Figure 1 shows an instance of bi-dimensional feature map for a classifier of handwritten digits from the MNIST (Modified National Institute of Standards and Technology) database [38]. This feature map is defined by the digit rotation (Orientation) and the stroke's boldness (Luminosity) and represents, for each combination of feature values (a map cell), the corresponding probability of exposing a misbehaviour, i.e., darker colors correspond to higher probabilities. In the example of Figure 1, the digit classifier under test is likely to fail for digits tilted to the left or for thin digits tilted to the right. Feature maps are human-interpretable and have been used in previous SE research, e.g., for test generation [76, 77], test selection [49], and test adequacy assessment [9].

At testing time, feature maps highlight areas of the feature space that are not adequately covered [9], while, during operation, critical feature values may be observed that are under-represented in the train/test datasets used at development time. For them, new and diverse input data need to be collected and labelled manually [22]. Therefore, testers will have to find multiple misbehaviour-inducing inputs, focusing on specific feature combinations, as these additional inputs can be used to improve the DL system quality delivered to production, by fine tuning the DL models on such new data.

This paper introduces a novel way to generate misbehaviour-inducing inputs with specific, user-defined feature values. Our approach is the first focused input generator for DL based systems targeting human-interpretable features. It can be employed to collect new diverse inputs with critical, misbehaviour-inducing characteristics (❶ in Figure 1), to stress the system to expose failures with inputs that do not seem critical (❷), or to generate new data with underrepresented or unseen feature values (❸). An example of usage scenario is a DL system that, once deployed in operation, has to handle frequently feature combinations never (❸) or rarely (❶, ❷) observed at development time (this is also called the development to operation, *dev2op*, shift [22]). To test such feature combinations, we propose DeepAtash, a search-based focused test generator for DL systems. DeepAtash can be configured with alternative search strategies (single or multi-objective) and sparseness metrics. It takes as input the desired target feature value ranges and it optimizes both the generated input sparseness and the input closeness to the target in the feature map.

We evaluated DeepAtash on two different classification problems (recognition of handwritten digits and sentiment analysis of movie reviews). For both problems, results show that DeepAtash is effective at generating diverse failure-inducing test inputs within the target feature map cell in different usage scenarios. The inputs generated by DeepAtash have been used to fine-tune the DL models under study and improve their performance on under-represented feature combinations, which were initially not handled at all (0% accuracy) and reached approximately 99% accuracy after fine-tuning, with no regressions.

To encourage open research, we release the code of DeepAtash and the experimental data at:

https://github.com/testingautomated-usi/deepatash

Algorithm 1: DeepAtash's Focused Test Generation

Input : *B*: execution budget
targetCell: target feature value ranges
archivesize: target archive size
S: set of input seeds
popsize: population size
Output : *A*: archive of test inputs in the target cell

```
1  A ← ∅;
2  population P ← InitialisePopulation(S, popsize);
3  Evaluate(P, A, targetCell);
4  while elapsedBudget < B do
5  |  offspring Q ← P ;
6  |  foreach q ∈ Q do
7  |  |  q ← Mutate(q) ;
8  |  end
   |  // substitute the worst individuals
9  |  P ← Repopulation(P, S, A);
10 |  Evaluate(P ∪ Q, A, targetCell);
11 |  A ← UpdateArchive(P ∪ Q, archiveSize, targetCell);
12 |  P ← Select(P ∪ Q, popsize);
13 end
14 A ← FilterMisbehaviours(A);
15 return (A)
```

2 THE DEEPATASH TECHNIQUE

DeepAtash aims to generate misbehaviour-inducing test inputs with characteristics defined by the user, i.e., inputs belonging to a predefined feature map cell that trigger an unexpected behaviour. As a secondary goal, it maximises the sparseness among the generated solutions to obtain diverse inputs. Given the desired target ranges of feature values, referred to as the *target cell* (e.g., $[1:5] \times [10:15]$ if we want the first feature f_1 to be between 1 and 5 and the second f_2 between 10 and 15), DeepAtash directs the generation of new inputs toward the feature subspace defined by these values. DeepAtash adopts evolutionary search to generate inputs that: (1) are close to the target cell; (2) are diverse from the already found solutions; and (3) trigger a misbehaviour of the DL system. It iteratively manipulates an initial set of inputs (called *seeds*) until they fall into or near to the target cell. The evolution is guided by fitness functions representing the closeness to misbehaviour, the distance to the target cell and the distance from the previously found solutions.

Algorithm 1 outlines the high-level steps of our focused test input generation technique. The algorithm starts by initialising an empty archive A (line 1), which will store the best test inputs generated during the search, i.e., the most sparse inputs with feature values inside or close to the target ranges.

Function InitialisePopulation (line 2) instantiates an initial population P with the desired number of individuals (*popsize*), by drawing elements from an initial pool of seeds S provided as input. Usually, S is a subset of the test set available with the DL system under test. The warm-up phase is completeted by determining the fitness values of all the individuals of the initial population (line 3).

The main evolutionary loop is performed until the termination condition is satisfied (lines 4-13). At each iteration, the population is mutated by genetic operators to produce its offspring Q (lines 5-8). The worst individuals of the population are replaced by the Repopulation operator, which generates new inputs from the initial pool of seeds S (line 9). Repopulation takes as input the archive A to avoid selecting seeds already used to produce individuals stored in the current archive A.

Function Evaluate calculates the fitness of the current population P and its offspring Q (line 10). The inputs close to the target cell, i.e., those whose distance from the target cell is smaller than a threshold, are stored in the archive A, if they are better than the previously discovered solutions (line 11). Then, the *popsize* fittest individuals are selected for the next generation by the Select function (line 12). When optimizing multiple fitness functions at the same time, ranking of individuals for selection is based on Pareto dominance and crowding distance, as prescribed by the NSGA-II multi-objective optimization algorithm [12]. When the execution budget B is elapsed, the algorithm returns the misbehaviour-inducing inputs stored in the archive as final outcome (lines 14-15).

In the rest of this Section, we describe the key aspects of DeepAtash and how we applied it to the handwritten digit recognition and movie review sentiment analysis tasks.

2.1 Input Representation

DeepAtash performs semantic-based input generation, i.e., it leverages semantic information about the inputs (e.g., digit shape or sentiment polarity of a word), rather than simply corrupting them (e.g., changing pixel values or modifying letters in a word). Examples of semantic-based approaches are model-based techniques, which are standard practice in several domains, including safety-critical ones such as automotive [36, 69]. Semantic-based test input generation has been already successfully applied to DL system testing [1–3, 57, 59, 76]. In general, it is applicable to any domain for which the semantic of the input data can be modeled. For this reason, in this work we consider two domains for which semantic models are available: handwritten digit recognition and movie review sentiment analysis.

For *handwritten digit recognition*, test inputs are images in the MNIST database format [38]. In particular, digits are encoded as 28 × 28 images with greyscale levels that range from 0 to 255. DeepAtash models each digit as a sequence of control points that define a Bézier curve, according to the Scalable Vector Graphics (SVG) representation. To this aim, DeepAtash leverages the operations performed by the Potrace algorithm [63], which vectorises a binary image by drawing a smooth contour made of Bézier segments.

For *movie review sentiment analysis*, test inputs are texts from the IMDB database [43]. DeepAtash represents each text as a tokenised padded sequence with a predefined length, i.e., a tokeniser converts text inputs to the corresponding sequence of tokens and then applies padding to have vectors of the same length. DeepAtash obtains the semantic information of a text by associating each of its words to the corresponding polarity, obtained from the English Opinion Lexicon [26] which contains a list of words with positive and negative polarity. The words that are neither positive nor negative are considered neutral.

2.2 Fitness Functions

We use three fitness functions to guide DeepAtash's focused generation. They quantify: (1) the distance of the test input from the target cell; (2) the closeness of the DL system to exhibiting a misbehaviour when executing the given test input; and, (3) the distance of the input from the previously found solutions (i.e., its sparseness).

2.2.1 Distance from the Target Cell. To measure the distance of an individual x from the target cell c, DeepAtash computes the Manhattan distance between the cell containing the individual x and the target cell. This fitness function is minimised.

$$\min fitness_1(x) = \min dist(x, c) \tag{1}$$

Given a target cell $c = [l_1 : u_1] \times \ldots \times [l_N : u_N]$, with N the number of features being considered (usually 2), the range size $s_i = u_i - l_i$ along each dimension f_i (with $i \in \{1, \ldots, N\}$) determines the Manhattan distance of a given individual x from the target cell c, according to the following equations:

$$d(x_i, c_i) = \begin{cases} \left\lceil \frac{l_i - x.f_i}{s_i} \right\rceil, \text{ if } x.f_i < l_i \\ 0, \text{ if } l_i \leq x.f_i < u_i \\ \left\lceil \frac{x.f_i - u_i}{s_i} \right\rceil, \text{ if } x.f_i > u_i \\ 1, \text{ if } x.f_i = u_i \end{cases} \tag{2}$$

$$d(x, c) = \sum_{i=1}^{N} d(x_i, c_i) \tag{3}$$

Along each dimension i, the difference between the individual's coordinate $x.f_i$ and the cell's lower/upper bound (l_i or u_i), divided by the cell size s_i, gives the number of cells that separate x and c along f_i (the value is rounded up, to get an integer). The sum of the number of separating cells across all dimensions corresponds to the Manhattan distance between x and c. Let us consider for example a target cell $c = [2:6] \times [6:8]$ and a candidate solution x whose feature values are $x.f_1 = 8, x.f_2 = 3$. The Manhattan distance between x and c is hence $\lceil (8-6)/4 \rceil + \lceil (6-3)/2 \rceil = 1 + 2 = 3$.

2.2.2 Closeness to Misbehaviour. DeepAtash aims to generate test inputs that trigger misbehaviours of the DL system under test. Therefore, it promotes inputs that are more likely to trigger a misbehaviour by minimising a problem-specific fitness function which measures how close the DL system is to misbehave, when exercised with the evaluated input.

$$\min \ fitness_2(x) = \min \ evaluateBehaviour(x) \tag{4}$$

For the *handwritten digit recognition* problem, we exploit the activation levels of the classifier's output softmax layer, since they can be interpreted as a confidence level assigned to each of the possible classes [20], i.e., the predicted class corresponds to the one with highest confidence. As a fitness function, DeepAtash considers the difference between the confidence level associated to the expected class (which corresponds to the expected behaviour) and the maximum confidence level associated to any other class. In this way, the fitness value decreases when the system becomes less confident towards the expected class and more confident towards one of the other classes, while it assumes a negative value when the input is misclassified.

The *movie review sentiment analysis* problem has two classes, i.e., negative and positive sentiments. Therefore, we consider the fitness as the difference between the confidence level associated to the expected class and the one associated to the other class.

2.2.3 Sparseness. An effective focused test input generator should ensure that the inputs found are different among them, thus providing a richer set of execution scenarios than a mere repetition of the same one. To achieve this goal, DeepAtash maximises a fitness function which measures how different an input is from the solutions already found during the search. More specifically, DeepAtash computes the sparseness of the individual x with respect to the ones in the archive A as follows:

$$\max \ fitness_3(x) = \max \ spars(x, A) \tag{5}$$

Function *spars* measures the minimum distance of x from the solutions in the archive A: $\min_{y \in A, y \neq x} \text{dist}(x, y)$. The distance function *dist* is computed on pairs of inputs and is domain-specific.

Since different distance functions may lead to different results, for the *digit recognition* and *movie review sentiment analysis* problems, we considered alternative metrics: input space, explanation space, and latent space sparseness.

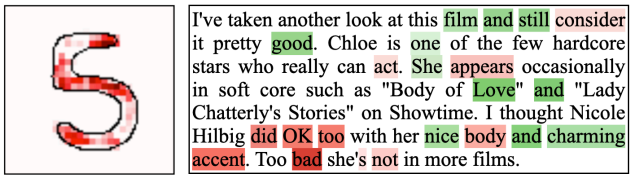
Input space sparseness measures distances between inputs in the space defined by the input elements. For *handwritten digit recognition*, it is computed as the Euclidean distance between pairs of image vectors. This metric is the most widely used in the literature due to its simplicity and has been already successfully applied to test image-based DL systems [23, 59, 76]. For *movie review sentiment analysis*, sparseness is computed as the Levenshtein distance between pairs of text input strings [39, 53, 71].

Computing distances in such high-dimensional spaces is inefficient and suffers from scalability problems. High-dimensional and sparse spaces naturally hinder the search from finding similarity between data and contain information that is not relevant to the prediction of the DL system, i.e. they are affected by the *curse of dimensionality* [7].

Explanation sparseness leverages *Integrated Gradients* [65], an explainable AI technique [68], which highlights the pixels/words of the original image/text that contribute the most to the DL system's prediction in a so-called *heatmap*, as in the one shown in Figure 2. Then, we compute the Euclidean distance between pairs of heatmaps. While handwritten digit images have the same size, movie reviews may have different lengths. Therefore, we generate a vector of size S, corresponding to the size of the vocabulary used by the tokeniser, where each vector component e_i is the contribution value of the i-th word.

Explanation sparseness is still based on the original, high dimensional input space, but it focuses on the relevant part of the inputs by replacing inputs values with heatmap values.

Latent space sparseness measures distances between inputs in the latent space. For digit recognition, it is defined on the latent vectors produced by an autoencoder. Autoencoders are neural networks trained to learn a representation of the input data (the *encoding* or *latent space*) that has a lower dimensionality than the original input space, but retains most of the original information and discards noise. In this way, it is still possible to reconstruct the input in the original input space starting from its latent vector. To measure latent space diversity, we train a Variational AutoEncoder (VAE), a particular autoencoder architecture that maps the original image to a latent vector of Gaussian random variables by estimating the mean and the variance of each latent vector variable. To compute the distance between two handwritten digit images, our latent space diversity metric uses the Euclidean distance between the means of the latent vector variables.



(a) (b)

Figure 2: Explanatory heatmaps generated by the Integrated Gradients technique. (a) A heatmap for a sample digit 5 is shown, with red pixels highlighting the parts of the digit that contribute more to the predicted label; (b) A heatmap for sample text is presented, green shades highlight the words contributing to the positive sentiment, while red shades highlight the words contributing to the negative sentiment.

For movie review sentiment analysis, the latent space sparseness is defined on the latent vectors generated by a Doc2Vec model [37], which is an unsupervised DL algorithm for representing documents as vectors in a lower-dimensional space. More specifically, Doc2Vec represents each document as a single vector which encapsulates the semantics of the whole document.

Latent space sparseness tackles the issues of high-dimensional and noisy input spaces by focusing on lower-dimensional representations of the relevant information carried by the inputs.

2.3 Archive of Solutions

The archive of solutions stores the best individuals encountered during the search and, at the end of the last search iteration, it contains the final solutions. The archive is particularly useful to prevent the *cycling* phenomenon, i.e., when the search moves from one cell of the feature space to another and back again, with no memory of the cells it has already explored [48].

The UpdateArchive function manages the archive and is described in Algorithm 2. When the archive is not full, all the candidate individuals placed on target or in the neighbouring feature map cells, i.e. those with a distance to the target cell lower than 1, are included into the archive (lines 2-4). Otherwise, if the archive is full, the new candidate input competes with the worst individual in the archive based on their values of $fitness_1$, $fitness_2$ and $fitness_3$. The worst individual in the archive is the one with the highest distance to the target and (for equal distances to the target) the lowest sparseness (line 6). If the candidate individual has lower distance to the target than the worst archived individual, UpdateArchive replaces the former with the latter within the archive (lines 7-9). When the compared inputs have equal distance to the target, the algorithm evaluates their closeness to misbehaviour and keeps in the archive the best one, which is closer to exposing a misbehaviour

Algorithm 2: The UpdateArchive function

```
   Input  : P: population
            archiveSize: target size of the archive A
            targetCell: target feature value ranges
1  foreach p ∈ P do
2      if dist(p, targetCell) ≤ 1 then
3          if A is not full and p ∉ A then
4              A.insert(p) ;
5          else
6              ind ← getWorstIndividual(A);
7              if dist(p, targetCell) < dist(ind, targetCell)
                then
8                  A.insert(p) ;
9                  A.remove(ind) ;
10             else
11                 if dist(p, targetCell) ==
12                 dist(ind, targetCell) then
13                     if p.behaviour < ind.behaviour then
14                         A.insert(p) ;
15                         A.remove(ind) ;
16                     else
17                         if p.behaviour == ind.behaviour &
                            p.spars > ind.spars then
18                             A.insert(p) ;
19                             A.remove(ind) ;
20                         end
21                     end
22                 end
23             end
24         end
25     end
26 end
```

(lines 11-15). If the compared individuals have the same distance to the target and to a misbehaviour, they compete on the basis of their sparseness: the sparser one is kept, while the other is discarded (lines 17-19).

Since the archive may contain correctly-behaving inputs, the FilterMisbehaviours function is performed at the end of the search to keep only misbehaviour-inducing inputs (see Algorithm 1).

2.4 Search Strategies

In this work, we evaluated two different search strategies for DeepAtash: Single-Objective search, which optimizes only the distance to the target, and Multi-Objective search, which explicitly rewards also the closeness to misbehaviour and the sparseness.

2.4.1 Single-Objective Search. As single-objective search strategy, we adopt a Genetic Algorithm (GA) since it previously showed to be very effective for test generation [18]. In particular, we adopt a population-based GA that minimises the Manhattan distance to the target cell. At each iteration, the best individuals in the current population and the offspring are selected, based on their single fitness value, to be part of the next population.

2.4.2 Multi-Objective Search. In this strategy, we cast the focused test generation problem as a multi-objective search problem, by optimising all three fitness functions defined in Section 2.2 at the same time. In particular, we adopt the NSGA-II algorithm [12] since it is widely used and it is reported to be very effective in test case generation [2, 35, 45, 51, 57, 59]. NSGA-II applies Pareto front analysis and promotes the solutions that are not dominated by any other individual, i.e., those representing the best trade-offs among the fitness functions. More precisely, a solution x dominates another solution y if x is not worse than y in all fitness values, and x is strictly better than y in at least one fitness value. The final ranking of individuals is based on Pareto dominance (i.e., non dominated fronts are selected and removed from the solutions one after the other) and crowding distance (i.e., within the same Pareto front, distant individuals are selected), as recommended by the NSGA-II multi-objective optimisation algorithm [12]. While this search strategy comes with some overhead, as it computes multiple fitness functions, dominance and crowding distances, it may improve the archived solutions by explicitly promoting diverse and misbehaviour-inducing inputs.

2.5 Population Management

DeepAtash starts its search from an initial population of size *popsize*, which is obtained by randomly choosing from a pool of inputs, named *seeds*. Function InitialisePopulation (line 2 in Algorithm 1) selects *popsize* different initial individuals among the seeds.

More specifically, for *handwritten digit recognition*, seeds are all the inputs from the MNIST test set. Instead, for *movie review sentiment analysis*, seeds are the inputs in the IMDB test set that are closer than a predefined threshold *maxDist* to the target cell. This choice is due to the large size of the IMDB test set: we consider only the most promising inputs. We determined *maxDist* after some preliminary DeepAtash runs with increasing values of *maxDist* (starting from 0) and selected the minimum value that resulted in a reasonable number of archived solutions (see Table 1). One common issue in search-based approaches is that the exploration could get stuck in local optima, despite the use of mechanisms to promote diversity such as our fitness function in Equation 5. To mitigate this situation and further vary the population, DeepAtash uses the Repopulation operator, which replaces at each iteration a fraction of the worst performing individuals in the current population, i.e., the individuals with the lowest fitness. The new individuals are generated starting from a randomly chosen seed, which is not already represented in the current population and the archive.

This genetic operator can be tuned by setting the *repopulation upperbound* hyperparameter, that determines the range from which the number of individuals to replace is uniformly sampled. As an example, if the repopulation upper bound is set to 100, at each iteration, a number *nrep* is uniformly sampled between 1 and 100, and the *nrep* worst individuals in the current population are replaced by newly generated individuals (see Table 1).

2.6 Mutation

A new input is obtained from an existing one by applying the Mutate operator (line 7 in Algorithm 1). This operator applies a perturbation to the original input by leveraging its semantic (i.e., the digit

model control points or the word's synonyms). For the *handwritten digit recognition* problem, the mutation operator randomly chooses a control point of the SVG model and applies a displacement to it in the two-dimensional space. For the *movie review sentiment analysis* problem, we defined three mutation operators: (1) replacing a word with its synonym obtained from Wordnet[1]; (2) adding an "and" conjunction after an adjective, followed by a synonym of the adjective; and (3) duplicating a sentence.

Each time an input is mutated, DeepAtash verifies that the mutant complies with the ad hoc constraints to ensure that the input still belongs to the input validity domain and preserves its original label [60]. When the mutated individual is considered invalid it is discarded and its parent is mutated again. For the *handwritten digit recognition problem*, DeepAtash verifies that the Euclidean distance between the mutant and the starting seed is greater than 0 and lower than 2. For the *movie review sentiment analysis* problem, the mutated individual is considered invalid if (1) the number of sentiment words differs more than a threshold *sentimentDist* from the initial one and (2) the proportion between positive and negative words is substantially different from the initial one. To validate such heuristic constraints, we manually inspected a set of test inputs produced by DeepAtash and found that in all cases label preservation and validity were confirmed. Therefore, we are confident that misbehavior-inducing inputs are actually producing misbehaviours with high probability. When moving to a different domain, proper heuristic validation functions must be designed for domain-specific mutation operators.

3 EXPERIMENTAL EVALUATION

3.1 Subject Systems

The MNIST system for handwritten digit recognition and the IMDB system for movie review sentiment analysis have been widely used in the literature to evaluate testing techniques for DL systems [58, 74]. The MNIST system solves an image classification problem, i.e., it recognises handwritten digits from the MNIST dataset [38]. It is a Deep Neural Network (DNN) that predicts which digit is represented in a greyscale image. We considered the convolutional DNN architecture provided by Keras [11] and trained it on the the MNIST training set. In particular, we used its default configuration, i.e. 12 epochs, batches of size 128, and a learning rate equal to 0.001, which achieved 99.11% test accuracy. For our experiments, we used three features defined for MNIST digits [76]: (1) *Luminosity (Lum)*: number of light pixels of the image, i.e., pixels whose value is above 127; (2) *Orientation (Or)*: vertical orientation of the digit, obtained by computing the angular coefficient of the linear regression of the non-black pixels; (3) *Moves* (Mov): sum of the Euclidean distances between pairs of consecutive sections of the digit.

The IMDB system solves a text classification problem, as it determines the sentiment of movie reviews from the IMDB dataset [42]. It is a DNN that predicts whether the review has positive or negative sentiment. We used a convolutional DNN architecture with an embedding layer provided by Keras [50], which accepts as input tokenised (with vocabulary size equals to 10K) and padded text with length limited to 2K words. We trained the model on the IMDB training set with 10 epochs, batches of size 32 with early stopping, and the *Adam* optimizer, achieving 88.19% test accuracy. For our experiments, we used three features defined for IMDB movie reviews: (1) *Positive words count (Pos)*: number of words in the text with positive polarity, obtained by counting the words tagged as positive in the English Opinion Lexicon by Liu and Hu [26]; (2) *Negative words count (Neg)*: number of words in the text with negative polarity, obtained by counting the words tagged as negative in the aforementioned lexicon; (3) *Verb count (Verb)*: number of verbs in the text, a proxy for the text complexity, computed by counting the words with a *verb* tag, according to the part-of-speech tagging performed by the NLTK library[2].

3.2 Research Questions

The goal of our evaluation is to understand the effectiveness of our approach in generating misbehaviour-inducing test inputs with the desired features. In particular, we consider different possible configurations of DeepAtash, compare it with an existing state-of-the-art test generator (DeepHyperion), and investigate the usefulness of the generated test inputs. Therefore, we answer the following research questions:

RQ1 (Effectiveness): *Which DeepAtash configuration is the most effective in generating focused test inputs?*

As detailed in Sections 2.2 and 2.4, DeepAtash can be configured with alternative search strategies (single- or multi-objective) and distance metrics (sparseness can be measured on the input, latent or explanation space). This RQ aims at comparing the effectiveness of such six alternative configurations.

RQ2 (Comparison): *How does DeepAtash compare with the state of the art tool DeepHyperion?*

In this RQ, we are interested in whether our focused approach is more effective than DeepHyperion in generating test inputs in proximity of and within the target cell. We compare the best performing DeepAtash configuration (obtained from RQ1) against DeepHyperion, as the latter is the only state-of-the-art test generator that targets the feature space at large by means of an illumination search algorithm. Unlike Active Learning techniques [55] or unguided test generators, DeepHyperion tries to cover all feature combinations and thus it is more likely to produce inputs on the selected target. On the contrary, random techniques produce few or no inputs on the target, making the comparison with DeepAtash impossible.

Actually, to the best of our knowledge, no state of the art DL test generator is a *focused* test generator, capable of targeting a specific region of the feature space. DeepHyperion [77] is a model-based test generator that is applicable to MNIST and IMDB. DeepHyperion explores the feature space using the same input representation and mutation genetic operators as DeepAtash. Therefore, our experimental comparison can effectively rule out all confounding factors and assess the actual contribution of our focused algorithm in isolation.

RQ3 (Usefulness): *Can the test inputs generated by DeepAtash be used to improve the DL system under the test?*

In this RQ, we aim to investigate the usefulness of DeepAtash in a common DL usage scenario. We simulate a scenario in which a dev2op data shift has been observed, i.e., a feature combination is

[1] https://wordnet.princeton.edu

[2] Natural Language Toolkit - https://www.nltk.org

frequently observed during operation, but it was scarcely (or not) represented at development time. A tester can use DeepAtash to target the feature values of interest and fine tune the DL system with the generated tests inputs, in order to improve its quality without introducing regressions.

3.2.1 Metrics: We evaluate the focused test generator's effectiveness by measuring the *Tests Close to the target (TC)* as the number of generated failure-inducing inputs in the proximity of the target feature map's cell, i.e. the solutions in the archive whose distances to the target are lower than or equal to 1. Moreover, we assess the generator's ability to reach the target by computing the number of *Tests on Target (TT)*, i.e., the number of failure-inducing inputs that fall within the boundaries of the target cell.

For a given target feature map cell, we prefer a generator that produces diversified inputs. To evaluate this aspect, we measure test input diversity by introducing the *Tests Close to the target Diversity (TCD)* and *Tests on Target Diversity (TTD)* metrics. To this aim, we represent the generated inputs in a lower dimensional space by using the *t-distributed Stochastic Neighbor Embedding (t-SNE)* algorithm [25, 70], which projects similar inputs to neighbouring points and dissimilar inputs to distant points with high probability. Then, we project the inputs generated by all approaches being compared onto the same t-SNE space and compute the clusters of neighbouring points in such a space. The diversity value of each approach is computed as the number of clusters containing at least one input generated by the corresponding approach divided by the total number of clusters [10]: TCD and TTD measure the relative coverage of the clusters by each approach.

We configured t-SNE by choosing 2 as number of dimensions since it performed well in preliminary runs and it eases the results' interpretation. As regards the t-SNE perplexity (which affects the way inputs are scattered or concentrated), we set it to 0.1 after a visual inspection of the plots obtained with different values. For clustering, we applied the *k-Means* algorithm [6] and performed Silhouette analysis [62] to determine the optimal number of clusters k^*, i.e., the value that better balances between cohesion and separation of the clusters.

Figure 3 exemplifies the diversity comparison between three DeepAtash configurations. The points represent the inputs generated by each configuration in the 2D t-SNE space. Each configuration is assigned a different color. Points are grouped into clusters (represented as circles) and diversity is computed as the number of clusters covered by each configuration. In this example, the diversity value for NSGAII-Input is 0.1; it is 0.5 for NSGAII-Explanation and 0.7 for NSGAII-Latent.

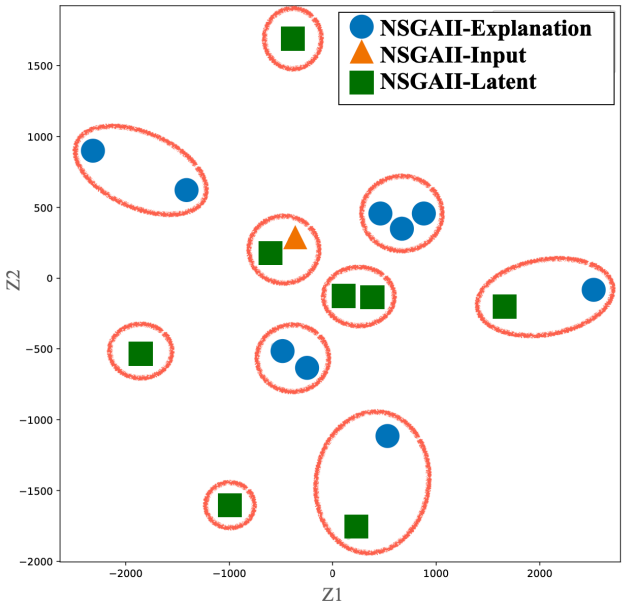


Figure 3: Example t-SNE plot to explain the computation of *test diversity* metrics, with clusters represented as empty circles containing inputs (smaller, solid shapes).

3.3 Evaluation Scenarios

A crucial aspect of focused input generation is the choice of the target cells. Developers can use information from the operation environment to identify feature map cells that occur in operation, but are under-represented in the train/test set. Since we do not have access to operation data, we chose our targets starting from the *misbehaviour probability map* (such as the one in Figure 1), a feature map that encodes the DL failure probability for different feature combinations. More precisely, misbehaviour probability maps are feature maps that report, for each cell, the Average Misbehaviour Probability (AMP) observed in different test executions. AMP is computed as the ratio of the number of misbehaviour-inducing inputs to the total number of inputs in each cell. Since some cells may contain only a small number of inputs, the corresponding AMP values might be affected by a large error. The confidence of AMP can be computed by means of Wilson's confidence interval estimator for binomial random variables, which indicates whether the misbehaviour probability estimation for a given cell suffers from a high error or not. A combination of feature values is considered to produce misbehaviours with high confidence if its AMP is > 0.8 and its confidence interval's lower bound is > 0.65 [76]. In the misbehaviour probability map shown in Figure 1, the darkness level of the cells is proportional to their AMP values; thick borders highlight combinations producing misbehaviours with high confidence, while blank cells correspond to uncovered feature combination values. We leveraged misbehaviour probability maps and AMP to mimick various possible user choices by evaluating DeepAtash in the following scenarios:

Dark Targets: targets selected among the dark, thick-bordered cells (misbehaviour probability > 0.8 and confidence interval > 0.65). These cells correspond to error-prone feature values and mostly contain individuals with high probability of causing misbehaviours. In this scenario, the user wants to collect diverse new inputs with critical characteristics, e.g., to fine tune the DL system;

Grey Targets: targets corresponding to covered cells with misbehaviour probability ≤ 0.8. These cells correspond to feature value combinations for which the DL system generally behaves correctly. Therefore, in this scenario the user wants to stress the DL system with inputs whose characteristics seem not critical, to see if they can possibly trigger any misbehaviour;

Table 1: Hyperparameters used in the experiments

Parameter	MNIST	IMDB
seed pool size	800	1000
population size	100	100
time budget (s)	3600	3600
mutation lower bound	0.01	-
mutation upper bound	0.6	-
sentimentDist	-	5
maxDist	-	5
repopulation upper bound	10	10
target archive size	81	42
number of epochs for retraining	6	5
learning rate for retraining	0.001	0.0001

White Targets: targets corresponding to uncovered cells, i.e., cells that do not contain inputs. In this scenario, the user wants to generate new data with missing feature combinations or check the feasibility of the selected combination of feature values.

Since the usage scenario we are mimicking is one where operation data occur in regions of the feature map that are underpopulated, for dark and grey cells we apply the additional filter that the selected cell must contain a number of individuals lower than the average number of individuals observed across all feature map cells (this filter is not necessary for white cells, which are not populated at all).

3.4 Experimental Procedure

To answer our research questions, we ran DeepAtash in the three evaluation scenarios introduced in Section 3.3 along with DeepHyperion on the subject systems, in all the possible 2D combinations of the proposed features. For each scenario, the first step is the selection of the target cell from the misbehaviour probability maps generated by DeepHyperion.

As regards the dark and grey targets, we chose a cell among the underpopulated dark and grey cells of the misbehaviour probability map. More specifically, we randomly chose a cell for which the coverage (i.e., number of individuals assigned to the cell) achieved by DeepHyperion is lower than the average cell coverage, computed by considering all cells in all DeepHyperion runs. As white targets, we chose uncovered cells in the DeepHyperion misbehaviour probability maps. Since uncovered cells may be unfeasible, we considered white cells in the neighborhood of covered cells.

Then, we ran DeepAtash focusing on the identified target cells and collected the resulting archives of solutions. The hyperparameters of DeepAtash were obtained whenever possible from the experiments conducted with DeepHyperion [76] and were fine tuned in a few preliminary runs to ensure the target cells are reachable with them. The resulting hyperparameter values are reported in Table 1. For DeepHyperion, we used the latest version of the tool along with the configuration reported in the corresponding paper [77].

To facilitate a fair comparison, we used the same initial seeds for the compared tools. The seeds for MNIST were obtained by considering all test set inputs that belong to class "5". For IMDB, the seeds were selected from the test set inputs, all belonging to class "positive". Since we had a huge number (i.e., 12500) of positive reviews in the IMDB test set, we randomly selected an initial seed pool of 1000 inputs among the ones closer to the target (with distance lower than a threshold *maxDist*; see Table 1). For IMDB reviews, we adopted the first validity constraint (see Section 2.6) in our experiments, i.e. the number of sentiment words must not differ more than a threshold *sentimentDist* from the initial one. The second validity constraint is also available in the implementation of DeepAtash, but it was not enabled in our experiments, as the first was sufficient to ensure validity with high probability.

To allow statistical analysis, we ran each approach 10 times for each target, for a total of 240 runs on each subject. To ensure a fair comparison, we ran all tools with the same time budget (1h for MNIST and IMDB). To assess the statistical significance of the comparisons between different DeepAtash configurations (RQ1), and between DeepAtash and DeepHyperion (RQ2), we applied the Mann-Whitney U-test and measured the effect size by means of the Vargha-Delaney's $\hat{A}_{12}$ statistic [5].

To answer RQ3, we fine tuned [8] the original DL models and trained them for additional epochs at the same or lower learning rate (see Table 1) with the inputs generated by DeepAtash close (TC) and within (TT) the considered targets. For each feature combination, we collected the inputs generated by DeepAtash within and close to the targets. Then, we equally divided these inputs into two sets, i.e., $training_{DA}$ and $test_{DA}$. The combination of the original training set and $training_{DA}$ was used to fine tune the DL system, while the original test set and $test_{DA}$ were used to evaluate the accuracy of the fine-tuned DL system. We repeated the fine tuning procedure 10 times for each run of DeepAtash on each target cell, to measure the statistical significance of the accuracy improvement.

4 RESULTS

4.1 RQ1: Effectiveness

Table 2 reports the results achieved on MNIST and IMDB by the 6 alternative configurations of DeepAtash, obtained by combining the search strategies (GA and NSGA-II) and the sparseness metrics (i.e., distance computed in the Input, Latent and Explanation space).

For each evaluation scenario described in Section 3.3, the table reports a row for each feature combination. The columns report the number of failure-inducing inputs generated in the proximity of the target (TC), those of them exactly on target (TT), and their diversity (TCD and TTD, respectively). For each target-feature combination (row), the largest value is highlighted in bold, while the underlined values are comparable among them (p-value ≥ 0.05) and significantly higher than the remaining ones (p-value < 0.05, large or medium effect size), which, when they exist, are not underlined.

As regards MNIST (Table 2 - top), the configuration NSGA-II + Latent produced significantly more and sparser inputs close to the target (TC and TCD) than the other configurations, on average across all the target-feature combinations (39% of them on target). Instead, in terms of TT and TTD, NSGA-II + Latent performed comparably to GA + Latent and GA + Input (p-value > 0.05), and significantly better than the other DeepAtash configurations.

For IMDB (Table 2 - bottom), NSGA-II + Latent performed significantly better than the other configurations, on average across all the target-feature combinations and for all the considered metrics.

Table 2: RQ1 - Tests close to target (TC), tests on target (TT), tests close to target diversity (TCD), and tests on target diversity (TTD) by alternative DeepAtash configurations for MNIST and IMDB. In each row, boldface indicates the maximum; underline indicates values statistically indistinguishable from the maximum.

			Input				Latent				Explanation			
			GA		NSGA-II		GA		NSGA-II		GA		NSGA-II	
		Features	TC [TCD]	TT [TTD]	TC [TCD]	TT [TTD]	TC [TCD]	TT [TTD]	TC [TCD]	TT [TTD]	TC [TCD]	TT [TTD]	TC [TCD]	TT [TTD]
MNIST	Dark	Mov-Lum	41.10 [**0.38**]	41.10 [0.48]	43.10 [0.36]	23.90 [0.12]	**51.70** [0.37]	**51.70** [0.47]	50.50 [0.31]	46.80 [**0.58**]	16.40 [0.24]	1.00 [0.05]	33.60 [0.28]	1.80 [0.25]
		Mov-Or	**69.10** [0.28]	4.40 [0.10]	58.60 [0.22]	1.60 [0.15]	66.10 [**0.30**]	**10.20** [**0.45**]	55.30 [0.22]	1.20 [0.13]	28.50 [0.14]	0.00 [0.00]	26.70 [0.12]	3.00 [0.25]
		Or-Lum	**70.30** [0.27]	24.20 [0.65]	32.10 [0.14]	5.90 [0.35]	65.50 [**0.31**]	**25.90** [**0.70**]	64.30 [0.22]	15.60 [0.50]	55.30 [0.21]	8.60 [0.55]	42.80 [0.18]	5.80 [0.45]
	Grey	Mov-Lum	41.40 [0.60]	**38.30** [**0.73**]	21.80 [0.28]	0.00 [0.00]	22.10 [0.38]	16.20 [0.35]	**53.40** [**0.78**]	28.40 [0.40]	5.10 [0.25]	0.60 [0.08]	13.40 [0.49]	0.40 [0.10]
		Mov-Or	18.50 [0.45]	3.00 [0.20]	15.30 [0.43]	2.40 [0.20]	16.00 [0.24]	0.70 [0.11]	**20.50** [**0.53**]	**4.70** [**0.28**]	13.10 [0.30]	1.80 [0.15]	14.90 [0.49]	0.40 [0.10]
		Or-Lum	10.10 [0.29]	**10.10** [0.34]	22.90 [0.47]	8.60 [0.27]	9.20 [0.23]	9.20 [0.30]	19.20 [0.52]	6.80 [0.19]	6.40 [0.29]	3.50 [0.24]	**28.10** [**0.56**]	6.20 [**0.55**]
	White	Mov-Lum	14.30 [0.36]	**11.60** [0.28]	28.20 [0.61]	2.30 [0.30]	20.60 [0.42]	10.70 [**0.36**]	**29.60** [0.54]	10.40 [0.25]	10.90 [0.38]	5.10 [0.15]	15.40 [0.60]	6.20 [0.55]
		Mov-Or	24.60 [0.44]	2.00 [0.21]	11.30 [0.31]	**7.70** [0.10]	22.10 [0.50]	5.90 [**0.35**]	**25.10** [**0.65**]	1.80 [0.18]	6.70 [0.43]	0.00 [0.00]	7.30 [0.32]	0.00 [0.00]
		Or-Lum	23.30 [0.48]	21.60 [0.58]	30.20 [0.52]	10.10 [0.38]	28.70 [0.51]	24.30 [**0.71**]	**51.00** [**0.66**]	**28.00** [0.65]	21.50 [0.51]	6.80 [0.48]	20.80 [0.52]	4.30 [0.48]
		AVG	34.74 [0.39]	**17.37** [0.40]	29.28 [0.37]	6.94 [0.21]	33.56 [0.36]	17.20 [**0.42**]	**40.99** [**0.49**]	15.97 [0.35]	18.21 [0.31]	3.00 [0.19]	22.56 [0.40]	2.68 [0.26]
IMDB	Dark	Neg-Pos	33.00 [0.68]	22.00 [0.5]	29.40 [0.53]	6.90 [0.34]	25.80 [0.58]	14.30 [0.53]	40.30 [0.74]	33.40 [0.64]	25.70 [0.49]	7.8 [0.31]	29.90 [0.51]	9.60 [0.37]
		Neg-Verb	7.20 [0.31]	3.20 [0.12]	9.20 [0.36]	6.30 [0.16]	5.00 [0.16]	0.70 [0.10]	27.10 [0.63]	14.60 [0.34]	11.20 [0.53]	3.60 [0.15]	19.70 [0.54]	6.60 [0.17]
		Pos-Verb	33.80 [0.53]	33.80 [0.43]	31.60 [0.45]	31.60 [0.47]	33.70 [0.50]	33.40 [0.58]	37.60 [0.52]	37.60 [0.57]	28.80 [0.50]	27.80 [0.38]	28.30 [0.47]	6.60 [0.17]
	Grey	Neg-Pos	**7.50** [0.30]	5.20 [0.35]	6.40 [0.22]	3.50 [0.09]	5.00 [0.25]	3.80 [0.06]	**10.80** [**0.52**]	**8.30** [**0.35**]	0.80 [0.05]	0.00 [0.00]	5.60 [0.34]	0.10 [0.00]
		Neg-Verb	7.30 [0.45]	7.30 [0.41]	15.00 [0.57]	**14.90** [0.61]	10.70 [0.51]	10.70 [0.49]	**15.70** [**0.63**]	13.10 [**0.62**]	9.50 [0.54]	8.60 [0.54]	12.20 [0.45]	11.20 [0.55]
		Pos-Verb	27.10 [0.50]	27.10 [0.60]	28.70 [0.49]	28.70 [0.49]	24.10 [0.62]	27.30 [0.41]	**32.00** [0.53]	**32.00** [0.56]	27.30 [0.41]	27.30 [**0.68**]	25.50 [0.62]	25.50 [0.58]
	White	Neg-Pos	24.20 [0.65]	15.40 [**0.65**]	31.10 [0.62]	22.40 [0.63]	29.40 [0.64]	18.00 [0.53]	**38.90** [0.60]	**29.20** [0.63]	24.30 [0.52]	20.90 [0.58]	26.40 [**0.67**]	14.20 [0.58]
		Neg-Verb	4.10 [0.26]	1.80 [0.10]	0.00 [0.00]	0.00 [0.00]	5.60 [**0.52**]	0.00 [0.00]	**13.00** [0.37]	**3.60** [**0.25**]	0.70 [0.06]	0.00 [0.00]	0.40 [0.02]	0.00 [0.00]
		Pos-Verb	6.10 [0.13]	2.30 [0.10]	25.10 [**0.60**]	**9.40** [0.38]	3.00 [0.10]	1.70 [0.07]	**25.70** [0.48]	3.80 [0.19]	8.40 [0.15]	0.00 [0.00]	15.90 [0.53]	1.30 [0.07]
		AVG	16.70 [0.42]	13.12 [0.36]	19.60 [0.43]	13.70 [0.35]	15.81 [0.43]	11.86 [0.33]	**26.80** [**0.56**]	**19.51** [**0.46**]	19.60 [0.43]	13.70 [0.35]	18.21 [0.46]	10.80 [0.30]

We can observe that in terms of sparseness (metrics TCD, TTD), heatmaps (column "Explanations") perform consistently worse across most configurations, with only a few exceptions. This may be either due to the information that they discard by considering only input elements that contribute to a prediction, or to the heatmap computation itself, which introduces an overhead and hence consumes the overall test generation budget more quickly.

RQ1: *Overall, the multi-objective DeepAtash configuration guided by the sparseness metric computed in the latent space generated a significantly larger number of diverse inputs close to the target than the other tool configurations. For IMDB, this configuration performs significantly better also in terms of tests on target.*

4.2 RQ2: Comparison

Table 3 compares the results achieved on MNIST and IMDB by the best DeepAtash configuration (i.e., NSGA-II - Latent) with the baseline tool DeepHyperion.

For MNIST and IMDB, DeepAtash achieved significantly larger TC, TCD, TT and TTD values than DeepHyperion, on average across all the target-feature combinations. DeepAtash generated up to 29.2 more inputs on targets than DeepHyperion. In white cells where the competitor generated none (e.g., for IMDB, White target, Neg-Pos feature combination), DeepAtash was able to find some inputs with the desired feature combinations. DeepAtash outperformed DeepHyperion for all the target-feature combinations (with statistical significance 80% of the times).

These results confirm that our focused approach can generate inputs in feature space areas where state-of-the-art, general-purpose generator DeepHyperion can generate few or no inputs.

RQ2: *DeepAtash outperforms the state of the art tool DeepHyperion in generating misbehaviour-inducing inputs with target feature value combinations.*

4.3 RQ3: Usefulness

Table 4 shows the accuracy improvement achieved by fine tuning the considered DL systems with $training_{DA}$, the training partition of the inputs generated by DeepAtash. The "before" columns show the accuracy of the original DL systems on the original test set and $test_{DA}$, the test set partition generated by DeepAtash. The "after" columns show the accuracy values achieved after fine tuning the DL systems with $training_{DA}$. All values are underlined, which indicates statistically significant accuracy improvement after the fine tuning (p-value < 0.05, large effect size).

Since we selected state-of-the-art DL systems, their initial accuracy on the original test set was quite high, i.e., 99.11% for MNIST and 88.19% for IMDB. On the other hand, their initial accuracy on $test_{DA}$ was obviously 0% since we considered misbehaviour-inducing inputs generated by DeepAtash.

Quite surprisingly, by fine tuning the considered DL systems using $training_{DA}$, we improved their accuracy on the original test set despite their initial high quality. In fact, for all feature combinations, the accuracy on the original test set significantly increased (up to 1.39% for Neg-Pos). This might be due to an increased generalization capability induced by the additional training on inputs with under-represented feature combinations. So, not only we observed

Table 3: RQ2 - Results achieved by the compared tools for MNIST and IMDB. Tests close to target (TC) and their diversity (TCD); tests on target (TT) and their diversity (TTD). In each row, boldface is the maximum; underline indicates values statistically indistinguishable from the maximum.

		Features	DeepAtash TC [TCD]	DeepAtash TT [TTD]	DeepHyperion TC [TCD]	DeepHyperion TT [TTD]
MNIST	Dark	Mov-Lum	**50.50** [**0.90**]	**46.80** [**0.97**]	18.30 [0.38]	2.30 [0.07]
		Mov-Or	**55.30** [**0.86**]	1.20 [0.27]	37.60 [0.47]	**2.40** [**0.42**]
		Or-Lum	**64.30** [**0.95**]	**15.60** [**0.74**]	13.80 [0.30]	2.70 [0.15]
	Grey	Mov-Lum	**53.40** [**0.81**]	**28.40** [**0.50**]	22.70 [0.50]	3.70 [0.10]
		Mov-Or	20.50 [**0.88**]	**4.70** [**0.45**]	**24.70** [0.45]	1.10 [0.15]
		Or-Lum	**19.20** [**0.81**]	**6.80** [**0.39**]	2.20 [0.19]	0.10 [0.01]
	White	Mov-Lum	**29.60** [**0.74**]	**10.40** [**0.40**]	11.90 [0.42]	0.00 [0.00]
		Mov-Or	**25.10** [**0.71**]	**1.70** [**0.20**]	20.70 [0.50]	0.00 [0.00]
		Or-Lum	**51.00** [**1.00**]	**28.00**[**1.00**]	0.80 [0.05]	0.00 [0.00]
		AVG	**40.99** [**0.85**]	**15.96** [**0.55**]	16.97 [0.36]	1.37 [0.10]
IMDB	Dark	Neg-Pos	**40.30** [**0.94**]	**33.40** [**1.00**]	8.20 [0.11]	1.60 [0.05]
		Neg-Verb	**27.10** [**1.00**]	**14.60** [**0.43**]	10.80 [0.05]	4.50 [0.07]
		Pos-Verb	**32.00** [**0.95**]	**32.00** [**1.00**]	2.40 [0.05]	1.10 [0.05]
	Grey	Neg-Pos	**10.80** [**0.73**]	**8.30** [**0.40**]	10.20 [0.20]	1.00 [0.20]
		Neg-Verb	**15.70** [**0.95**]	**13.10**[**0.93**]	7.70 [0.11]	1.80 [0.08]
		Pos-Verb	**37.60** [**0.95**]	**37.60** [**0.95**]	12.00 [0.15]	5.20 [0.11]
	White	Neg-Pos	**38.90** [**1.00**]	**29.20** [**1.00**]	0.20 [0.00]	0.00 [0.00]
		Neg-Verb	**13.00** [**0.50**]	**3.60** [**0.30**]	0.30 [0.10]	0.00 [0.00]
		Pos-Verb	**25.70** [**0.70**]	**3.50** [**0.30**]	0.70 [0.10]	0.00 [0.00]
		AVG	**26.79** [**0.86**]	**19.48** [**0.70**]	5.83 [0.10]	1.69 [0.06]

Table 4: RQ3 - Model Accuracy (ACC) on the original test set and on the test set generated by DeepAtash, before and after fine tuning the DL system with the training partition of generated inputs. In each row, boldface indicates the maximum; underline indicates values statistically significant.

	Features	Original Test Set ACC before	Original Test Set ACC after	DA Test Set ACC before	DA Test Set ACC after
MNIST	Mov-Lum		**99.23**		**99.92**
	Mov-Or	99.11	**99.24**	0.00	**99.65**
	Or-Lum		**99.23**		**99.02**
IMDB	Neg-Pos		**89.58**		**98.36**
	Neg-Verb	88.19	**89.56**	0.00	**99.47**
	Pos-Verb		**89.56**		**97.35**

no sign of regressions, but we also achieved a slight accuracy improvement on the original test set. As expected, the accuracy on $test_{DA}$ dramatically increased from 0% to at least 97.35%.

> **RQ3**: *DeepAtash is useful to improve the accuracy of a DL system trough fine tuning, by targeting feature combinations under-represented or unseen during development.*

4.4 Threats to Validity

External Validity: The choice of subjects might have threatened the external validity. We chose DL systems widely used in SE research that belong to separate domains. In particular, we chose subjects for which semantic information on the inputs is accessible. In fact, the key requirement for DeepAtash is that a generative model of the inputs exists, such that genetic operators can operate on the generative model's parameters. Therefore, our main limitation is the availability of a generative input model. Generative models are widely used in many domains, such as cyber-physical systems, where the environment is often modeled and simulated. In domains where a model would be prohibitively expensive, like image processing, generative neural networks (e.g., GANs [15, 19]) can be used as an input model. Replication of our experiments on additional case studies would be important to corroborate our findings. Another external validity threat is introduced by the choice of the targets, because results may not generalize to different choices of the target. To mitigate this threat, we chose three types of targets (Dark, Grey, White), corresponding to different usage scenarios.
Conclusion Validity: The stochastic nature of DL systems and search-based approaches may affect the results. Therefore, we ran each experiment multiple times and conducted standard statistical tests to assess the results' significance.
Reproducibility: Our results can be replicated using the replication package and experimental data we provided for DeepAtash.

5 RELATED WORK

While focused test case generation has been extensively investigated for traditional software, no existing work adopted it for DL software. Correspondingly, related works can be organized into focused test generation for traditional software and general (unfocused) test generation for DL based systems.

5.1 Automated and Focused Test Generation

In the SE literature, several approaches have been proposed to automatically generate test inputs to exercise, which often means to cover, the software under test and to possibly reveal its faults [73]. Among these approaches, search-based ones proved to be effective and efficient for problems with complex input spaces, for which an exhaustive or symbolic analysis would be impractical [46].

Shamshiri et al. [64] conducted an empirical study showing that state-of-the-art test generators achieve low fault detection rate, despite being able to extensively cover the program code. One of the reasons for this issue was that covering faulty code is often not enough to trigger a failure, because it is also necessary to exercise the program with specific inputs. Focused test generation approaches can address this issue by generating diverse inputs that target a specific part of the software under test.

Alipour et al. [4] propose *directed swarm testing*, which randomly generates tests that have increased probability of covering selected source code targets. Gotlieb and Petit [21] exploit a constraint solver to cover a specific control flow path of the program under test. The Diversified Focused Testing approach by Menendez et al. [47] combines model checking and search-based testing to generate inputs that reach specific program points, while promoting input diversity. IFRIT by Romdhana et al. [61] exploits Reinforcement Learning (RL) to generate diverse test inputs for a focused point of the program under test. IFRIT's RL agent rewards the generated inputs if they are diverse from the previously generated solutions and they are able to reach the target.

The approaches listed above were proposed for traditional software. In this work, we adapt search-based focused generation of

diverse test inputs to target a specific region of the feature space of DL systems.

5.2 Test Generation for DL Based Systems

Traditional code coverage adequacy criteria, such as branch coverage, cannot assess whether DL systems are adequately exercised by a test set and, thus, cannot effectively drive test generation. In fact, the DL systems' behaviour mostly depends on their training data, rather than their code. Therefore, test generation approaches for DL systems introduced adequacy metrics designed to take into account DL-specific characteristics.

Pei et al. [52] proposed DeepXplore, a test generator guided by neuron coverage, i.e., the number of activated neurons. In particular, a neuron is considered activated if its output value is higher than a predefined threshold. Several approaches extended the neuron coverage adequacy metric [41] or used neuron coverage metrics to guide test generation [13, 14, 23, 41, 67, 72]. Lei et al. [40], proposed DeepCT, which is guided, instead, by combinatorial criteria that consider the interactions between neurons. These neuron-based testing approaches are focused on covering specific sets of neurons or model layers. However, empirical results showed that higher neuron coverage does not correlate with a higher number of detected failures and leads to the generation of less natural inputs [24].

Kim et al. [31–34] designed surprise adequacy criteria, based on the degree of "surprise" of an input for the neural network, i.e., the novelty of a test input with respect to the training data. In particular, the authors define k buckets of consecutive surprise ranges that must be covered by the considered test set. Zhang et al. [75] studied the distribution of test inputs on different uncertainty patterns, i.e. combinations of alternative uncertainty metrics such as prediction confidence and variation ratio. They did not explicitly define test adequacy metrics, but they recommended to generate additional test inputs to cover the least covered uncertainty patterns. DeepAtash focuses test generation on input features rather than on surprise values or uncertainty patterns, since the latter are not easily interpretable by humans and do not provide hints on the characteristics of the inputs that make the system fail.

Other works adapted mutation adequacy criteria to DL [27, 29, 30]. DeepMetis [57] is a testing approach to increase the mutation killing ability of a test set, by generating test inputs that behave correctly on original DL models and misbehave on mutants. Vahdat Pour el al. [54] proposed an approach to generate adversarial inputs for source code processing DNNs, guided by the mutation killing criterion defined by Hu et al. [27]. No study investigated the interplay between mutation adequacy and human-interpretable input features (such as the ones used by DeepAtash), which is worth to be investigated in future work.

NSGAII-DT [2] is a test generator for vision-based control systems that uses decision trees to guide its evolutionary algorithm towards input space areas which are likely to cause misbehaviours. While DeepAtash targets predefined input features' values and promotes input diversity, NSGAII-DT focuses on the most critical feature values' combinations learned during the exploration. HUDD [16] is a technique that automatically identifies root causes of DNN failures by clustering the heatmaps of test inputs, so as to group together inputs having common characteristics.

SEDE [17] leverages search based algorithms to generate diverse inputs with predefined critical features, mapped to one or more clusters identified by HUDD. Differently from HUDD and SEDE, we exploit higher-level input explanations (i.e., feature maps), rather than low-level ones (i.e., heatmaps).

Active Learning (AL) techniques sample candidate inputs from an unlabeled dataset according to metrics like uncertainty or diversity [55], without targeting any specific feature combination. Hence, AL may cover a specific region of the feature space just by chance. We instead go one step ahead, as DeepAtash generates misbehaviour-inducing inputs with predefined feature combinations. Indeed, software engineers may know which feature combinations represent critical in-field scenarios that are important to test. Instead, AL is driven only by uncertainty or diversity metrics.

6 CONCLUSIONS AND FUTURE WORK

DeepAtash is the first automated focused test generator for DL systems. Our experiments show that it outperforms the state of the art in generating diverse misbehaviour-inducing inputs on predefined targets. Results show also that fine tuning a DNN on underrepresented inputs produced by DeepAtash not only increases its prediction accuracy on them, but also its generalization ability on the whole input distribution.

In our future work, we plan to generalise our results to additional DL systems, including industrial ones. In particular, we will apply DeepAtash to domains where inputs are too complex to be abstracted into a model-based representation, e.g., by integrating GAN-based input representations into our approach.

ACKNOWLEDGMENTS

This work was partially supported by the H2020 project PRECRIME, funded under the ERC Advanced Grant 2017 Program (ERC Grant Agreement n. 787703).

REFERENCES

[1] Raja Ben Abdessalem, Shiva Nejati, Lionel C. Briand, and Thomas Stifter. 2016. Testing advanced driver assistance systems using multi-objective search and neural networks. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*. ACM, 63–74. https://doi.org/10.1145/2970276.2970311

[2] Raja Ben Abdessalem, Shiva Nejati, Lionel C. Briand, and Thomas Stifter. 2018. Testing Vision-based Control Systems Using Learnable Evolutionary Algorithms. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM, Gothenburg, Sweden, 1016–1026. https://doi.org/10.1145/3180155.3180160

[3] Raja Ben Abdessalem, Annibale Panichella, Shiva Nejati, Lionel C. Briand, and Thomas Stifter. 2018. Testing Autonomous Cars for Feature Interaction Failures Using Many-objective Search. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018)*. ACM, Montpellier, France, 143–154. https://doi.org/10.1145/3238147.3238192

[4] Mohammad Amin Alipour, Alex Groce, Rahul Gopinath, and Arpit Christi. 2016. Generating focused random tests using directed swarm testing. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 70–81.

[5] Andrea Arcuri and Lionel Briand. 2014. A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability* 24, 3 (2014), 219–250. https://doi.org/10.1002/stvr.1486

[6] David Arthur and Sergei Vassilvitskii. 2006. *k-means++: The advantages of careful seeding*. Technical Report. Stanford.

[7] Richard Bellman. 1966. Dynamic Programming. *Science* 153, 3731 (1966), 34–37. https://doi.org/10.1126/science.153.3731.34 arXiv:https://www.science.org/doi/pdf/10.1126/science.153.3731.34

[8] Yoshua Bengio. 2011. Deep Learning of Representations for Unsupervised and Transfer Learning. In *Proceedings of the 2011 International Conference on Unsupervised and Transfer Learning Workshop - Volume 27 (UTLW'11)*. JMLR.org,

Washington, USA, 17–37.
[9] Matteo Biagiola, Stefan Klikovits, Jarkko Peltomaki, and Vincenzo Riccio. 2023. SBFT Tool Competition 2023 - Cyber-Physical Systems Track. In *16th IEEE/ACM International Workshop on Search-Based And Fuzz Testing, SBFT 2023, Melbourne, Australia, May 14, 2023*.
[10] Matteo Biagiola and Paolo Tonella. 2023. Testing of Deep Reinforcement Learning Agents with Surrogate Models. *arXiv preprint arXiv:2305.12751* (2023).
[11] François Chollet. 2020. Simple MNIST convnet. https://github.com/keras-team/keras-io/blob/master/examples/vision/mnist_convnet.py.
[12] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE transactions on evolutionary computation* 6, 2 (2002), 182–197.
[13] Samet Demir, Hasan Ferit Eniser, and Alper Sen. 2020. DeepSmartFuzzer: Reward Guided Test Generation For Deep Learning. In *Proceedings of the Workshop on Artificial Intelligence Safety 2020 (IJCAI-PRICAI 2020), Yokohama, Japan, January, 2021 (CEUR Workshop Proceedings, Vol. 2640)*. CEUR-WS.org, 134–140. http://ceur-ws.org/Vol-2640/paper_19.pdf
[14] Swaroopa Dola, Matthew B. Dwyer, and Mary Lou Soffa. 2021. Distribution-Aware Testing of Neural Networks Using Generative Models. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 226–237. https://doi.org/10.1109/ICSE43902.2021.00032
[15] Isaac Dunn, Hadrien Pouget, Daniel Kroening, and Tom Melham. 2021. Exposing Previously Undetectable Faults in Deep Neural Networks. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 56–66. https://doi.org/10.1145/3460319.3464801
[16] Hazem Fahmy, Fabrizio Pastore, Mojtaba Bagherzadeh, and Lionel Briand. 2021. Supporting Deep Neural Network Safety Analysis and Retraining Through Heatmap-Based Unsupervised Learning. *IEEE Transactions on Reliability* 70, 4 (2021), 1641–1657.
[17] Hazem Fahmy, Fabrizio Pastore, and Lionel Briand. 2022. Simulator-based explanation and debugging of hazard-triggering events in DNN-based safety-critical systems. *arXiv preprint arXiv:2204.00480* (2022).
[18] Gordon Fraser and Andrea Arcuri. 2011. Evolutionary generation of whole test suites. In *2011 11th International Conference on Quality Software*. IEEE, 31–40.
[19] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2020. Generative Adversarial Networks. *Commun. ACM* 63, 11 (oct 2020), 139–144. https://doi.org/10.1145/3422622
[20] Ian J. Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press. http://www.deeplearningbook.org.
[21] Arnaud Gotlieb and Matthieu Petit. 2010. A uniform random test data generator for path testing. *Journal of Systems and Software* 83, 12 (2010), 2618–2626.
[22] Antonio Guerriero, Roberto Pietrantuono, and Stefano Russo. 2021. Operation is the hardest teacher: estimating DNN accuracy looking for mispredictions. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 348–358.
[23] Jianmin Guo, Yu Jiang, Yue Zhao, Quan Chen, and Jiaguang Sun. 2018. DLFuzz: differential fuzzing testing of deep learning systems. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*. ACM, 739–743. https://doi.org/10.1145/3236024.3264835
[24] Fabrice Harel-Canada, Lingxiao Wang, Muhammad Ali Gulzar, Quanquan Gu, and Miryung Kim. 2020. *Is Neuron Coverage a Meaningful Measure for Testing Deep Neural Networks?* Association for Computing Machinery, New York, NY, USA, 851–862. https://doi.org/10.1145/3368089.3409754
[25] Geoffrey E Hinton and Sam Roweis. 2002. Stochastic neighbor embedding. *Advances in neural information processing systems* 15 (2002).
[26] Minqing Hu and Bing Liu. 2004. Opinion Lexicon. https://www.cs.uic.edu/~liub/FBS/sentiment-analysis.html.
[27] Qiang Hu, Lei Ma, Xiaofei Xie, Bing Yu, Yang Liu, and Jianjun Zhao. 2019. DeepMutation++: A Mutation Testing Framework for Deep Learning Systems. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1158–1161. https://doi.org/10.1109/ASE.2019.00126
[28] Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. 2020. Taxonomy of Real Faults in Deep Learning Systems. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20)*. Association for Computing Machinery, Seoul, South Korea, 1110–1121. https://doi.org/10.1145/3377811.3380395
[29] Nargiz Humbatova, Gunel Jahangirova, and Paolo Tonella. 2021. DeepCrime: Mutation Testing of Deep Learning Systems based on Real Faults. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*.
[30] Gunel Jahangirova and Paolo Tonella. 2020. An Empirical Evaluation of Mutation Operators for Deep Learning Systems. In *IEEE International Conference on Software Testing, Verification and Validation (ICST'20)*. IEEE, 12 pages.
[31] Jinhan Kim, Robert Feldt, and Shin Yoo. 2019. Guiding deep learning system testing using surprise adequacy. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. IEEE / ACM, 1039–1049. https://doi.org/10.1109/ICSE.2019.00108
[32] Jinhan Kim, Jeongil Ju, Robert Feldt, and Shin Yoo. 2020. *Reducing DNN Labelling Cost Using Surprise Adequacy: An Industrial Case Study for Autonomous Driving*. Association for Computing Machinery, New York, NY, USA, 1466–1476. https://doi.org/10.1145/3368089.3417065
[33] Seah Kim and Shin Yoo. 2020. *Evaluating Surprise Adequacy for Question Answering*. Association for Computing Machinery, New York, NY, USA, 197–202. https://doi.org/10.1145/3387940.3391465
[34] Seah Kim and Shin Yoo. 2021. Multimodal Surprise Adequacy Analysis of Inputs for Natural Language Processing DNN Models. In *2021 IEEE/ACM International Conference on Automation of Software Test (AST)*. https://doi.org/10.1109/AST52587.2021.00017
[35] Kiran Lakhotia, Mark Harman, and Phil McMinn. 2007. A Multi-objective Approach to Search-based Test Data Generation. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation (GECCO '07)*. ACM, London, England, 1098–1105. https://doi.org/10.1145/1276958.1277175
[36] Craig Larman. 1997. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*. Prentice Hall.
[37] Quoc Le and Tomas Mikolov. 2014. Distributed representations of sentences and documents. In *International conference on machine learning*. PMLR, 1188–1196.
[38] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.
[39] Vladimir I Levenshtein et al. 1966. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, Vol. 10. Soviet Union, 707–710.
[40] Lei Ma, Felix Juefei-Xu, Minhui Xue, Bo Li, Li Li, Yang Liu, and Jianjun Zhao. 2019. DeepCT: Tomographic Combinatorial Testing for Deep Learning Systems. In *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019*. IEEE, 614–618. https://doi.org/10.1109/SANER.2019.8668044
[41] Lei Ma, Felix Juefei-Xu, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Chunyang Chen, Ting Su, Li Li, Yang Liu, Jianjun Zhao, and Yadong Wang. 2018. DeepGauge: Multi-granularity Testing Criteria for Deep Learning Systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018)*. ACM, Montpellier, France, 120–131. https://doi.org/10.1145/3238147.3238202
[42] Andrew Maas, Raymond E Daly, Peter T Pham, Dan Huang, Andrew Y Ng, and Christopher Potts. 2011. Learning word vectors for sentiment analysis. In *Proceedings of the 49th annual meeting of the association for computational linguistics: Human language technologies*. 142–150.
[43] Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. 2011. Learning Word Vectors for Sentiment Analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics, Portland, Oregon, USA, 142–150. http://www.aclweb.org/anthology/P11-1015
[44] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. 2008. *Introduction to Information Retrieval*. Cambridge University Press.
[45] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective Automated Testing for Android Applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*. ACM, Saarbrücken, Germany, 94–105. https://doi.org/10.1145/2931037.2931054
[46] Phil McMinn. 2004. Search-based software test data generation: a survey. *Software testing, Verification and reliability* 14, 2 (2004), 105–156.
[47] Héctor D Menéndez, Gunel Jahangirova, Federica Sarro, Paolo Tonella, and David Clark. 2021. Diversifying focused testing for unit testing. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 4 (2021), 1–24.
[48] Jean-Baptiste Mouret and Jeff Clune. 2015. Illuminating search spaces by mapping elites. arXiv:1504.04909 [cs.AI]
[49] Vuong Nguyen, Stefan Huber, and Alessio Gambi. 2021. SALVO: Automated Generation of Diversified Tests for Self-driving Cars from Existing Maps. In *2021 IEEE International Conference on Artificial Intelligence Testing (AITest)*. IEEE, 128–135.
[50] Mark Omernick and François Chollet. 2020. Text classification from scratch. https://github.com/keras-team/keras-io/blob/master/examples/nlp/text_classification_from_scratch.py.
[51] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2018. Automated Test Case Generation as a Many-Objective Optimisation Problem with Dynamic Selection of the Targets. *IEEE Transactions on Software Engineering* 44, 2 (2018), 122–158.
[52] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2019. DeepXplore: Automated Whitebox Testing of Deep Learning Systems. *Commun. ACM* 62, 11 (Oct. 2019), 137?145. https://doi.org/10.1145/3361566
[53] Eva Pettersson, Beáta Megyesi, and Joakim Nivre. 2013. Normalisation of historical text using context-sensitive weighted Levenshtein distance and compound splitting. In *Proceedings of the 19th Nordic conference of computational linguistics (Nodalida 2013)*. 163–179.

[54] Maryam Vahdat Pour, Zhuo Li, Lei Ma, and Hadi Hemmati. 2021. A Search-Based Testing Framework for Deep Neural Networks of Source Code Embedding. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 36–46.

[55] Pengzhen Ren, Yun Xiao, Xiaojun Chang, Po-Yao Huang, Zhihui Li, Brij B Gupta, Xiaojiang Chen, and Xin Wang. 2021. A survey of deep active learning. *ACM computing surveys (CSUR)* 54, 9 (2021), 1–40.

[56] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. "Why Should I Trust You?": Explaining the Predictions of Any Classifier. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '16)*. Association for Computing Machinery, New York, NY, USA, 1135–1144. https://doi.org/10.1145/2939672.2939778

[57] V. Riccio, N. Humbatova, G. Jahangirova, and P. Tonella. 2021. DeepMetis: Augmenting a Deep Learning Test Set to Increase its Mutation Score. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, Los Alamitos, CA, USA, 355–367. https://doi.org/10.1109/ASE51524.2021.9678764

[58] Vincenzo Riccio, Gunel Jahangirova, Andrea Stocco, Nargiz Humbatova, Michael Weiss, and Paolo Tonella. 2020. Testing machine learning based systems: a systematic mapping. *Empir. Softw. Eng.* 25, 6 (2020), 5193–5254. https://doi.org/10.1007/s10664-020-09881-0

[59] Vincenzo Riccio and Paolo Tonella. 2020. Model-based Exploration of the Frontier of Behaviours for Deep Learning System Testing. In *Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20)*. Association for Computing Machinery, 13 pages. https://doi.org/10.1145/3368089.3409730

[60] Vincenzo Riccio and Paolo Tonella. 2023. When and Why Test Generators for Deep Learning Produce Invalid Inputs: an Empirical Study. In *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE '23)*. IEEE/ACM.

[61] Andrea Romdhana, Mariano Ceccato, Alessio Merlo, and Paolo Tonella. 2022. IFRIT: Focused Testing through Deep Reinforcement Learning. In *IEEE International Conference on Software Testing, Verification and Validation (ICST'22)*. IEEE.

[62] Peter J Rousseeuw. 1987. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of computational and applied mathematics* 20 (1987), 53–65.

[63] P. Selinger. 2003. Potrace: a polygon-based tracing algorithm. http://potrace.sourceforge.net/potrace.pdf

[64] Sina Shamshiri, René Just, José Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. 2015. Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of Effectiveness and Challenges (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 201–211. https://doi.org/10.1109/ASE.2015.86

[65] Mukund Sundararajan, Ankur Taly, and Qiqi Yan. 2017. Axiomatic attribution for deep networks. In *International conference on machine learning*. PMLR, 3319–3328.

[66] Chakkrit Tantithamthavorn and Jirayus Jiarpakdee. 2021. . Monash University. https://doi.org/10.5281/zenodo.4769127 Retrieved 2021-05-17.

[67] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. DeepTest: Automated Testing of Deep-neural-network-driven Autonomous Cars. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM, Gothenburg, Sweden, 303–314. https://doi.org/10.1145/3180155.3180220

[68] Erico Tjoa and Cuntai Guan. 2020. A survey on explainable artificial intelligence (xai): Toward medical xai. *IEEE transactions on neural networks and learning systems* 32, 11 (2020), 4793–4813.

[69] Mark Utting, Alexander Pretschner, and Bruno Legeard. 2012. A taxonomy of model-based testing approaches. *Software testing, verification and reliability* 22, 5 (2012), 297–312.

[70] Laurens Van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-SNE. *Journal of machine learning research* 9, 11 (2008).

[71] Jiapeng Wang and Yihong Dong. 2020. Measurement of text similarity: a survey. *Information* 11, 9 (2020), 421.

[72] Xiaofei Xie, Lei Ma, Felix Juefei-Xu, Minhui Xue, Hongxu Chen, Yang Liu, Jianjun Zhao, Bo Li, Jianxiong Yin, and Simon See. 2019. DeepHunter: A Coverage-Guided Fuzz Testing Framework for Deep Neural Networks. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2019)*. Association for Computing Machinery, Beijing, China, 146–157. https://doi.org/10.1145/3293882.3330579

[73] Qian Yang, J Jenny Li, and David M Weiss. 2009. A survey of coverage-based testing tools. *Comput. J.* 52, 5 (2009), 589–597.

[74] J. M. Zhang, M. Harman, L. Ma, and Y. Liu. 2020. Machine Learning Testing: Survey, Landscapes and Horizons. *IEEE Transactions on Software Engineering* Early Access, – (2020), 1–1. https://doi.org/10.1109/TSE.2019.2962027

[75] Xiyue Zhang, Xiaofei Xie, Lei Ma, Xiaoning Du, Qiang Hu, Yang Liu, Jianjun Zhao, and Sun Meng. 2020. Towards Characterizing Adversarial Defects of Deep Learning Software from the Lens of Uncertainty. In *Proceedings of 42nd International Conference on Software Engineering (ICSE '20)*. ACM, 12 pages.

[76] Tahereh Zohdinasab, Vincenzo Riccio, Alessio Gambi, and Paolo Tonella. 2021. Deephyperion: exploring the feature space of deep learning-based systems through illumination search. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. Virtual, Denmark, 79–90.

[77] Tahereh Zohdinasab, Vincenzo Riccio, Alessio Gambi, and Paolo Tonella. 2022. Efficient and Effective Feature Space Exploration for Testing Deep Learning Systems. *ACM Trans. Softw. Eng. Methodol.* (jun 2022). https://doi.org/10.1145/3544792 Just Accepted.

Received 2023-02-16; accepted 2023-05-03