Efficient and Effective Feature Space Exploration for Testing Deep Learning Systems

TAHERE H ZOHDINASAB*, Università della Svizzera Italiana, Switzerland
VINCENZO RICCIO, Università della Svizzera Italiana, Switzerland
ALESSIO GAMBI, University of Passau, Germany
PAOLO TONELLA, Università della Svizzera Italiana, Switzerland

Assessing the quality of Deep Learning (DL) systems is crucial, as they are increasingly adopted in safety-critical domains. Researchers have proposed several input generation techniques for DL systems. While such techniques can expose failures, they do not explain which features of the test inputs influenced the system's (mis-) behaviour. DeepHyperion was the first test generator to overcome this limitation by exploring the DL systems' feature space at large. In this paper, we propose DeepHyperion-CS, a test generator for DL systems which enhances DeepHyperion by promoting the inputs that contributed more to feature space exploration during the previous search iterations. We performed an empirical study involving two different test subjects (i.e., a digit classifier and a lane-keeping system for self-driving cars). Our results proved that the contribution-based guidance implemented within DeepHyperion-CS outperforms state-of-the-art tools and significantly improves the efficiency and the effectiveness of DeepHyperion. DeepHyperion-CS exposed significantly more misbehaviours for 5 out of 6 feature combinations and was up to 65% more efficient than DeepHyperion in finding misbehaviour-inducing inputs and exploring the feature space. DeepHyperion-CS was useful for expanding the datasets used to train the DL systems, populating up to 200% more feature map cells than the original training set.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: software testing, deep learning, search based software engineering, self-driving cars

ACM Reference Format:
Tahereh Zohdinasab, Vincenzo Riccio, Alessio Gambi, and Paolo Tonella. 2021. Efficient and Effective Feature Space Exploration for Testing Deep Learning Systems. 1, 1 (May 2021), 38 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Using Deep Learning (DL) has become widespread for modern software systems that must process complex inputs and timely solve challenging tasks. For example, image classifiers [31, 62] can analyse images to diagnose diseases, while intelligent driving agents use sensor information (e.g., from cameras and LiDARs) to drive vehicles [10]. Since DL systems are applied also in safety-critical domains, ensuring their dependability is *literally* vital.

Authors' addresses: Tahereh Zohdinasab, tahereh.zohdinasab@usi.ch, Università della Svizzera Italiana, Lugano, Switzerland, 6900; Vincenzo Riccio, Università della Svizzera Italiana, Lugano, Switzerland, vincenzo.riccio@usi.ch; Alessio Gambi, University of Passau, Passau, Germany, alessio.gambi@uni-passau.de; Paolo Tonella, Università della Svizzera Italiana, Lugano, Switzerland, paolo.tonella@usi.ch.

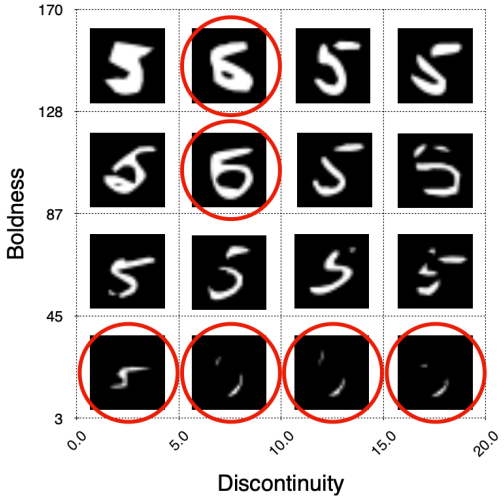


Fig. 1. Feature map for a handwritten digit classifier. The two axes quantify the *discontinuity* and *boldness* of digits. Circled cells highlight misclassified inputs.

Fig. 2. Feature map for a lane keeping system. The two axes quantify the *complexity* and *smoothness* of virtual roads. Circled cells highlight inputs in which the driving agent went astray.

Unlike traditional software, DL systems' behaviour is not explicitly coded, being instead indirectly learned from training examples [49]. This fundamental difference of DL systems from traditional software has profound implications on how their quality is assessed.

In fact, DL systems' source code analysis does not allow to predict their runtime behaviour. On the other hand, it is difficult to understand to what extent DL systems can be trusted once deployed in the real world, as they could process inputs that might be not sufficiently represented in the data used to train them. Most importantly, when such underrepresented inputs are discovered by testing techniques, test results should be interpretable, as developers need to understand which characteristics of the test inputs might have caused a system's misbehaviour (e.g., which features of an input image make the system wrongly classify it or which features of a driving scenario make the system drive the autonomous vehicle off the road). In this manuscript, the term *feature* indicates high-level and human-interpretable characteristics of test inputs, as done in Evolutionary Computation and Robotics [52]. In particular, we consider both structural features (i.e., characteristics of the input itself) and behavioural features (i.e., characteristics of the output of the DL system when exercised by the input).

Several test generation approaches have been proposed for automatically testing DL systems [58, 69]. Some of them aim to pragmatically expose the highest number of misbehaviours [1, 22, 70]. Other approaches, instead, are guided by *ad-hoc* test adequacy metrics, such as neuron coverage [26, 54, 64, 68] or surprise coverage [38], since traditional code coverage metrics fail to measure whether DL systems have been adequately tested. These approaches are effective in triggering multiple misbehaviours, but their output cannot be directly used to explain the behaviour of the DL system under test. For instance, using neuron coverage reports, developers cannot easily understand why the DL system did not handle correctly the misbehaviour-inducing inputs.

DeepHyperion [73] addresses this limitation by offering an interpretable characterisation of DL systems' behaviours.

Its output consists of *feature maps* representing the generated inputs along with their performance (i.e., closeness to exposing a misbehaviour), in the space of the relevant, domain- and problem-specific structural and behavioural features (i.e., the feature space). As an example, for testing handwritten digit classifiers, DeepHyperion can use features like the number of disconnected segments within each digit and the boldness of the stroke.

Feature maps are N-dimensional grids, in which each axis corresponds to a considered feature. These maps are discretised so that each of their cells correspond to an interval of features' values. Test inputs are automatically assigned to a map cell, computed by measuring the metric that quantifies each feature. Figure 1 illustrates a 2-dimensional feature map for a handwritten digit classifier, where the x-axis corresponds to the Discontinuity feature, while the y-axis corresponds to digits' Boldness. Each feature's value range is discretised into 4 intervals, resulting in a 4×4 map.

The resulting feature map highlights that the classifier under test cannot correctly handle thin strokes (i.e., the bottom row of the feature map), as well as bold strokes with moderate discontinuity. Similarly, Figure 2 shows that the driving agent under test is in trouble on roads with sharp curves (low values of Smoothness) regardless of their Complexity, where road Complexity is measured as the number of times the road changes direction significantly. DeepHyperion [73] comes with a systematic methodology to identify and quantify the feature dimensions in the domain of interest.

DeepHyperion explores the DL system's feature space at large and triggers diverse misbehaviours by means of a Multi-dimensional Archive of Phenotypic Elites (MAP-Elites), using an implementation of the illumination search algorithm proposed by Mouret and Clune [52]. At each iteration of the search process, DeepHyperion randomly selects one input from the feature map, mutates it and decides whether it is good enough to be placed on the map, i.e., it covers unseen feature combinations or achieves a better performance with respect to similar, previously generated inputs.

In this work, we propose DeepHyperion-CS, a test generator for DL systems which enhances DeepHyperion by promoting the inputs that contributed more to the feature map exploration during the search process. In particular, DeepHyperion-CS features a novel selection operator which chooses with higher probability inputs that have a higher *Contribution Score (CS)*, i.e., those that in previous iterations generated new inputs which were successfully placed in the feature map.

We evaluated DeepHyperion-CS in two different application domains: recognition of handwritten digits from the MNIST (Modified National Institute of Standards and Technology) database [42], which is a classification problem, and steering angle prediction for self-driving in the BeamNG driving simulator [7], which is a regression problem.

Our empirical study consists of: (1) an evaluation of DeepHyperion-CS's effectiveness and efficiency; (2) a comparison with state-of-the-art test generators, including AsFault [22]; and (3) white-box assessment of the intermediate test inputs produced by test generators, in addition to the black-box assessment of the final outputs.

Our empirical results show that the contribution-based guidance implemented within DeepHyperion-CS outperforms the state of the art and significantly improves the efficiency and the effectiveness of DeepHyperion in finding misbehaviour-inducing inputs and exploring the feature space. Moreover, we also showed that DeepHyperion-CS can also help DL developers by characterising the deficiencies of the DL systems' training dataset and by providing new data to expand it.

In comparison to the original paper describing DeepHyperion [73], the main extensions that can be found in this paper are:

- Contribution Score (CS), a novel metric to select the candidate inputs that are more likely to increase the exploration of the feature space;
- DeepHyperion-CS, a tool that implements feature space exploration based on the guidance offered by CS;

- A large empirical study which shows that DeepHyperion-CS's contribution-based guidance outperforms DeepHyperion and other state-of-the-art test generators in exploring the feature space and exposing misbehaviours. The new empirical study considers both a black-box scenario, in which only the final outputs of the DL test generation tools are available, and a white-box scenario, in which the inputs generated during the search process are also available and can be projected onto the resulting feature map. Moreover, we introduced an additional metric to compute the diversity of the generated inputs.
- On the self-driving car case study, an empirical comparison with an important baseline, AsFault [22];
- An empirical assessment of the efficiency of DeepHyperion-CS, which complements the assessment of its effectiveness. Efficiency is measured by tracking performance indicators (i.e., mapped misbehaviours and filled cells) over time and measuring the area under the curve (AUC).
- An experiment which shows the usefulness of DeepHyperion-CS for a practical DL development task, i.e., training set expansion.

We believe in the fundamental importance of open research and reproducible results [23], therefore we release the code of DeepHyperion-CS, the dataset, and all the scripts to replicate the experimental evaluation at:

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

2 DEFINITION OF THE FEATURES

A crucial element of our approach is the choice of the *features* that DeepHyperion-CS uses to drive test generation. These features capture the dimensions along which the automatically generated test inputs or the system's behaviours may vary. They should represent meaningful properties of the system under test, defining both the search space of DeepHyperion-CS and the feature space of interest to the users [52]. Additionally, features should be discriminative, interpretable, and quantifiable to be useful for automated testing of DL systems. We distinguish two types of features, the ones that characterise *structural features* of the test inputs, and the ones that characterise the output of the DL systems under test, i.e. *behavioural features*.

In our previous work [73], we proposed a systematic methodology that developers can follow to identify possible features in new domains of investigation with the aid of domain experts. We followed that procedure to identify the features in two application domains, i.e., digit recognition and autonomous driving. Noticeably, we considered the same application domains also in the experimental evaluation of DeepHyperion-CS. Therefore, to make this paper self-contained, we briefly summarise this methodology and exemplify its application in the two considered domains.

As Figure 3 illustrates, feature selection consists of two macro-steps: (i) *Open Coding* [60], which aims to select the features that better characterise the generated tests, and (ii) *Metric Identification*, which aims to design procedures that quantify the selected features. The former enables developers to identify a set of features capturing independent variables describing the tests, while the latter allows DeepHyperion-CS to position the tests in the feature map according to the values of their features.

2.1 Open Coding

During Open Coding, human assessors manually analysed a set of existing inputs to select the relevant features in a given domain. Assessors independently tagged the inputs assigned to them with a feature label. Each feature label is composed of a feature name, paired with the corresponding feature value, chosen from a rating scale with five levels, e.g., ranging from -2 to +2.

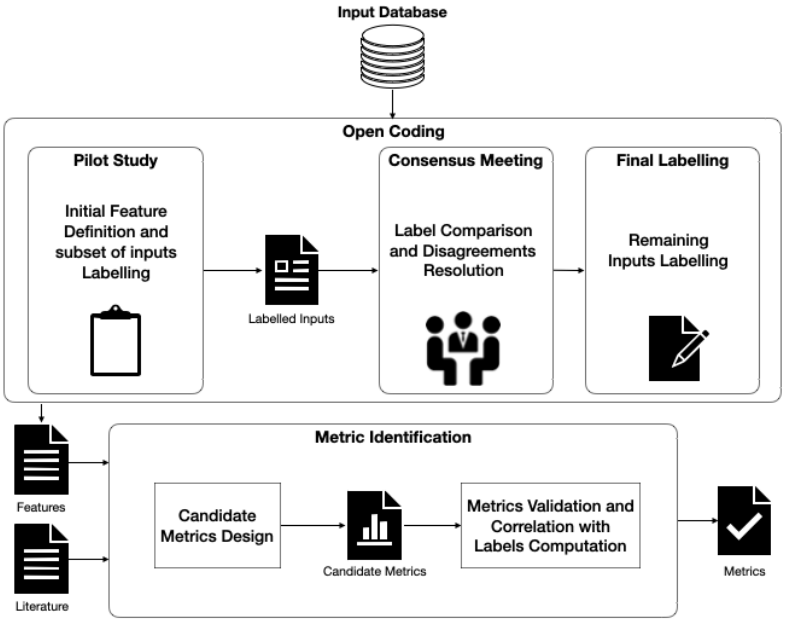


Fig. 3. Feature Selection Methodology

Table 1. Feature selection and validation: output of the proposed methodology in the two reference application domains

Application Domain	Case Study	Feature Name	Metric Name	Agreement	Correlation	p-value
Digit Recognition	MNIST	Boldness	Lum	100%	0.67	<0.002
		Smoothness	AvgAng	66%	0.05	0.241
		Discontinuity	Mov	100%	0.90	<0.002
		Rotation	Or	100%*	0.43	<0.002
Autonomous Driving	BeamNG	Smoothness	Curv	95.8%	-0.60	<0.002
		Complexity	TurCnt	87.5%	0.63	<0.002
		Orientation	DirCov	89.5%	0.66	<0.002
		Passenger Comfort	StdSA**	–	–	–
		Safety	MLP**	–	–	–

* *Rotation was identified* during *the consensus meeting, after all the assessors agreed upon its meaning (i.e., Agreement*= 100%*).*
** *StdSA and MLP were identified in the study about quality of driving metrics for self-driving cars by Jahangirova et al. [35]*

We set up a pilot study in which independent assessors analysed an initial set of samples and proposed potentially relevant structural and behavioural features, while labelling these samples. During this study, each input was labelled by multiple assessors in order to let them gain confidence in the labelling procedure.

This procedure is supported by a Web application that we developed, which ensures that unlabelled inputs are equally distributed among the assessors, enables assessors to label inputs according to the existing features as well as to define new features. Figure 4 shows a snapshot of this Web application for labelling road images. There are an interactive image of the road with arrows indicating the sample positions of the car in the road ❶ and a text box to

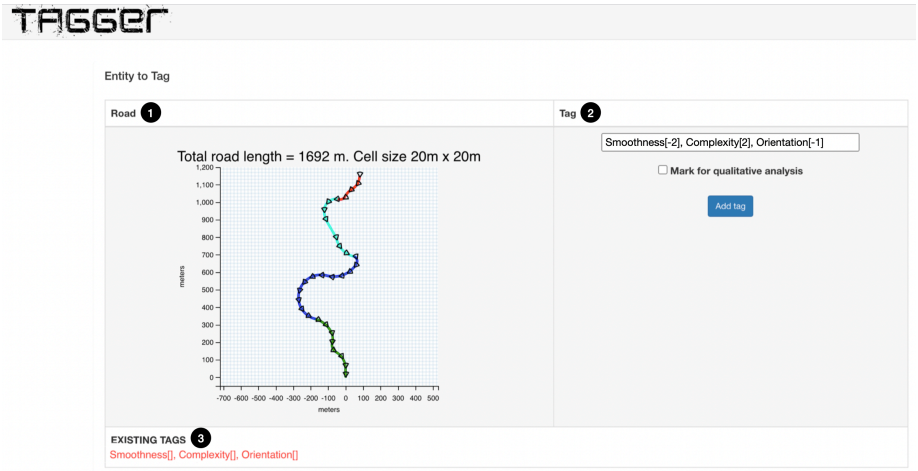


Fig. 4. A view of the Web application used for labelling the inputs. The (interactive) left panel shows the aerial view of a virtual road divided in road segments. In this panel, little triangles depict the field of view of an hypothetical vehicle driving in the middle of the road, while additional meta-data about the road (e.g., total length) are given for reference. The right panel shows the form used by the labellers to tag the input or define additional tags. Finally, the bottom panel shows the set of tags currently defined for virtual roads. Note: a similar page (not shown in the figure) is used to label the images of handwritten digits.

assign the label ❷. As shown at the bottom of the figure, the Web application provides the list of already created labels ❸, which can be reused by the assessors. This choice helps them to use consistent naming without introducing substantial bias [33]. In the example in Figure 4, the assessor carefully inspected the road shape provided in the left panel ❶ and decided that it has a very sharp angle, a very large number of turns, and covers a moderately small range of directions. Consequently, the assessor filled the text box in the right panel ❷ by assigning a value in the range [-2;+2] to each feature. Notably, the assessor reused the suggested existing tags ❸ as they satisfactorily encoded the identified features. Otherwise, the assessor could have introduced different tags, which would have been proposed to all the assessors in the bottom panel. Based on our experience, this procedure took up to 1 minute for each image.

After the pilot study, the assessors performed a consensus meeting in which they agreed on the features' definition and the interpretation of the rating values collected during manual labelling. Consensus meetings are important for assessors to find a common way of labelling the samples and, thus, to be consistent with each other. In fact, in these meetings assessors discussed their interpretation of each feature and the reasons behind their decisions (e.g. assessors agreed that the presence of sharp angles in a digit is the most relevant aspect for determining its smoothness). As a result, the study produced a consistent set of features that assessors used to label the remaining samples (*Final Labelling* step in Figure 3). In the final labelling, since assessors had reached a common understanding of the features and of their possible values, they independently evaluated the remaining unlabelled samples, i.e., each sample was evaluated by a single assessor.

In the digit recognition domain, three assessors took part in the pilot study and analysed 30 images from the MNIST database. In particular, each assessor was assigned 20 images, so that each input was evaluated by two assessors. Assessors identified four features that could potentially characterise images of digits: **boldness**, which indicates the thickness of the handwriting's strokes; **smoothness**, which indicates the absence of sharp angles in the digit; **discontinuity**, which indicates the presence and the extent of disconnected segments forming the digit; **rotation**,

which indicates how much the digit is tilted with respect to the vertical axis. The first three features were proposed during the pilot study, whereas the last one emerged during the consensus meeting. As reported in Table 1 (column *Agreement*), the assessors fully agreed upon boldness, discontinuity, and rotation, while they only partially agreed on smoothness. Agreement is measured as the proportion of labelled images on which the two assessments differ by at most one point in the considered rating scale (we adopted 5 point rating scales for all features). After the consensus meeting, each assessor labelled 200 images for a total of 600 images from MNIST.

In the autonomous driving domain, four assessors analysed 40 virtual roads. We assigned 20 roads to each assessor, so that each road was evaluated by two assessors. The assessors identified three structural features that could potentially characterise virtual (i.e., simulated) roads: **smoothness**, which indicates the presence of sharp turns; **complexity**, which indicates how windy the road is; **orientation**, which indicates how many directions (e.g., N, S, W, E) the road covers. In this case, the assessors agreed upon all the proposed features, but their agreement was slightly lower than for digits. After the consensus meeting, each assessor labelled 100 roads for a total of 400 virtual roads.

Figures 1 and 2 show examples of handwritten digits and virtual roads organised in feature maps defined using some of the identified features.

2.2 Metric Identification

We identified metrics that quantify the features, either by referring to standard metrics proposed in the literature or by designing ad-hoc metrics. Then, we validated the proposed metrics by utilising correlation analysis [39], i.e., we kept only the metrics that significantly correlate with the rating values collected during open coding.

In the digit recognition domain, we identified the following metrics to capture the relevant features of handwritten digits: **luminosity (Lum)**, which counts the number of light pixels whose values are above 127) of the image and quantifies boldness; **average angle (AvgAng)** of the Bézier curves in the SVG representation of the digit, which quantifies its smoothness; **moves (Mov)**, which quantifies discontinuity by measuring the cumulative Euclidean distance between consecutive digit segments' end-points (to obtain the segments of a digit, we convert its bitmap to SVG through vectorisation); and **orientation (Or)**, which quantifies rotation measuring the angle between the digit's principal direction and the vertical axis, obtained by computing the angular coefficient of the linear regression of the non-black pixels, i.e., pixels with value greater than 0. As columns *Correlation* and *p-value* of Table 1 show, luminosity, moves, and orientation positively and significantly correlate with the corresponding features (p-value < 0.05). Instead, the correlation between AvgAng and smoothness is weak and not significant enough (p-value > 0.05). Furthermore, since the assessors did not fully agree upon the meaning of smoothness, we decided to discard this feature.

In the autonomous driving domain, we identified the following metrics to capture the structural features of virtual roads which are computed on the points defining the roads' centre lines: **maximum curvature (Curv)**, which quantifies road smoothness as the inverse of its turns' radius; **turn count (TurCnt)**, which counts how many times a road significantly changes direction (i.e., by more than 5 degrees) and quantifies its complexity; and **direction coverage (DirCov)**, which quantifies orientation by counting the angular sectors covered by the road.

In this work, we slightly modified the metric to measure the roads' smoothness in order to improve its expressiveness with regards to sharper turns. In fact, in our previous work we used the minimum turns' radius (*i.e., MinRadius*), rather than its inverse. We deemed the MinRadius metric as sub-optimal since it showed a wide range of values for smooth roads, which are usually less problematic for driving agents.

As reported in Table 1, all these metrics significantly correlate with the features; hence, we accepted them all. However, since these metrics characterise only the inputs' structural features, we considered additional metrics to

characterise the quality of self-driving, i.e., metrics to quantify the behavioural features. In particular, we selected the following two metrics from the study by Jahangirova et al. [35]: **standard deviation of steering angle (StdSA)**, which measures the activity of the driving agent on the steering wheel and can be used to quantify passenger comfort; and the car's **mean lateral position (MLP)**, which measures how close the driving agent drives from to the lane margins and can be used to measure safety [22, 59**?**].

3 GUIDING ILLUMINATION SEARCH WITH CONTRIBUTION SCORE

DeepHyperion-CS aims to extensively explore the feature space of a DL system to find inputs with diverse characteristics that induce the system to deviate from the expected behaviour. In our previous work [73], we demonstrated that this goal can be achieved with Multi-dimensional Archive of Phenotypic Elites (MAP-Elites), an Illumination Search algorithm proposed by Mouret and Clune [52].

Given N dimensions of variation of interest, which define the feature space (i.e., the feature map to explore), MAP-Elites looks for test inputs that expose misbehaviours in the system under test at each point in the space defined by those dimensions (i.e., the map's cells). Its goal is to fill the feature map with the fittest individuals, i.e., inputs that expose or are close to exposing misbehaviours.

MAP-Elites needs a domain- and problem-specific *fitness function* to measure the degree of misbehaviour exhibited by the system when executed with a given candidate solution as input. For example, when testing Deep Neural Networks (DNNs) that recognise handwritten digits in greyscale images, two dimensions of interest may be the boldness and discontinuity of the handwriting stroke (see section 2). In this case, DeepHyperion-CS uses the misclassification distance as fitness function [12, 18, 59] to generate greyscale images containing digits written using strokes with different boldness and discontinuity (see Figure 1). The misclassification distance is computed as the difference between the activation value of the neuron associated with the correct label and the highest incorrect activation from the DNN's *softmax* layer output (hence, it becomes negative as a misclassification occurs).

The original MAP-Elites algorithm uses uniform random individual selection to perform the search, i.e., at each iteration, it generates a new input by modifying an individual randomly chosen among the ones already occupying some map cells. The motivation behind this choice is that random selection avoids biasing the search and possibly achieving suboptimal solutions [52]. For instance, selecting the fittest individual at each iteration may drastically reduce the population's diversity and lead to premature convergence of the search.

However, smarter selection mechanisms usually improve search-based algorithms compared to the random baseline, e.g. survival of the fittest [21] and promotion of the most diverse individuals [44, 50]. Therefore, in this work we integrate a novel selection operator into MAP-Elites, specifically designed to promote individuals that contribute more to the map exploration. The ability of an individual to generate many and diverse new inputs is captured by our novel metric, named *contribution score (CS)* (see section 3.5 for a detailed description). Our assumption is that an individual contributes to the search when it generates mutants that occupy previously empty cells or are fitter than existing individuals.

If an individual contributes to the search more often, it could be more useful to generate better mutants also in next iterations. On the other hand, if an individual does not contribute to the search for several iterations, it is unlikely that it will generate better mutants later; in this case, we progressively reduce that individual's CS score in order to give it a lower priority during the selection. Consequently, selecting individuals with higher CS can lead to fill more cells of the feature map, possibly in fewer iterations, than uniform random selection. Moreover, exploring more feature combinations may also lead to exposing more misbehaviours.

Algorithm 1: DeepHyperion-CS's Illumination Search

Input : B: execution budget
featurelist: list of features
seedsize: seed pool size
popsize: population size
rankselectionprob: rank selection probability
rankbias: rank bias
Output : M: feature map

```
1  map M ← InitializeMap(featurelist);
2  seeds S ← GenerateSeeds(seedsize);
3  foreach s ∈ S do
4  |  Evaluate(s);
5  end
6  population P ← InitialisePopulation(S, popsize);
7  foreach ind ∈ P do
8  |  M ← UpdateMap(ind) ;
9  end
10 while elapsedBudget < B do
11 |  ind ← CS-RankSelection(M, rankselectionprob, rankbias);
12 |  ind_μ ← Mutate(ind);
13 |  Evaluate(ind_μ);
14 |  M ← UpdateMap(ind_μ);
15 |  ind ← UpdateCS(M);
16 end
17 return (M)
```

Algorithm 1 outlines the high-level steps of the Illumination Search approach implemented by DeepHyperion-CS and highlights the lines that differ from DeepHyperion. The algorithm starts by filling an empty N-dimensional and discretised feature map M (line 1) with an initial population P to be evolved (line 6), where N is the number of features in the *featurelist* provided as input. The initial population is drawn from a pool S of valid candidate inputs, called *seeds*, that are evaluated using the EVALUATE function which computes the fitness function (i.e. closeness to misbehaviour) and the features' values of the considered individual (lines 3–5). The cost of the input evaluation is domain-dependent and spans from a simple model prediction, e.g., for handwritten digits, to performing expensive simulations, e.g., for lane keeping. On the basis of the computed features, the candidate inputs are placed into M following the update map rule (lines 7–9), i.e., each map cell can be occupied only by the fittest individual with feature values corresponding to that cell. After creating P, the algorithm performs the main evolutionary loop (lines 10-15) until a termination condition on the execution budget is met. At each loop iteration, an individual *ind* is chosen from the current map using the CS-based rank selection operator (line 11). The selected individual is mutated to generate a new input ind_μ (line 12) and, then, its features and fitness are evaluated (line 13). The map is updated with ind_μ (line 14): if it has a higher fitness value than the individual in the map cell it occupies, it replaces the existing entry in the map (this is done also if the map entry is currently empty). Finally, the CS of the parent individual *ind* is updated according to whether its mutant

ind_μ contributed to the exploration or not (line 15). In the next sections, we detail the key aspects of DeepHyperion-CS and describe how we applied it to the chosen application domains.

3.1 Model-Based Input Representation

DeepHyperion-CS is a model-based test input generation technique [65]: it generates complex inputs (e.g., greyscale images) by manipulating a *model* of the input, instead of directly modifying the raw input data (e.g., pixels). Consequently, DeepHyperion-CS requires a generative model of the input data processed in its application domain. Generative input models are largely domain-specific and are commonly employed in several domains, including safety-critical ones [41].

A possible alternative to model-based manipulation could be to directly modify the raw input data (e.g., pixels for MNIST) as done in traditional adversarial Machine Learning (ML). Adversarial ML techniques focus on applying the minimal changes that can trigger a misbehaviour and are guaranteed to achieve this goal [9]. However, they are not focused on generating inputs with different structural features and, thus, covering the feature map. Another alternative for input generation are generative ML approaches that approximate the input distribution, such as Variational Auto-Encoders (VAEs) [37] and Generative Adversarial Networks (GANs) [18]. VAEs and GANs are very useful when a model of the inputs is not available, e.g., real-world images from ImageNet. However, generative ML based approaches rely on the quality of both a representative training set and trained generative ML models, which might be hard to achieve for complex problems.

We evaluated DeepHyperion-CS on two reference problems, handwritten digit recognition in the image classification domain and lane-keeping in the automotive domain.

For the handwritten digits recognition problem, we refer to the image format adopted by the MNIST database [42] that consists of 70 000 greyscale 28×28 images of handwritten digits. DeepHyperion-CS abstracts each digit as a sequence of (start, end, and control) points that define Bézier segments by utilising the Potrace algorithm [61] and stores them as Scalable Vector Graphics (SVG)[1] files, as shown in Figure 5. In particular, Potrace performs a sequence of operations, including binarisation, despeckling and smoothing, which draw a smooth contour made of Bezier segments around the considered image. We used Potrace since it represents the state of the art in vector model extraction from images, (see, e.g., the Inkscape tool[2]), and can be easily integrated into Python code via the `pypotrace`[3] API.

For the lane-keeping problem, we refer to the simulated driving scenarios defined by state-of-art approaches for testing lane-keeping systems [22, 53, 59] that consist of flat, two-lane, two-way, asphalt roads surrounded by green grass on which the ego-car (i.e., the vehicle under the test) has to drive on the right lane. The environment is set to a clear day without fog. DeepHyperion-CS abstracts roads as sequences of control points in a bi-dimensional space. DeepHyperion-CS interpolates the control points using Catmull-Rom cubic splines [13] to transform them into virtual roads to be rendered in the simulator. Figure 6 shows the control points of the centre line spline as larger red dots and the interpolated points that define the road as smaller grey dots.

3.2 Fitness Function

Intuitively, a suitable fitness function for testing DL systems should quantify how close the DL system is to exhibit a misbehaviour [22, 30, 59]. In the following, we describe the two fitness functions we designed to address the handwritten digit recognition and lane-keeping problems, respectively.

[1] https://www.w3.org/Graphics/SVG/
[2] https://wiki.inkscape.org/wiki/Potrace
[3] https://github.com/flupke/pypotrace

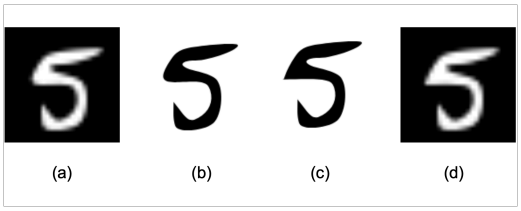


Fig. 5. Digit input representation and mutation. (a) original input; (b) original SVG model after vectorization; (c) SVG model mutated by moving a control point; (d) mutated input.

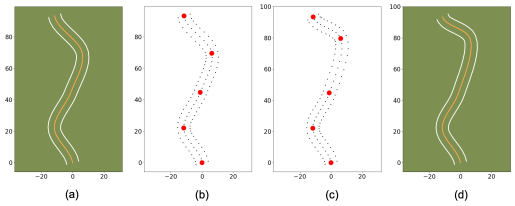


Fig. 6. Road input representation and mutation. (a) original input; (b) original model; (c) model mutated by moving a control point; (d) mutated input.

For the handwritten digit recognition problem, we rely on the fact that the DNN under test recognises the digits in the input image by selecting the class with the highest activation level in its softmax output layer [25]. Therefore, by computing the difference between the activation level of the neuron associated with the correct class and the maximum activation level associated with the other, incorrect classes, we can effectively measure whether the prediction was correct (positive fitness value) or wrong (negative fitness value). More importantly, using this fitness function, DeepHyperion-CS can measure how close an input is to cause a misbehaviour and can expose misbehaviours by minimising the fitness value.

For the lane-keeping problem, we adopt a fitness function that scores higher tests causing the ego-car to drive closer to, or even across, the lane's margins. Specifically, DeepHyperion-CS calculates the fitness of a test as $\min(w/2 - d)$, where w is the width of the lane the ego-car travels on, and d is the distance of the ego-car from the lane centre. The position of the car is approximated by its centre of mass. The fitness function returns its maximum value $w/2$ when the car is at the lane centre. DeepHyperion-CS aims to minimise this fitness function causing the ego-car to drive over the lane's margins (negative fitness values).

Since it is more important to find all unique misbehaviours that happen in different conditions, rather than finding test inputs that cause a "large amount" of misbehaviour (large negative value of the fitness function), DeepHyperion-CS aims at generating as many diverse misbehaviour-inducing tests as possible, while the fitness function is capped to a small negative value (i.e., -0.1) independently of the misbehaviour, thus avoiding that DeepHyperion-CS ends up replacing individuals that caused already discovered misbehaviours with individuals causing more extreme misbehaviours that happen in similar or the same conditions. This strategy has the advantage of making it hard for the fittest individuals to dominate the selection, which might lead to premature convergence [5].

3.3 Feature Map

A feature map represents the feature space defined by N dimensions of variation (i.e., the features) that are relevant for characterising the tests generated by DeepHyperion-CS.

DeepHyperion-CS characterises each individual by placing it into the feature map M using the following mapping function:

$$x_i = \lfloor \alpha_i \cdot ind.f_i \rfloor \tag{1}$$

where $ind.f_i, \forall i \in [1 : N]$ refers to an individual's feature values. According to Equation 1, DeepHyperion-CS computes the i-th index of a cell (i.e., the integer x_i that defines its coordinate along the i-th dimension) by scaling the feature value $ind.f_i$ using the scaling factors α_i.

It should be noticed that if a feature f_i can have negative values, the resulting index x_i becomes also negative. Correspondingly, the feature map will span between negative and positive integers (one way to achieve this in the implementation is to use the index x_i as a hash key and to display a grid spanning along all keys). DeepHyperion-CS uses α to control the map's granularity based on the expected range of each feature and to transform continuous values into the map coordinate system, which is based on integers. The granularity of the feature map (i.e., the number of cells along each dimension) is decided by the user of our approach when setting the scaling factor α_i. Specifically, α_i can be empirically computed as the ratio between the desired granularity, i.e., the desired number of cells, and the expected range of the corresponding feature f_i. The choice of the map granularity affects its discriminative power, since a too low granularity might be insufficient to characterize the misbehaviours and to distinguish them from correct behaviours. However, in our experience any reasonably high choice (as a rule of thumb, more than 25 cells) is enough to ensure good discrimination.

For instance, if the i-th feature's values are expected to range between 0.0 and 2.0, and the desired granularity is 100, a suitable value of α_i would be 50. The α values remain constant during the search, while the size of the map M dynamically increases as DeepHyperion-CS generates individuals with feature values outside the current map boundaries. Initially, M contains no cells; then, as soon as DeepHyperion-CS generates new tests with features that map to indexes outside the current range of values, it grows M to accommodate the newly discovered individuals and adjusts the range of values along the extended dimensions. For instance, if the first mapped individual in a hypothetical bi-dimensional map has indexes $(x_1, x_2) = (2, 3)$, the initial empty map M would be updated to have one cell in each direction at position $(2, 3)$ and ranges $([2 : 2], [3 : 3])$. If later DeepHyperion-CS maps another individual to $(x_1, x_2) = (5, 1)$, M grows along its first dimension to the range $[2 : 5]$ and to $[1 : 3]$ along the second dimension. At this point, the feature map contains 12 cells, 2 of which are filled, i.e., they contain an individual.

We could have considered dynamic scaling factors α_i with maps of fixed size, as they could be beneficial for features with previously unknown ranges. However, dynamic scaling would require to recompute the best solutions within each cell at each rescaling, because the local competition triggered by the search algorithm depends on which individuals are mapped to each cell. In turn, this might cause instabilities of the algorithm, because convergence toward the final map is driven by the winners of the local competitions, which could change completely each time a new rescaling is applied. Moreover, the memory requirements of the algorithm would grow, as all the generated inputs must be kept in memory to repeat the local competition process each time a new scale is adopted. Hence, we decided to rely on a preliminary estimation of the feature ranges to define a set of fixed scaling factors α_i.

Since the size of the final dynamically discovered map may be different between various runs of the algorithm, which may hinder visual inspection of the results, DeepHyperion-CS allows testers to define the granularity of the final map produced by the algorithm. Therefore, it obtains the final map by rescaling the search results as follows:

$$x'_i = \left\lfloor \mathrm{GS}_i \cdot \frac{ind.f_i - min_i}{max_i - min_i} \right\rfloor \tag{2}$$

where GS_i is the desired grid size of the final map, and min_i and max_i are the minimum and maximum values empirically observed for the i-th feature. Rescaling the feature maps eases the comparison of maps produced by DeepHyperion-CS and other test generators across various runs, when results that have different ranges. Therefore, we rely on map rescaling in the experimental evaluation.

In particular, we rescale the maps so that they have the same size and features' ranges across all runs of all test generators, hence avoiding any misalignment between maps.

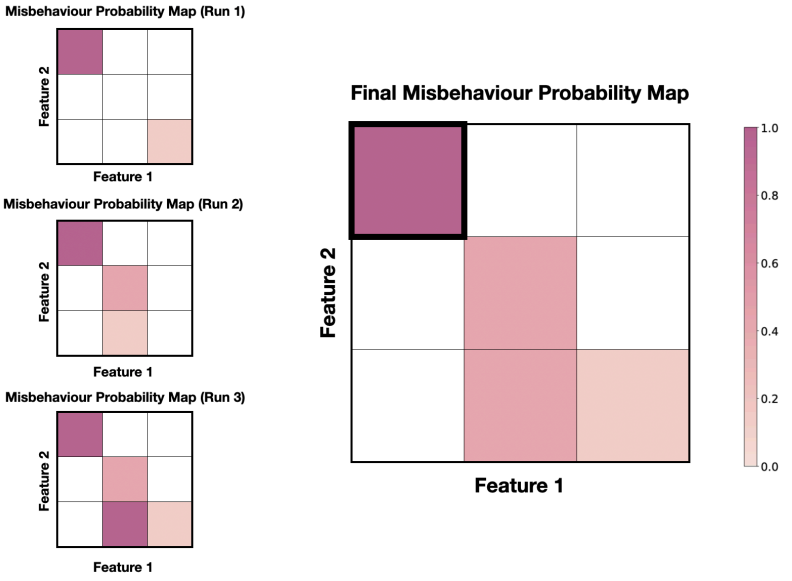


Fig. 7. Misbehaviour probability maps: darker cells correspond to feature combination values that are more likely to induce misbehaviours, dark borders highlight cells with high confidence of producing a misbehaviour.

The generation of multiple inputs that belong to the same cell, e.g., through multiple DeepHyperion's runs, allows the identification of feature map regions where the probability of observing misbehaviours is higher. In fact, a combination of feature values that often corresponds to a misbehaviour may suggest that such feature values are very likely to induce a misbehaviour. In this way, DeepHyperion provides developers with a powerful tool to understand the causes of misbehaviours. Therefore, we synthesise the information collected by DeepHyperion across multiple runs (i.e., all the generated inputs and the corresponding outputs) in *misbehaviour probability maps*, that report the Average Misbehaviour Probability (AMP) associated with each cell. We compute these maps by (1) measuring, within each cell, the ratio of the number of misbehaviour-inducing inputs to the total number of inputs generated by DeepHyperion during each run, and then (2) averaging the resulting values per cell across all the tool's runs. Since DeepHyperion may generate only a small number of inputs in some cells, the corresponding AMP values might be affected by a large error. Hence, we also compute the confidence interval of AMP. In particular, we use Wilson's confidence interval estimator for binomial random variables, which indicates whether the misbehaviour probability estimated for a particular combination of feature values has a low or high error range. We consider a combination of feature values to produce misbehaviours with high confidence if its AMP value is greater than 0.8 and the lower bound of its confidence interval is above 0.65. As shown in Figure 7, misbehaviour probability maps contain blank cells corresponding to feature combination values that have never been observed, while the other cells are shaded proportionally to their AMP values. Combinations producing misbehaviours with high confidence have thick borders.

3.4 Initial Population

DeepHyperion-CS generates an initial population of size *popsize* by choosing inputs from a larger pool of seeds of size *seedsize*, consisting of valid inputs for the system under test. For the handwritten digit recognition problem, these

seeds are existing images randomly drawn from the MNIST database and converted to SVG, while for the lane-keeping problem, seeds are valid roads that are generated randomly.

Generation of DeepHyperion-CS's initial population aims to create a set of diverse individuals from the feature space. Therefore, after evaluating the seed fitness and computing the seed position on the map based on the feature values, DeepHyperion-CS greedily selects individuals from the seed pool so as to maximise their pairwise Manhattan distance (sum of the absolute differences of the map coordinates) [40].

3.5 Contribution Score-Based Rank Selection

As shown in Algorithm 1, each DeepHyperion-CS's evolutionary iteration starts by selecting an existing individual to be mutated from the non-empty cells of the feature map. To select such individual, DeepHyperion-CS can use either a random strategy (RandomSelection) or our novel strategy based on the contribution score (CS-RankSelection). The rank selection probability parameter *rankselectionprob* controls the frequency of usage of each selection strategy. Specifically, increasing values of *rankselectionprob* result in adopting CS-RankSelection more frequently than RandomSelection.

RandomSelection is a standard selection strategy, in which an individual is uniformly sampled among the ones already placed in the feature map.

CS-RankSelection implements a rank selection scheme which selects individuals with a probability proportional to their rank, such that high-ranked individuals are selected with higher probability than low-ranked ones.

Individuals are ranked by CS-RankSelection according to their *Contribution Score (CS)*, which represents the individual's contribution to exploration. An individual has contributed to exploration if it has generated mutants which filled previously empty cells or which replaced existing individuals with better ones. In detail, *CS* is computed as follows:

$$CS(x) = \begin{cases} \frac{CC(x)}{SC(x)} & \text{if } SC(x) > 0 \\ 1 & \text{otherwise} \end{cases} \tag{3}$$

where $CC(x)$, the *Contribution Count*, indicates the number of times mutants of individual x have been successfully placed in the map, while $SC(x)$, the *Selection Count*, indicates the number of times the individual has been selected during the search. According to its definition, *CS* is always bounded between $(0, 1]$. Each individual's *CS* is initially set to 1, which means that *(i)* all the individuals are selected with equal probability before collecting any observation (unbiased initial selection); and *(ii)* individuals who have never been selected are more likely to be selected than others (promoted exploration).

During the search, *CS* values are updated to reflect each individual's actual contribution. For instance, if an individual x_1 with $CC(x_1) = 1$ and $SC(x_1) = 1$ is selected but its mutant is not placed on the map (i.e., the existing individual already placed into its same cell has higher fitness), $SC(x_1)$ increments but $CC(x_1)$ remains the same. As a result, $CS(x_1)$ drops from a solid 1.0 to a less considerable value of 0.5, halving the chances to select x_1 in the following iterations. Interestingly, the contribution score of an individual does not always monotonically decrease during the search since every time $SC(x)$ increases, the corresponding value of $CC(x)$ may or may not increase. The initial value of $CS(x)$ is 1.0 to promote individuals that have not been yet selected. Regime values for $SC(x)$ are usually big, making the difference between contiguous values of $CS(x)$ small, i.e., *CS* is overall smoothly changing. However, there is an initial, transient phase where by design $CS(x)$ is less smooth, e.g. jumping from 1.0 to CS = 0.0 (non contributing individual) and then to CS = 0.5 (contributing individual), to quickly lower the rank of individuals that proved not to contribute to the search.

To perform rank selection of individuals, we use the linear ranking function proposed by Whitley [67]. This approach to rank individuals is widely used in search-based software testing since it addresses the problem of maintaining a constant selective pressure of genetic algorithms throughout the search [6, 20, 24, 28, 29, 51]. In accordance with this technique, DeepHyperion-CS sorts the individuals in ascending order based on their CS value. Then, it selects from the ordered list the individual corresponding to the index computed with the following formula:

$$index = size(Individuals) \times \frac{\left(\sqrt{rankbias^2 - 4 \times (rankbias - 1) \times random(0,1)}\right)}{2.0 \times (rankbias - 1)} \tag{4}$$

where function *size* returns the length of a list and the function $random(0, 1)$ returns a random floating point number between 0 and 1. The *rankbias* parameter ranges between 1.0 and 2.0 and influences the algorithm's behaviour by biasing the selection towards individuals with higher ranks (i.e., *rankbias* values close to 2.0) or lower ranks (i.e., *rankbias* values close to 1.0) [67]. n the former case, the selection operator does not always select the best individual, which helps to avoid local optima.

In summary, by adopting our rank selection operator based on contribution score, DeepHyperion-CS can bias the search towards solutions with high contribution scores, hence exploring the feature space at large, "illuminating" the search space as much as possible. Additionally, by exposing parameters such as *rankselectionprob* and *rankbias*, DeepHyperion-CS enables testers to control the selection pressure and the level of bias imposed by the rank selection on the overall search process.

3.6 Model-Based Mutation Operators

After selecting an individual, DeepHyperion-CS mutates it to generate a new input. Since DeepHyperion-CS is a model-based test generator, its mutation operators manipulate an input model rather than the input itself. DeepHyperion-CS uses mutation operators that apply small perturbations to the input models within a customisable range.

For the handwritten digit recognition problem, DeepHyperion-CS manipulates the SVG image model's points to mutate the corresponding digit shape while preserving realism [59]. Then it applies a rasterisation operation to obtain the input in the MNIST database format[4] (see Figure 5). For the lane-keeping problem, DeepHyperion-CS mutates the road geometry by applying a displacement to the coordinates of the model's control points (see Figure 6).

Despite the small perturbations applied by DeepHyperion-CS, the generated mutants may not be different from their parents or even valid once concretised into actual test inputs. Therefore, DeepHyperion-CS verifies that the mutants are different from their parents and comply with the constraints of the input domain before evaluating them. DeepHyperion-CS keeps mutating the same parent individual until a valid mutant is found.

For the handwritten digit recognition problem, DeepHyperion-CS (i) computes the Euclidean distance between the mutant and its parent, which must be greater than 0; (ii) computes the Euclidean distance between the mutant and the starting seed, which must be greater than 0 and lower than 2.

For the lane-keeping problem, it checks that mutated roads (i) do not have control nodes identical to the parent's control nodes; (ii) are entirely contained within a squared bounding box of fixed size (i.e., the driving simulator's map boundaries); and (iii) do not self-intersect.

[4] DeepHyperion-CS utilises the open-source graphic libraries LibRsvg and Cairo for rasterising SVG images to the MNIST format.

4 EXPERIMENTAL EVALUATION

4.1 Research Questions

Our evaluation aims at understanding the effectiveness of the Contribution Score in guiding DeepHyperion toward feature map generation. Hence, we seek answers to the following research questions:

RQ1. *Does the guidance provided by Contribution Score improve DeepHyperion's effectiveness?*

Effective test generators produce tests that trigger as many diverse misbehaviours as possible, i.e., they expose multiple problems but do not expose repeatedly the same problems. Otherwise, they waste computational resources in generating tests that do not provide any new insight into the quality of the DL system under test.

Metrics: We assess effectiveness by counting the *Mapped Misbehaviours (MM)*, i.e., the cells of the feature map that contain misbehaviour-inducing inputs. To measure the diversity of the misbehaviour-inducing inputs, we compute the *Misbehaviour Sparseness*. In particular, we compute the average Manhattan distance between cells containing misbehaviours and the average maximum Manhattan distance between cells containing misbehaviours. We consider two slightly different sparseness metrics to take into account outliers and denser map regions:

$$\text{Misbehaviour Sparseness (Avg. Max)} = \frac{\sum_{i \in MM} \max_{j \in MM} \text{dist}(i, j)}{|MM|} \tag{5}$$

$$\text{Misbehaviour Sparseness (Avg.)} = \frac{\sum_{i,j \in MM, i \neq j} \text{dist}(i, j)}{|MM|(|MM| - 1)} \tag{6}$$

RQ2. *Does the guidance provided by Contribution Score allow DeepHyperion-CS to explore the feature space more extensively than DeepHyperion?*

Thorough testing should exercise many behaviours of the systems under test. This can be achieved by extensively exploring the feature space.

Metrics: We measure the thoroughness of exploration by counting the *Filled Cells (FC)* in the map, i.e., the cells of the feature map that contain at least one input. We quantify how broadly those filled cells spread over the feature space by measuring their sparseness, i.e., the *Coverage Sparseness*. Similarly to Misbehaviour Sparseness, we consider the following two sparseness metrics:

$$\text{Coverage Sparseness (Avg. Max)} = \frac{\sum_{i \in FC} \max_{j \in FC} \text{dist}(i, j)}{|FC|} \tag{7}$$

$$\text{Coverage Sparseness (Avg.)} = \frac{\sum_{i,j \in FC, i \neq j} \text{dist}(i, j)}{|FC|(|FC| - 1)} \tag{8}$$

RQ3. *How efficient is DeepHyperion-CS in exploring the feature space and generating test inputs that expose diverse misbehaviours?*

Testing DL systems can be costly, especially when it is conducted at the system level, as happens, e.g., with simulation-based testing of self-driving cars. Therefore, we evaluate how quickly test generators fulfill the testing objectives of triggering misbehaviours and extensively exploring the feature space.

Metrics: We assess test generation efficiency by measuring the *Area Under the Curve (AUC)* of Mapped Misbehaviours and Filled Cells. AUC is a standard performance metric, and higher values of AUC indicate more efficient test generators.

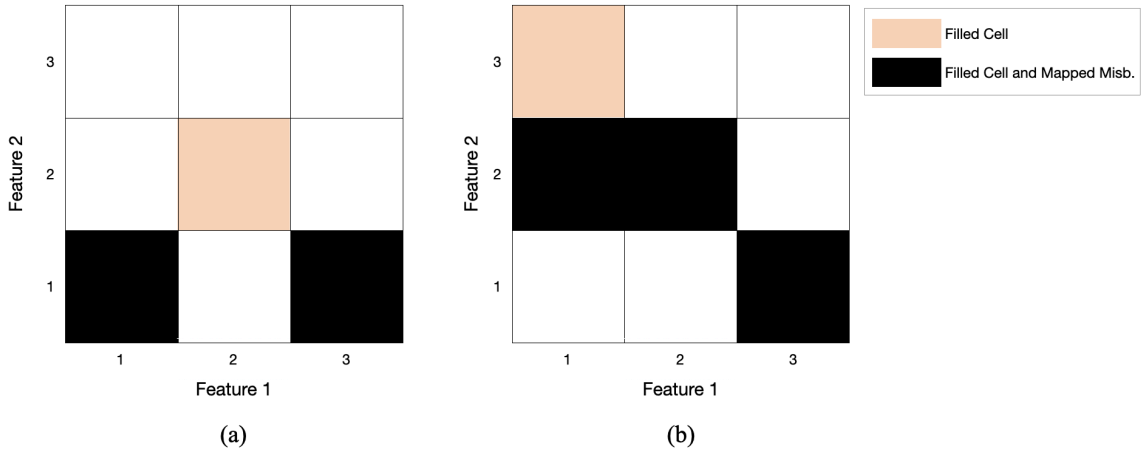


Fig. 8. Example feature maps to explain training set expansion: (a) training map; (b) DeepHyperion-CS map

The previous research questions aim to characterise DeepHyperion-CS as a test generator. The ability of DeepHyperion-CS to explore the feature space at large can be helpful to DL developers in several other tasks. For instance, *training set expansion*: the performance of DL systems is limited by the size and quality of the datasets used to train them [33]. DeepHyperion-CS can be used for characterising and expanding those datasets, which could lead to an improvement of the DL systems' quality. Consequently, we investigate the following additional research question:

RQ4. *Can DeepHyperion-CS be used to expand the training data? Can it find misbehaving inputs also in cells that were already occupied by non-misbehaving training data?*

The knowledge acquired by a DL system is limited by the diversity of the data that have been used to train it. We evaluate how DeepHyperion-CS can expand such knowledge beyond the training set, by identifying feature combinations that are not covered by the existing training set. New input data generated for such initially uncovered feature map cells can expand the training set and increase the generalisation capability of the system.

Such initially uncovered cells are particularly interesting when they contain an input triggering a misbehaviour. On the other hand, DeepHyperion-CS's fitness-guided local competition can also generate misbehaviour-inducing inputs for feature combinations that were not associated to any misbehaviour in the training set. Therefore, we also evaluate how many unknown misbehaviours are triggered by DeepHyperion-CS in cells that are either covered or uncovered by the training set.

Metrics: We answer this question by comparing the feature maps produced from the training set with those generated by DeepHyperion-CS. We measure the size of *Filled Cell Expansion* (FCE), computed as the cells filled by DeepHyperion-CS that were uncovered in the training set:

$$\text{FCE} = |FC_{DH} \setminus FC_{ts}| \tag{9}$$

where FC_{DH} is the set of cells filled by DeepHyperion-CS, whereas FC_{ts} is the set of cells filled with training set data.

In Figure 8, we show an example of a map corresponding to the training set (a) and a map produced by DeepHyperion-CS (b). In this example, |FCE| is 2, since DeepHyperion-CS covers two new cells, at coordinates $\langle 1, 2 \rangle$ and $\langle 1, 3 \rangle$, which are not covered in the training set map.

To measure how DeepHyperion-CS is able to generate new misbehaviours, we define *Mapped Misbehaviour Expansion* (MME) as:

$$MME = |MM_{DH} \setminus MM_{ts}| \tag{10}$$

where MM_{DH} is the set of cells containing misbehaviours in maps generated by DeepHyperion-CS, whereas MM_{ts} is the set of cells containing misbehaviours in the training set. In Figure 8, |MME| is 2 because two misbehaviours are new: those at coordinates $\langle 1, 2\rangle$ and $\langle 2, 2\rangle$.

MME consists of two distinct sets: (1) the misbehaviours generated by DeepHyperion-CS in cells not covered by the training set (MME_{uncov}); and, (2) the misbehaviours generated by DeepHyperion-CS in cells that are covered by training set data, but only by correctly behaving inputs (MME_{cov}). We define these sets as follows:

$$MME_{uncov} = MME \setminus FC_{ts} \tag{11}$$

$$MME_{cov} = MME \cap FC_{ts} \tag{12}$$

Considering the maps in Figure 8, $|MME_{uncov}| = 1$ since the cell $\langle 1, 2\rangle$ is empty in the leftmost map and contains a misbehaviour in DeepHyperion-CS's map. $|MME_{cov}| = 1$ since the cell $\langle 2, 2\rangle$ is covered in the training set map and contains a misbehaviour in the rightmost map. The definitions of MME_{uncov} and MME_{cov} are oblivious of the cells already containing a misbehaviour in the training set, since such cells represent known issues that were identified before the generation of new test inputs. Therefore, cells at coordinates $\langle 1, 1\rangle$ and $\langle 1, 3\rangle$ do not contribute to *MME*, since they contain known misbehaviours, already presented in the training set.

4.2 Subject Systems

We evaluate DeepHyperion-CS on MNIST and BeamNG, two different DL systems widely used in the literature to assess testing techniques for DL systems [58, 69].

The MNIST system recognises handwritten digits from the MNIST dataset [42]; hence, it performs a classification task. Its DNN predicts which digit is represented in a greyscale image. In particular, we consider the popular convolutional DNN architecture provided by Keras [14]. We trained this DNN on the MNIST training set using its default configuration, i.e., 12 epochs, batches of size 128, and a learning rate equal to 1.0. Our digit classifier achieved 99.8% classification accuracy on the MNIST testing set.

The BeamNG system is a simulation-based self-driving car. It implements an end-to-end, vision-based driving agent that can follow the lane in a road. BeamNG includes a DL-based Lane Keeping Assist System (LKAS), i.e., a DNN able to predict the steering angle of the car given the image of its onboard cameras; hence, it solves a regression problem. For this task, we adopted the widely known DAVE-2 architecture designed by Bojarski et al. at NVIDIA [10].

The whole DL system is tested in the BeamNG.research driving simulator [7], a state-of-the-art simulator widely used in research [53]. We trained the model for 4 600 epochs, with batches of size 128 and a learning rate equal to 0.001, achieving a mean squared error (MSE) of $4.31e^{-5}$ on the test set. Our training set consists of images captured by the on-board camera, labelled with the steering angles provided by the simulator's autopilot while driving on virtual roads up to $25Km/h$. To avoid biasing the results, we collected training images by letting the autopilot drive on the same 20 seed roads used by each of the input generators considered in the experimental evaluation.

Table 2. DeepHyperion-CS Configurations

Parameter	Test Subject	
	MNIST	BeamNG
seed pool size	900	80
population size	800	48
time budget (s)	3600	36000
mutation range lower bound	0.01	1
mutation range upper bound	0.6	6
ranked selection probability	0.5	0.5
rank bias	1.5	1.5
feature combinations	(Mov, Or)	(MLP, StdSA)
	(Or, Lum)	(MLP, TurnCnt)
	(Lum, Mov)	(StdSA, Curv)

4.3 Experimental Procedure

We addressed our research questions by running DeepHyperion-CS and other state-of-the-art test input generators against the two considered test subjects. At the end of the runs, we used the results generated by each tool to compute the corresponding feature maps. To ensure a fair comparison, all the maps were generated with the same number of cells for each feature, i.e. up to 25 cells. The extreme values defining the range for each feature are the ones observed across the runs of all the tools. Since the final maps produced by DeepHyperion and DeepHyperion-CS may have different ranges and higher number of cells, we rescaled them by using the formula described in Equation 2.

Since the test subjects are fundamentally different, we adopted two separate configurations for testing them (see Table 2). We empirically obtained those configurations after observing DeepHyperion-CS's behaviour in few preliminary runs. Specifically, DeepHyperion-CS obtained the seeds for MNIST by randomly selecting 900 inputs from the official MNIST test set, all belonging to the same class (i.e., digit "5") and then selecting the 800 most diverse inputs as initial population. The seeds for BeamNG were 80 valid roads randomly generated by DeepHyperion-CS. Each seed was defined by 10 control points in which the initial point was always at a fixed position, whereas the remaining points were placed at a random position 25 meters away from the previous one and deviating from the previous segment by an angle randomly chosen within a predefined range. DeepHyperion-CS then selects the 48 most diverse roads as initial population.

As regards the selected feature dimensions, we used the features identified by applying our methodology (see section 2). We considered only pairwise combinations of features to ease visualisation and discussion of the results, although DeepHyperion-CS can work also with higher-dimensional maps. For MNIST, we considered all the pairs obtained by combining Boldness (Lum), Discontinuity (Mov), and Rotation (Or), as these are the most significant features we found in our feature selection study (see Table 1). For BeamNG, we considered three out of ten possible pairs of features because executing driving simulations becomes soon prohibitively expensive and running experiments that cover all the possible combinations would take excessive computation time. Nevertheless, we believe that the results we achieved are representative as we cover all the three combinations types: two structural features, two behavioural features, and a combination of a structural and a behavioural feature.

To contextualise the results achieved by DeepHyperion-CS, we compare it against the original DeepHyperion and other state-of-the-art testing tools for DL systems. We configured those approaches according to the configurations that achieved the best performance in their papers.

Specifically, we compared DeepHyperion-CS against:

DLFuzz [26] This tool generates adversarial inputs for image classifiers, such as MNIST, by applying perturbations to the pixels of existing images. It is mainly used for testing the robustness of DL systems. However, since it can only manipulate individual images, we could not apply DLFuzz for testing BeamNG at the system level;

DeepJanus [59] This tool generates test inputs at the frontier of behaviours of DL systems, i.e., pairs of similar inputs that trigger different system behaviours, by using a multi-objective search algorithm. DeepJanus shares with DeepHyperion-CS the same model-based input representation; hence, we could apply it to both MNIST and BeamNG;

AsFault [22] This tool generates safety-critical virtual roads for testing lane-keeping systems utilising a single-objective genetic algorithm. Therefore, we could apply it for testing only BeamNG.

To enable a fair comparison with AsFault, we replaced its original failure identification mechanism with the one employed by the other tools (i.e., DeepHyperion-CS, DeepHyperion and DeepJanus). Additionally, since AsFault generates longer roads than the other tools, we split each road it generates into multiple segments when placing the inputs on feature maps, making it possible to directly compare its output with the other tools. However, in this way a single AsFault input may contribute to coverage of more than one cell in the feature map. While this might introduce an unfair advantage for AsFault, it is balanced by the increased time it takes to simulate longer roads. Therefore, given the same simulation budget, AsFault generates fewer inputs than the other tools, but each such input covers multiple cells.

As the purpose of the considered baseline tools is finding misbehaviour-inducing inputs or frontier inputs, not illuminating the feature space, in our experiments we consider the inputs generated (and possibly discarded) during the input generation process, in addition to the inputs reported as final outputs of the tools. In particular, we report separately the feature maps obtained from the final results of each tool *(black box results)* and the feature maps which contain all the inputs produced during a run of each tool *(white box results)*. Noticeably, for DeepHyperion and DeepHyperion-CS the white box and black box maps coincide by construction, as all generated inputs are used during the search in the local competition within each cell. For AsFault, white box results also coincide with black box results, as this tool returns all the inputs generated during the search.

We followed the guidelines by Arcuri and Briand [4] for comparing the considered randomised algorithms: we ran each tool multiple times and assessed the statistical significance of our conclusions by performing the Mann-Whitney U-test and measuring the effect size using the Vargha-Delaney's $\hat{A}_{12}$ statistic. To enable a fair comparison, we ran the tools in isolation on the same computing nodes and used the same generation budget: 1 hour for MNIST and 10 hours of simulation time for BeamNG. We considered simulated time rather than real time for BeamNG since the former is usually the bottleneck in simulation-based testing [2].

The reason for this remarkable difference between the generation budgets is that testing MNIST consists of feeding it small images and getting the corresponding predictions, an operation that takes milliseconds, while testing BeamNG requires the execution of real-time driving simulations that take minutes to complete. Correspondingly, we were able to repeat the MNIST experiments 30 times, whereas we repeated the BeamNG experiments between 10 and 20 times, adopting the following stopping condition (after 10 runs): no further repetition is conducted when the statistical test

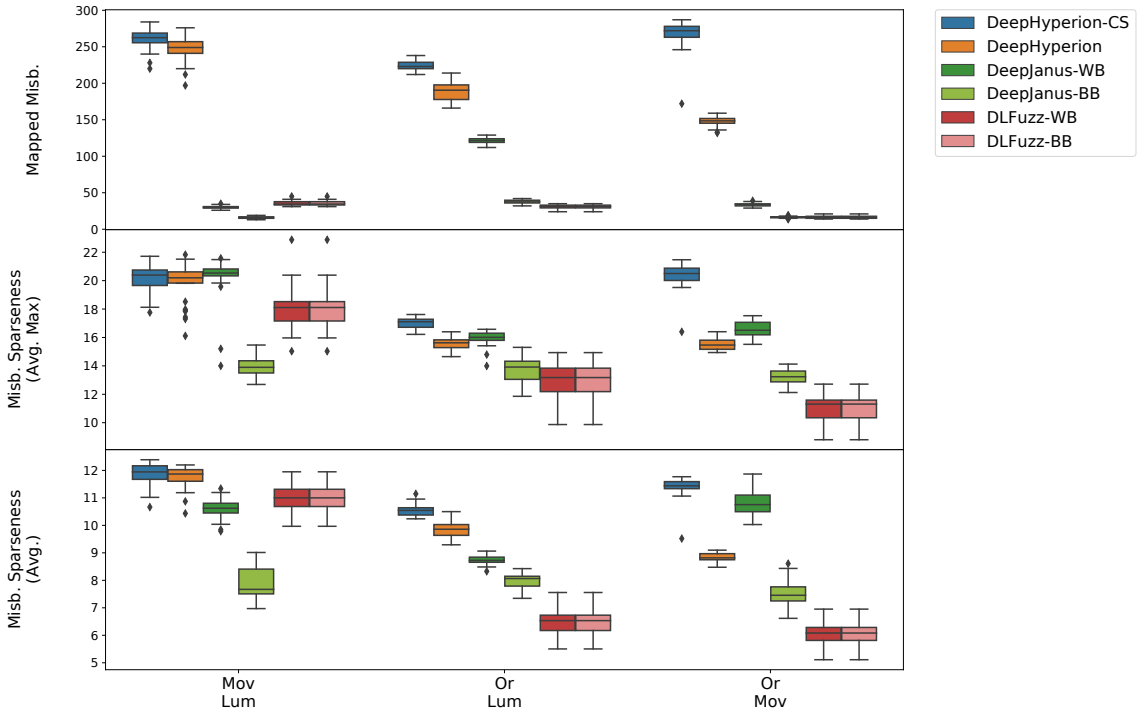


Fig. 9. RQ1: Mapped misbehaviours found on MNIST by the considered tools (top) and their sparseness (middle and bottom).

used to compare DeepHyperion-CS and DeepHyperion reaches statistical significance (p-value < 0.05) or when no statistical significance is reached, but there is sufficient statistical power (statistical power $\beta > 0.8$). In no case the number of repetitions exceeds the upper bound, 20.

5 RESULTS

5.1 RQ1. Does the guidance provided by Contribution Score improve DeepHyperion's effectiveness?

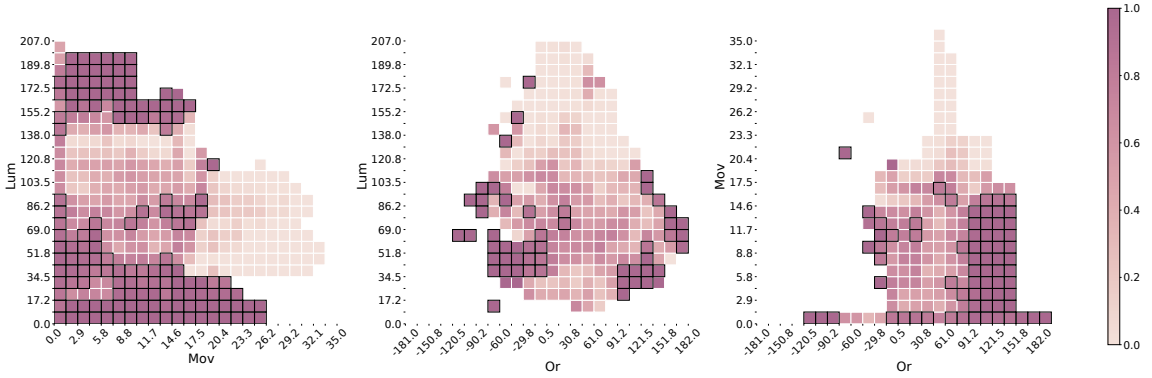
In this RQ, we investigate how many diverse inputs the test subjects failed to handle correctly (i.e., *Mapped Misbehaviours*) and how much they differ between them (i.e., *Misbehaviours Sparseness*).

Figure 9 reports the results achieved by the considered tools on MNIST as box plots grouped by feature combination.

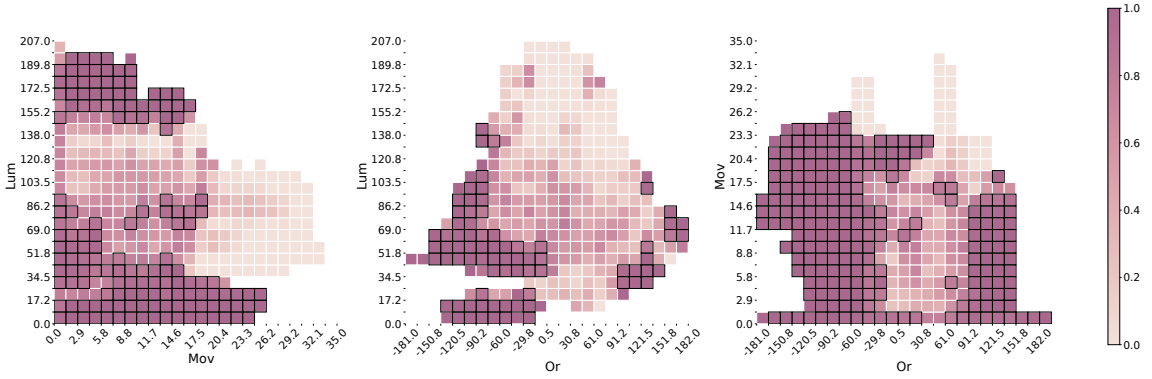
As DeepJanus and DLFuzz output only a subset of the generated inputs at the end of each run, we considered separately the misbehaviours reported at the end of the run (i.e., black box analysis) and the (possibly bigger) set of all the misbehaviours triggered during the same run (i.e., white box analysis).

Correspondingly, the boxes labelled as DeepJanus-WB report more misbehaviours than DeepJanus-BB. DLFuzz's final results (DLFuzz-BB) show the same values of the boxes obtained from all its generated inputs (DLFuzz-WB) because this tool returns to the user all the misbehaviours it triggers during a run.

Figure 9 (top) shows that DeepHyperion-CS found more than 200 diverse misbehaviours for each feature combination. The illumination search based tools, i.e. DeepHyperion and DeepHyperion-CS, always found a significantly larger number of mapped misbehaviours than the other tools (p-value < 0.05 and large effect size). DeepHyperion-CS



(a) DeepHyperion



(b) DeepHyperion-CS

Fig. 10. Misbehaviour probability maps generated by DeepHyperion (a) and DeepHyperion-CS (b) for MNIST

significantly improves DeepHyperion's effectiveness in triggering diverse misbehaviours (p-value < 0.05 and large effect size). In particular, DeepHyperion-CS produced a neatly higher number of mapped misbehaviours than DeepHyperion for the `Or-Mov` feature combination, with over 120 more misbehaviours.

Both misbehaviour sparseness metrics reported in Figure 9 show that DeepHyperion-CS produced comparably sparse or sparser misbehaviours than the other tools. In the majority of feature combinations (i.e. `Or-Lum` and `Or-Mov`), it produced significantly sparser misbehaviours than the other tools (p-values < 0.05 and large effect size). As for `Mov-Lum`, DeepHyperion-CS performed as good as DeepHyperion and DeepJanus-BB (p-values > 0.05) and significantly better than the other tools (p-values < 0.05 and large effect size) in terms of Misbehaviour Sparseness (Avg. Max). Moreover, DeepHyperion-CS is significantly better than DeepJanus-BB for `Mov-Lum` in terms of Misbehaviour Sparseness (Avg.) (p-values < 0.05 and large effect size).

This result was achieved despite DeepJanus explicitly rewards the generated inputs' diversity, having a fitness function that promotes the euclidean distance among solutions.

We further analyse the relationship between the results achieved by DeepHyperion-CS and DeepHyperion by comparing their misbehaviour probability maps (see Figure 10 and Figure 12).

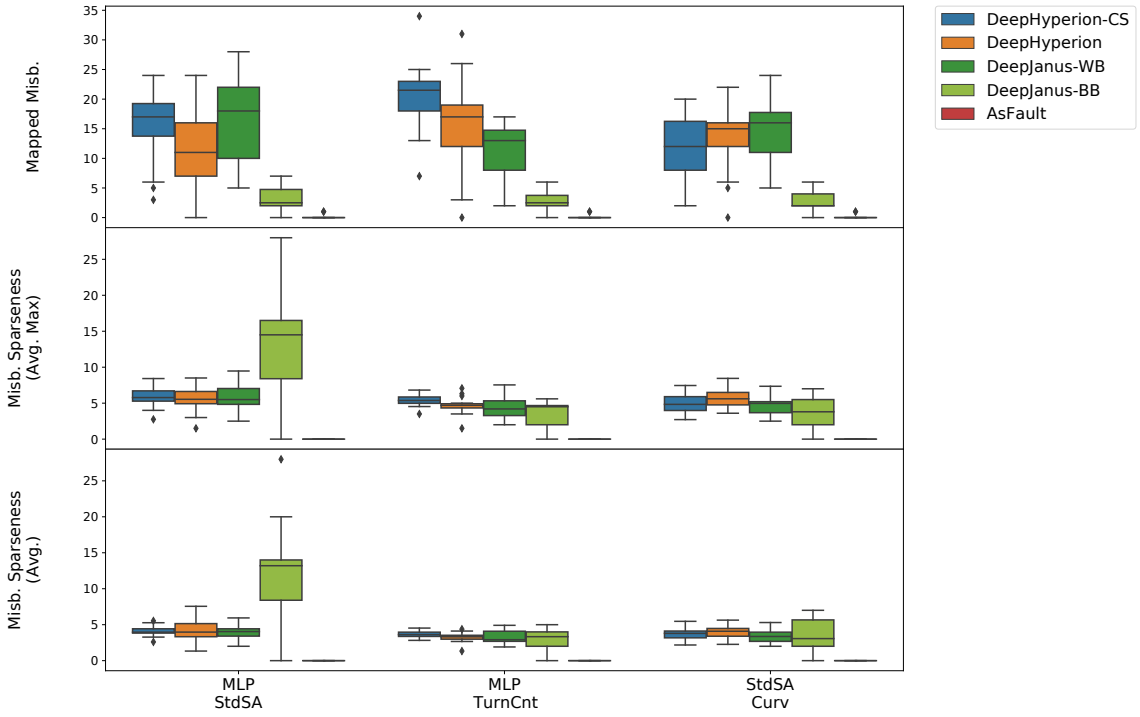


Fig. 11. RQ1: Mapped misbehaviours found on BeamNG by the considered tools (top) and their sparseness (middle and bottom).

The misbehaviour probability maps of DeepHyperion (see Figure 10b) and DeepHyperion-CS (see Figure 10a) for MNIST show well-characterised regions of the feature space that are likely to expose failures, e.g. continuous and thick digits. Therefore, our feature maps can be a powerful tool for developers to understand the conditions responsible for misbehaviours, similarly to the more traditional root-cause analysis.

The probability maps produced by the two tools for the `Mov-Lum` feature combination are similar, with DeepHyperion-CS's map containing slightly more dark cells and 2 more cells with thick border; this is expected, since the two tools achieved similar Mapped Misbehaviours and Misbehaviour Sparseness (see Figure 9).

For what concerns `Or-Lum` and `Or-Mov`, DeepHyperion-CS's misbehaviour probability maps are more informative (i.e. have less empty cells) than DeepHyperion's ones. Consistently with the boxplots in Figure 9, DeepHyperion-CS's maps contain more dark cells than DeepHyperion's maps.

Additionally, thanks to this visualisation we can easily identify the regions in the feature space that DeepHyperion-CS explored and DeepHyperion missed. For instance, DeepHyperion-CS was able to explore large feature space regions characterised by `Or` values smaller than −60.0 (i.e., the left side of `Or-Lum` and `Or-Mov` maps) that DeepHyperion missed. The exploration of those regions paid off, as DeepHyperion-CS discovered a massive number of new misbehaviours there.

Figure 11 shows the Mapped Misbehaviours and Misbehaviour Sparseness obtained when running the tools against our second subject system, BeamNG.

DeepHyperion-CS exposed several diverse misbehaviours for all three feature combinations (more than 10, on average). In particular, DeepHyperion-CS found significantly more mapped misbehaviours than the other tools for `MLP-TurnCnt` (more than 20, on average). The contribution score guidance of DeepHyperion-CS led to significant improvements over DeepHyperion for the `MLP-StdSA` and `MLP-TurnCnt` feature combinations (p-values < 0.05 with medium and large effect size, respectively).

DeepHyperion-CS performed almost always significantly better and never performed worse than the other competitors. In fact, only DeepHyperion for `StdSA-Curv` and DeepJanus-WB for `MLP-StdSA` and `StdSA-Curv` managed to achieve results comparable to DeepHyperion-CS (p-values > 0.05), whereas DeepHyperion-CS found a significantly higher number of mapped misbehaviours in all the other comparisons (p-values < 0.05, large effect size).

DeepJanus-WB reported significantly more mapped misbehaviours than DeepJanus-BB and proved to be a valid challenger to DeepHyperion-CS in two feature combinations out of three. Instead, AsFault reported almost no misbehaviour across all its runs, which suggests that it might be better suited for testing lane keeping systems at higher speeds and on longer roads than the ones considered in our experimental configuration (we divided the long roads generated by AsFault into segments to make them comparable to those generated by the other tools).

Misbehaviour sparseness metrics (see middle and bottom of Figure 11) show a similar trend. For `MLP-TurnCnt`, DeepHyperion-CS generated significantly sparser misbehaviours than all the other tools considering Misbehaviour Sparseness (Avg. Max), whereas it has a comparable misbehaviour sparseness to DeepJanus-BB in terms of Misbehaviour Sparseness (Avg.)

For `StdSA-Curv` and `MLP-StdSA`, DeepHyperion-CS's sparseness is higher than AsFault, comparable to DeepHyperion and DeepJanus-WB, but significantly lower than DeepJanus-BB for both misbehaviour sparseness metrics.

This result can be explained by considering the relatively small number of mapped misbehaviours reported by DeepJanus-BB and their distribution on distant places of the subject's behavioural frontier, which inflate the resulting Misbehaviour Sparseness metric.

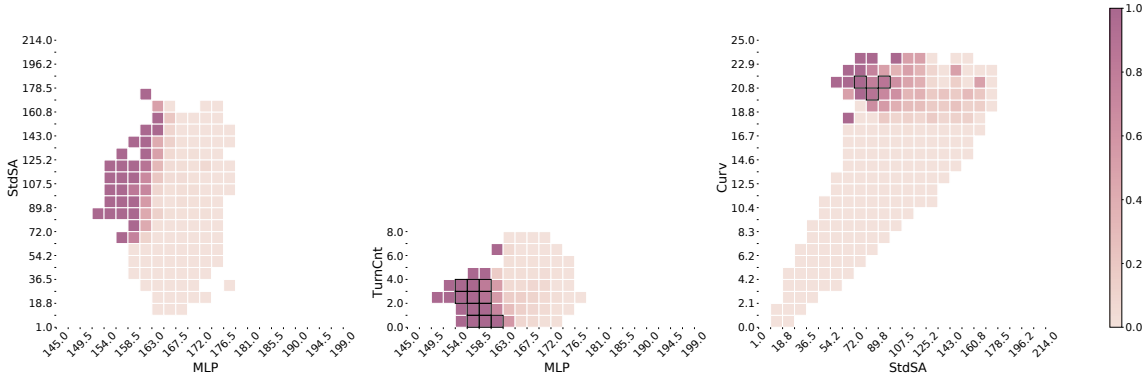
The misbehaviour probability maps of DeepHyperion (Figure 12a) and DeepHyperion-CS (Figure 12b) show almost the same pattern. The probability maps show also that both tools were able to trigger different test outcomes even when the same behavioural feature is exhibited by the driving agent: in the leftmost maps, the same values of the standard deviation of steering angle (StdSA) feature may or may not trigger misbehaviours, depending on the value of the mean lateral position feature.

> ***Summary:*** *DeepHyperion-CS can find diverse misbehaviour-inducing inputs for all feature combinations, detecting up to* 100 *more misbehaviours than its best competitor on MNIST. The guidance offered by contribution score significantly improved the DeepHyperion's effectiveness for* 5 *out of* 6 *feature combinations across both test subjects.*

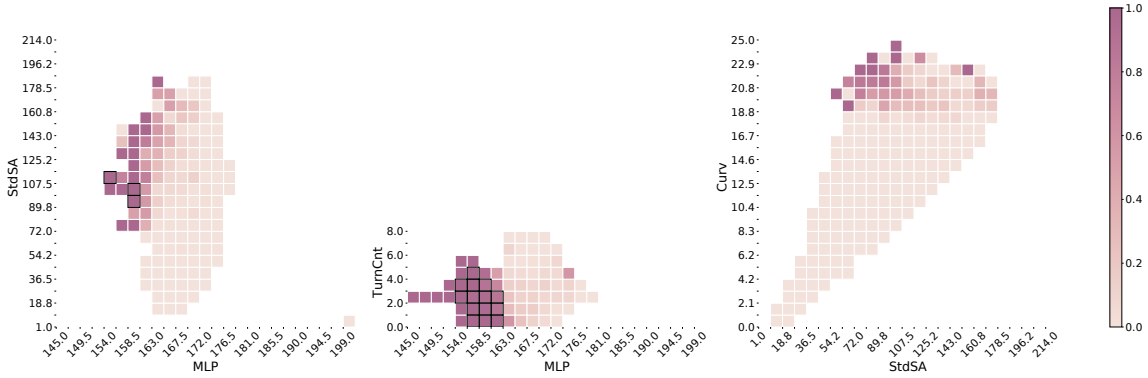
5.2 RQ2. Does the guidance provided by Contribution Score allow DeepHyperion-CS to explore the feature space more extensively than DeepHyperion?

RQ2 investigates the generated tests' adequacy in terms of their feature map coverage (i.e., *Filled Cells*) and diversity (i.e., *Coverage Sparseness*). Figure 13 reports the results for MNIST as box plots grouped by feature combination.

For DeepJanus and DLFuzz, we report the cells filled by the inputs returned at the end of the run (i.e., black box) and the number of all the cells filled during the same run (i.e., white box) as two separate boxes. DeepJanus finds pairs



(a) DeepHyperion



(b) DeepHyperion-CS

Fig. 12. Misbehaviour probability maps generated by DeepHyperion (a) and DeepHyperion-CS (b) for BeamNG

of similar inputs that trigger different behaviours (expected vs misbehaviours); since inputs within pairs are likely to occupy the same cell, DeepJanus's Filled Cells values are close to the corresponding Mapped Misbehaviours reported in Figure 9. DLFuzz-BB shows the same values reported for DLFuzz in Figure 9 since this tool returns only misbehaviours. Instead, DLFuzz-WB has higher values because it includes also the correctly behaving inputs produced during each run.

Figure 13 (top) shows that DeepHyperion-CS covered all feature maps significantly more extensively than the other tools (with p-values < 0.05 and effect size which is small for `Mov-Lum` and large for `Or-Lum` and `Or-Mov`).

Similarly to RQ1, DeepHyperion-CS produced the best results for the `Or-Mov` feature combination, almost doubling the coverage achieved by DeepHyperion and tripling the one achieved by DeepJanus-WB.

The sparseness metrics reported in Figure 13 show that DeepHyperion-CS produced significantly sparser inputs than all the other tools for two feature combinations out of three (i.e., `Or-Lum` and `Or-Mov`) with p-values < 0.05 and large effect size. For `Mov-Lum`, DeepHyperion-CS achieved a level of coverage sparseness comparable to DeepHyperion (p-values > 0.05), but significantly higher than all the other tools.

Figure 14 reports the Filled Cells and the Coverage Sparseness achieved by the tools on the BeamNG subject system. Figure 14 (top) shows that DeepHyperion-CS was again particularly good in covering the feature maps. In particular, for the `StdSA-Curv` feature combination, it achieved significantly higher coverage than DeepHyperion (p-value < 0.05,

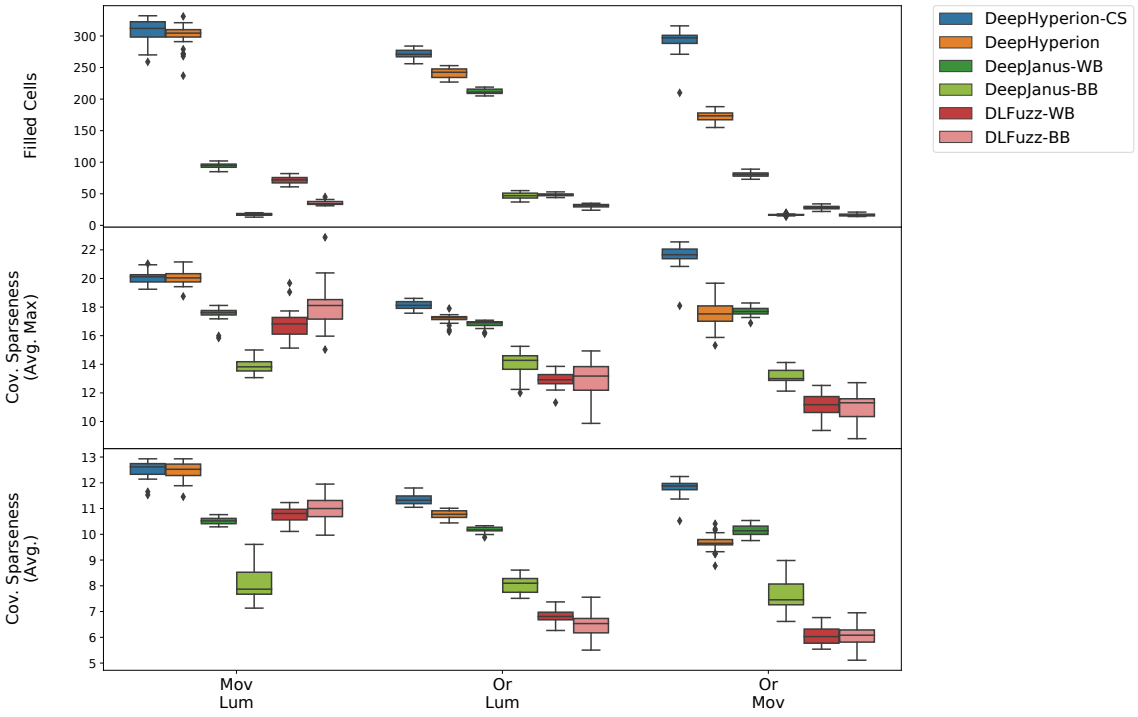


Fig. 13. RQ2: Filled cells (top) and coverage sparseness (middle and bottom) achieved by the considered tools on MNIST.

medium effect size). For the other feature combinations, it behaved comparably to DeepHyperion (p-values > 0.05) but filled significantly more cells than all the other tools (p-values < 0.05, large effect size).

As shown in, Figure 14 both sparseness metrics show that DeepHyperion-CS produced tests that are significantly sparser than DeepJanus-WB, comparably sparse as DeepHyperion, but less sparse than DeepJanus-BB and AsFault.

This result is due to the low feature map coverage achieved by DeepJanus-BB and AsFault, which amplifies the relative sparseness of the (few) filled cells.

> ***Summary:*** *Our illumination based test generators (i.e., DeepHyperion-CS and DeepHyperion) always explored the feature space more extensively than the other tools (up to 3× more for MNIST). The guidance provided by Contribution Score allowed DeepHyperion-CS to fill significantly more cells than DeepHyperion for the vast majority of feature combinations (i.e. 4 out of 6).*

5.3 RQ3. How efficient is DeepHyperion-CS in exploring the feature space and generating test inputs that expose diverse misbehaviours?

RQ3 investigates DeepHyperion-CS's efficiency by analysing the cumulative mapped misbehaviours (Figures 15a and 16a) and filled cells (Figures 15b and 16b) for MNIST and BeamNG. We visualise the evolution of mapped misbehaviours and

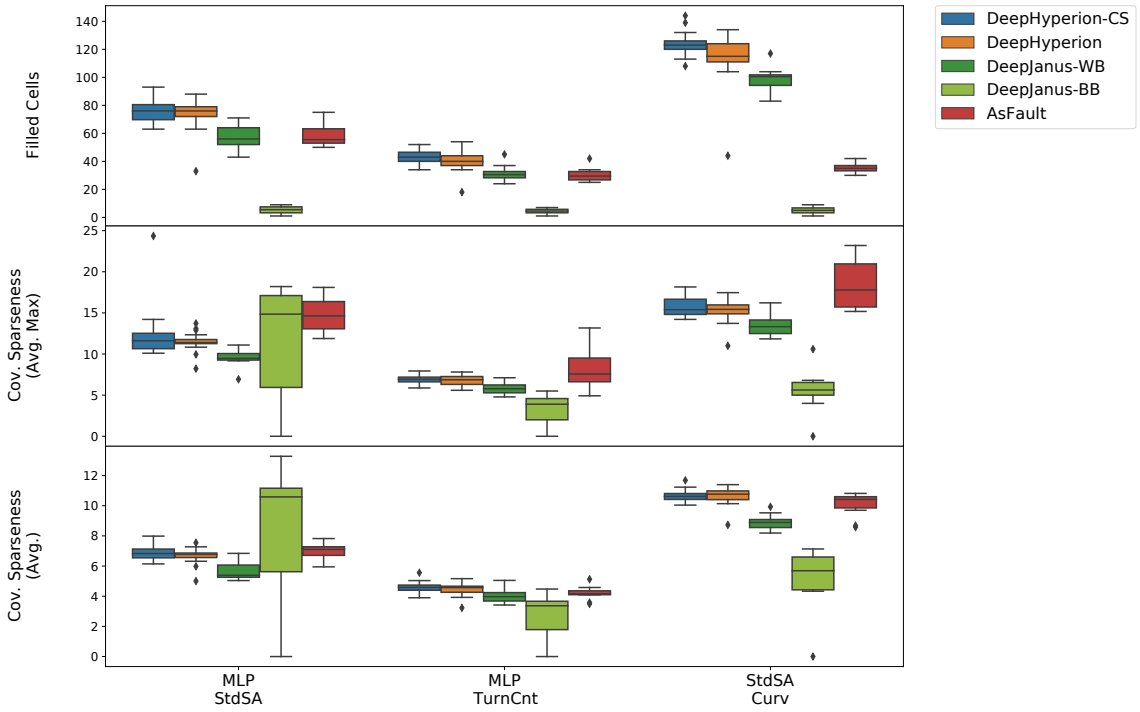


Fig. 14. RQ2: Filled cells (top) and coverage sparseness (middle and bottom) achieved by the considered tools on BeamNG.

filled cells throughout the runs by plotting their average values over time as solid lines, surrounded by their standard deviation as transparent shade.

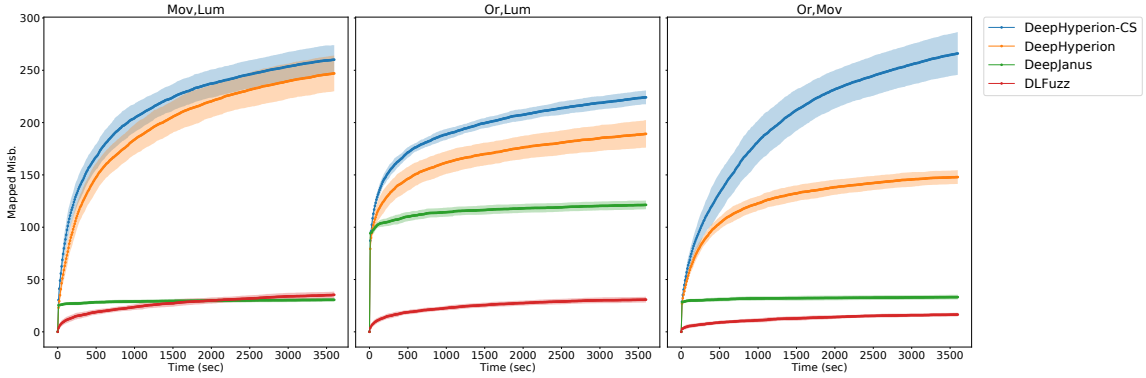
We consider the AUC for mapped misbehaviours and filled cells to quantitatively compare the considered tools' efficiency, i.e., the larger the AUC, the faster the metric increases to high values. For this RQ, we did not consider white box and black box performance separately since we can measure the evolution over time only when white box information is collected during the run.

For MNIST, Figure 15 shows that DeepHyperion-CS achieved a significantly greater AUC than all the other tools for both metrics (p-values < 0.05 and large effect size, with the only exception of mapped misbehaviours AUC for `Mov-Lum` vs DeepHyperion, in which the effect size is medium).

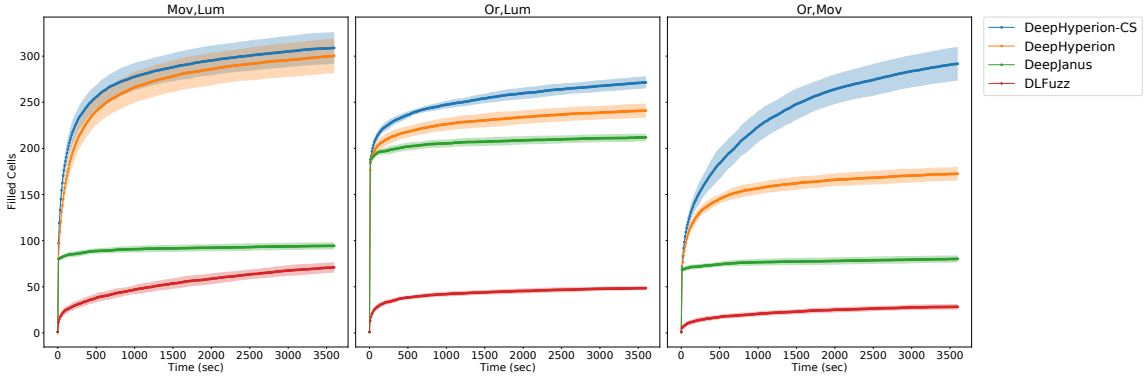
In particular, for `Or-Mov`, DeepHyperion-CS produced a higher number of mapped misbehaviours in remarkably less time than the other tools (AUC for Mapped Misbehaviours is 65% larger than the second best).

Figure 15 also shows that it took a small part of the 1-hour budget (i.e., less than three minutes) for DeepHyperion-CS and DeepHyperion to outperform the other tools, by generating a significantly higher number of misbehaviours and filled cells for all feature combinations (p-values < 0.05 and large effect size).

By comparing the results achieved over time by DeepHyperion-CS and DeepHyperion, we can notice that DeepHyperion-CS dramatically outperformed DeepHyperion after only three minutes for `Or-Lum` and eight minutes for `Or-Mov`, whereas they followed a similar trend for `Mov-Lum`.



(a) Mapped Misbehaviours over time for DeepHyperion-CS, DeepHyperion, DeepJanus and DLFuzz on MNIST



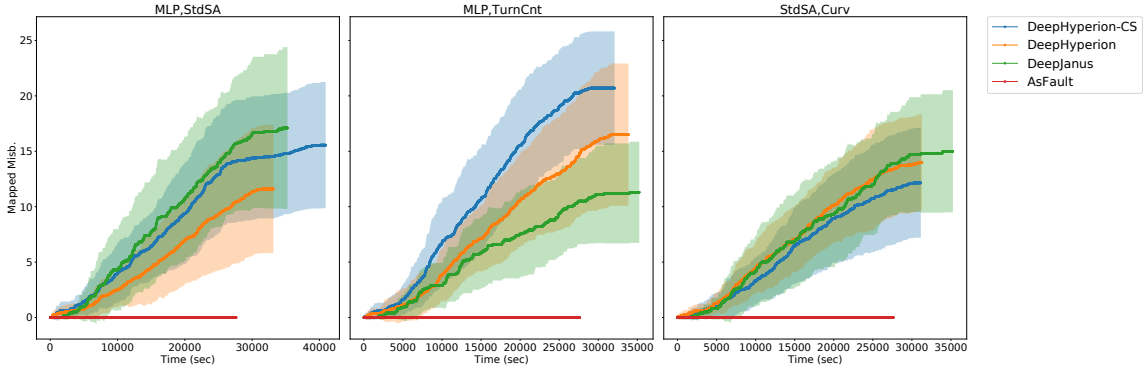
(b) Filled Cells over time for DeepHyperion-CS, DeepHyperion, DeepJanus and DLFuzz on MNIST

Fig. 15. RQ3: Filled Cells and Mapped Misbehaviours over time for DeepHyperion-CS, DeepHyperion, DeepJanus and DLFuzz on MNIST (shadows indicate standard deviations for each tool)

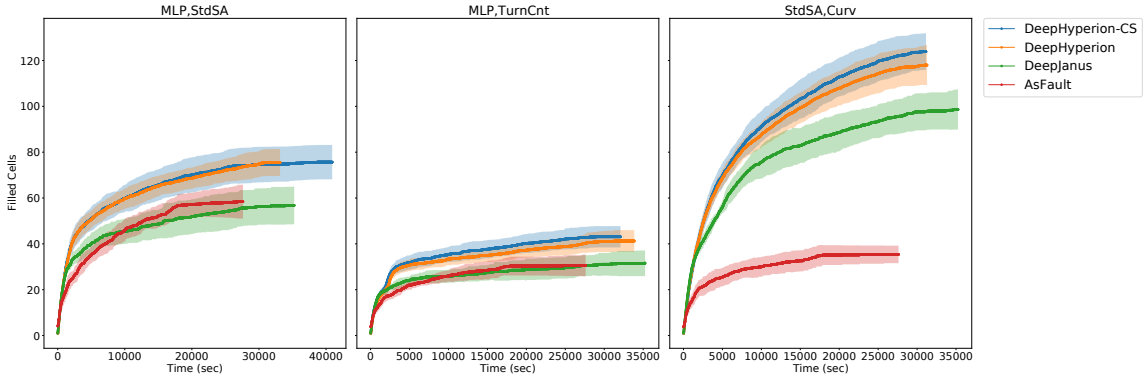
These results confirm that DeepHyperion-CS is extremely efficient from the very beginning of the search process in exploring the feature space and exposing diverse misbehaviours in MNIST. Moreover, DeepHyperion-CS kept discovering new cells, although at a lower pace as time progresses, which suggests that it did not reach saturation within the given time budget.

Figure 16 shows the evolution of mapped misbehaviours and filled cells for BeamNG. While we assigned all tools a budget of 10 hours of *simulation* time, in this RQ we are interested in assessing their practical efficiency and, thus, we compute the evolution of the considered metrics over *real* time. As a consequence, the results span across different time ranges for each tool. To guarantee a fair comparison, we compute the AUC by considering the minimum run time of all the tools.

Figure 16a shows that DeepHyperion-CS was significantly more efficient in finding diverse misbehaviours than all the other tools for `MLP-TurnCnt` (higher AUC, with p-values < 0.05, large effect size), while it never performed worse than the competitors for the other feature combinations: for `StdSA-Curv`, DeepHyperion-CS achieved AUC of mapped misbehaviours comparable to DeepHyperion and DeepJanus, and significantly better than AsFault; whereas only DeepJanus was as efficient as DeepHyperion-CS for `MLP-StdSA`.



(a) Mapped Misbehaviours over time for DeepHyperion-CS, DeepHyperion and DeepJanus on BeamNG



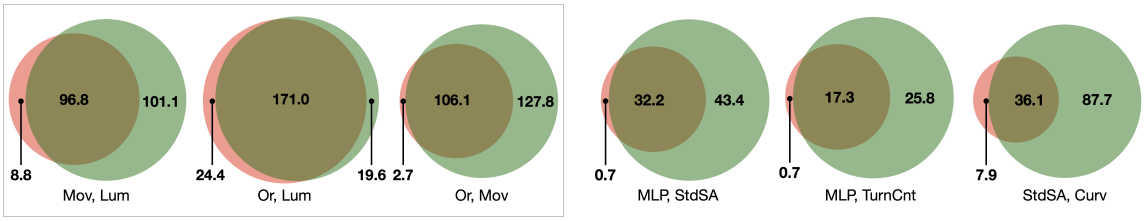
(b) Filled Cells over time for DeepHyperion-CS, DeepHyperion, DeepJanus and AsFault on BeamNG

Fig. 16. RQ3: Filled Cells and Mapped Misbehaviours over time for DeepHyperion-CS, DeepHyperion, DeepJanus and AsFault on BeamNG (shadows indicate standard deviations for each tool)

As regards AUC of filled cells, Figure 16b shows that DeepHyperion and DeepHyperion-CS covered the feature maps significantly more efficiently than DeepJanus and AsFault for all feature combinations (p-values < 0.05 and large effect size).

In particular, DeepHyperion-CS showed significantly better efficiency in filling cells than DeepHyperion for `MLP-TurnCnt` (p-value < 0.05, medium effect size), whereas, they achieved comparable efficiency for the other two feature combinations (p-values > 0.05).

Summary: *DeepHyperion-CS was extremely efficient, as it increasingly explored the feature space throughout the time budget and it found misbehaviours within the first few minutes of exploration. DeepHyperion-CS was always significantly more efficient than the competitors on MNIST. On BeamNG, DeepHyperion-CS showed either significantly higher or comparable efficiency in comparison with the other tools. The guidance of the contribution score remarkably improved efficiency (in 9 out of 12 comparisons against DeepHyperion).*



(a) MNIST (b) BeamNG

Fig. 17. RQ4: Average Filled Cells Expansion achieved by DeepHyperion-CS (green/right circles) over the training set (red/left circles).

Table 3. RQ4: Mapped misbehaviour expansion achieved by DeepHyperion-CS over the training set

Subject	Feature Combination	Mapped Misbehaviour Expansion	
		MME_{uncov}	MME_{cov}
MNIST	(Mov, Lum)	101.1 ± 12.0	60.9 ± 3.4
	(Or, Lum)	19.7 ± 4.9	19.6 ± 2.4
	(Or, Mov)	127.8 ± 16.5	48.3 ± 2.2
BeamNG	(MLP, StdSA)	12.4 ± 5.1	3.1 ± 2.1
	(MLP, TurnCnt)	11.2 ± 3.9	9.5 ± 2.6
	(StdSA, Curv)	12.1 ± 4.9	0.0 ± 0.0

5.4 RQ4. Can DeepHyperion-CS be used to expand the training data? Can it find misbehaving inputs also in cells that were already occupied by non-misbehaving training data?

RQ4 studies the relationship between the data used for training the DL system under test and the test inputs generated by DeepHyperion-CS, by identifying features that were under-represented in the training set or were not associated with any misbehaviour.

Venn diagrams in Figure 17a and Figure 17b illustrate the Filled Cell Expansion achieved by DeepHyperion-CS over the training set for each feature combination. The red circles (left) represent the cells filled by the training set, the green circles (right) represent the ones filled by DeepHyperion-CS's generated inputs, and the overlapping region represent cells that are covered by both.

DeepHyperion-CS achieved a remarkable filled cell expansion for the BeamNG system (see Figure 17b), where not only it filled most cells already covered by training data, but it also explored new uncovered regions in the feature maps, especially for the `MLP-StdSA` and `MLP-TurnCnt` feature combinations.

Figure 17a shows that DeepHyperion-CS was also able to improve the MNIST initial training set by adding samples that better cover some feature combinations (see, e.g., the Venn diagram for `Mov-Lum`). This task was not trivial since the MNIST training set by LeCun et al. [42] has been carefully crafted to be representative of its domain.

Table 3 summarises the average mapped misbehaviours expansion achieved by DeepHyperion-CS. Specifically, column MME_{uncov} reports the mapped misbehaviours that DeepHyperion-CS found in cells that were not covered by the training set, while column MME_{cov} reports the mapped misbehaviours found by DeepHyperion-CS in cells already covered by correctly behaving training inputs.

The results show that DeepHyperion-CS was always able to find new misbehaviour-inducing feature combination values in cells that were either uncovered or already covered by the training set. In particular, DeepHyperion-CS found misbehaviours in cells that did not expose any issue in the training set for 5 out of 6 feature combinations.

Each MME_{uncov} value is generally higher than the corresponding MME_{cov} for the same feature combination. This indicates that cells covered by correctly behaving training inputs could still trigger misbehaviours, but such misbehaviours are harder to expose.

> ***Summary:*** *DeepHyperion-CS was able to expand the initial training data for all feature combinations, achieving up to 200% more filled cells for the* `StdSA-Curv` *feature combination. Moreover, DeepHyperion-CS not only found new misbehaviour-inducing feature combination values in cells that were uncovered by the training set, but often also in covered ones.*

5.5 Threats to Validity

Construct Validity: the performance of the proposed approach and the quality of its results depend on the selected features and the procedures to quantify them. For instance, there is the risk that the adopted metrics do not accurately quantify the selected features. Moreover, the relevance of the features depends on the assessors' knowledge of the domain and the representativeness of the data used to extract the features. To mitigate this threat, we followed a systematic procedure to identify relevant features and utilised a well established statistical correlation analysis to check that the adopted metrics can quantify them. In particular, this procedure involved assessors that are experts on testing DL systems and relied on large dataset of 630 images for MNIST and 440 virtual roads for BeamNG.

Internal Validity: our main focus in this work was assessing the impact of the Contribution Score guidance on DeepHyperion's efficiency and effectiveness. To limit as much as possible the chance that the comparison between DeepHyperion and DeepHyperion-CS was affected by other con-causes, we used the same code base for both tools. As a result, we provide a unique framework in which the users can control the level of contribution-based selection pressure on the overall search process, i.e., the users can easily use the original DeepHyperion by setting the hyper-parameter controlling the probability of using the contribution score selection to 0. A threat that could affect the experimental comparison against existing test input generators is that their purpose is different from illuminating the feature space. Therefore, their output may contain only the most critical inputs but exclude interesting inputs found during their runs. We addressed this threat by considering also the inputs generated (and possibly discarded) during the input generation process, in addition to the ones reported as final result by the tools. Furthermore, the Open Coding's internal validity may be threatened by elements that may introduce inconsistencies in the assessors' evaluations independently of the data, such as the data order and the repetitiveness of the task. To mitigate this threat, we conducted a pilot study in which multiple assessors evaluated the same images, presented in a randomised order. Moreover, we granted the assessors a generous time budget to perform this task (i.e., one month) through our Web application, which allowed them to interrupt the task whenever they felt tired and resume it later on.

Reproducibility of our results is ensured by the online availability of the source code of DeepHyperion-CS, our objects, and the experimental data. Moreover, we considered only open source tools in our experimental comparison.

External Validity: The choice of subject DL systems is a possible threat to the external validity. To mitigate this threat, we chose two DL systems which solve two different problems, i.e., MNIST solves a classification problem, while BeamNG is a self-driving car software that solves a regression problem. We considered DL architectures which are

widely used in the literature and regarded as state of the art. Moreover, we adopted standard training procedures and validated them with standard performance metrics, i.e. classification accuracy and mean squared error. In comparison to the original DeepHyperion's experimental setup [73], in this work we extend the generalisability of the results by considering another state-of-the-art test generator, i.e., AsFault. The choice of relevant features introduces another threat to external validity as DeepHyperion might not identify misbehaviours that do not align with the selected features. Further studies involving more test subjects and domain experts, including practitioners from industry, should be carried out to fully assess the generalisability of our findings and the impact of the feature selection.

Conclusion Validity: Random variations might have affected the results, given the highly stochastic nature of both DL systems and the considered test generators. We mitigated this threat by following the widely adopted guidelines for comparing randomised test generation algorithms proposed by Arcuri and Briand [4]. In particular, we used a generous budget (i.e., multiple, long runs) and assessed the results' significance through standard statistical tests.

6 RELATED WORK

In the software engineering literature, DL systems' quality is mostly assessed by automatically generating new inputs that expose misbehaviours [22, 26, 47, 64]. Input generation for DL systems is often guided by novel test adequacy criteria which are either adapted from traditional software testing literature (e.g., coverage, combinatorial, mutant killing, number of triggered misbehaviours) or specifically crafted for DL (e.g., uncertainty or surprise) [58]. Only few works [2, 59] make use of interpretable properties. To the best of our knowledge, besides DeepHyperion [73] and DeepHyperion-CS, no technique aims at covering the feature space of DL systems.

6.1 Exploration-driven test generation

Traditional evolutionary search algorithms leverage an *exploitation* mechanism. This mechanism consists of rewarding inputs with higher values of one or more fitness functions, which provide a heuristic distance of each candidate solution from the searched optimum. Exploitation might return solutions which are concentrated in a small portion of the input space, especially when the search landscape includes local optima with a large basin of attraction. In software testing, finding good failure-inducing solutions is often insufficient, since testers may be interested in finding solutions spread across the entire input space, as those could reveal different faults of the software under test.

Exploration-driven search algorithms reward individuals that exhibit diversity from the previous solutions, instead of promoting only those that contribute to progress toward the optimum [43]. Exploration-driven algorithms, including Novelty Search [43] and Viability Evolution [48], require some definition of input diversity, which is domain-specific and far from trivial. Existing works in the software testing literature propose metrics to compute test case diversity in terms of structural features, e.g., Euclidean distance between input vectors [11] and normalized compression distance [19], or behavioural features, such as the output uniqueness proposed by Alshawan and Harman [3]. In this work, we adopted both structural and behavioural input features. Exploration-driven search proved to be a viable approach to software testing problems such as Web testing [8].

The combination of exploration and exploitation can be beneficial for evolutionary search, as shown for Android app testing by Vogel et al. [66]. MAP-Elites by Mouret and Clune [52] balances exploration across the whole input space with local competition between similar inputs. Other algorithms that combine exploration and exploitation are Novelty Search plus Local Competition (NS+LC) [45] and Multi-Objective Landscape Exploration (MOLE) [15]. Marculescu et al. [50] showed that MAP-Elites has higher exploration power, i.e., ability to investigate different areas of the feature space, than other exploration and exploitation approaches, for testing a clustering algorithm.

As for testing DL systems, Riccio and Tonella combined the traditional, exploitation-driven NSGA-II algorithm with an exploration mechanism that promotes structural diversity of solutions [59]. DeepHyperion [73], which adapts the MAP-Elites algorithm to test DL systems, proved to outperform the state-of-the-art approaches in extensively exploring the feature space of a DL system and finding failure-inducing inputs that are different in terms of structural and behavioural features. In this work, we improve DeepHyperion by implementing a mechanism which promotes the inputs that contributed more to feature space exploration during the previous search iterations.

6.2 Test Input Generation and Adequacy

Traditional test adequacy criteria, such as code coverage, cannot assess whether DL systems are adequately exercised by a test set. In fact, DL systems' behaviour mostly depends on their training data, architecture and the tuning of several hyper-parameters, rather than the code. Recent research defined ad-hoc test adequacy metrics for DL systems and proposed input generation techniques guided by these metrics. Pei et al. [54] defined neuron coverage, an adequacy criterion that measures the percentage of neurons whose activation level exceeds a certain threshold during testing. They also designed DeepXplore, a test generator guided by neuron coverage to detect behaviour inconsistencies between different DNNs. Several other approaches extended neuron coverage [47] or used it to drive test generation [16, 17, 26, 47, 64, 68]. DeepCT [46], instead, uses a set of combinatorial testing criteria for DNNs based on the interactions between neurons.

The test input generation approaches listed above generate adversarial inputs by adding small perturbations to the original inputs. Adversarial input perturbations are widely used by the Machine Learning community to affect model's predictions [9]. Our approach, instead, is based on model-based manipulations, which ensure at the same time more diversity and more control on the realism of the generated inputs. Moreover, we perform testing at the functional level and we are interested in diversified feature combinations that trigger misbehaviours, while adversarial attacks expose security and robustness vulnerabilities without promoting the spread of such attacks in the feature space.

Mutation adequacy criteria assess the ability of test data to expose artificially injected faults that simulate real faults (i.e., mutations). A notable work in this area is DeepCrime [34], a mutation tool built on top of the notion of statistical killing proposed by Jahangirova and Tonella [36], which injects mutations resembling real DL fault classes, such as the ones defined in the taxonomy by Humbatova et al. [33]. DeepMetis [57] is a search-based approach that generates test inputs that behave correctly on original models and misbehave on mutants, in order to increase the mutation killing ability of a test set. Vahdat Pour el al. [55] generate adversarial examples for source code processing DNNs, guided by the mutation killing criterion defined by Hu et al. [32].

Other input generators such as AsFault [22] and NSGAII-DT [2] aim to test advanced driver-assistance systems by generating extreme and challenging scenarios that maximise the number of exposed system failures. Ul Haq et al. [27] generate inputs to trigger diverse and extreme misbehaviours of DNN-based facial key-point predictors using many-objective search in order to cover as many failure-inducing key-points as possible.

X. Zhang et al. [71] proposed an approach for generating test inputs with diverse uncertainty patterns (i.e., prediction confidence score and variation ratio). They do not define adequacy criteria, but they suggest to generate test inputs that target the least frequently covered uncertainty patterns.

Kim et al. [38] measure the degree of "surprise" of test inputs with respect to the training set with their adequacy criterion named surprise adequacy. In their work, this criterion was used for test case selection and retraining, not for test input generation.

All the test generators we mentioned above aim at maximising some adequacy metric, such as neuron coverage, or at exposing misbehaviours. None of them considers the value combinations of interpretable features of the DL

system under test as the target of test generation. Hence, existing test generators might completely ignore parts of a feature map or might expose only misbehaviours that belong to a narrow map region. Our work is the first to provide developers with a map of such features, where the generated inputs, and exposed misbehaviours can be interpreted based on their position in the map.

6.3 Structural and Behavioural Properties

NSGAII-DT [2] is a testing approach targeting vision-based control systems. This approach builds on evolutionary multi-objective algorithms and uses decision trees to guide the generation of new test scenarios within the multi-dimensional space of the model parameters. Decision trees are used to identify the critical regions of the input space, i.e., the combinations of model parameter values that are more likely to cause misbehaviours. Decision trees provide interpretable information to developers as DeepHyperion-CS does with its feature maps. The variables that appear in the decision tree nodes are the control parameters of the input scenarios. Decision trees increase the search efficiency by focusing the search towards critical scenarios (collisions or near-collision at high speed with pedestrians). Instead, we aim at covering the feature map at large, so as to ensure that as many regions as possible are tested and that regions with misbehaviours are not left untested. Therefore, DeepHyperion-CS promotes inputs that contribute most to exploration, in order to visit the feature space at large. At the same time, it uses local competition to keep in the map only the fittest individuals, which can potentially lead to misbehaviour exposure.

DeepJanus [59] characterises a DL system's quality as its frontier of behaviours, i.e., pairs of similar inputs that trigger different (correct vs failing) behaviours of the system. DeepJanus provides users with a set of system's frontier inputs, but it does not explicitly characterise them based on structural or behavioural features. On the contrary, DeepHyperion-CS produces maps that allow developers to interpret the misbehaviour-inducing inputs in terms of their feature values.

The properties we use as feature dimensions are identified by experts during the open coding step of our feature selection methodology. In the literature, weak supervision approaches [72], e.g., the Data Programming paradigm [56], exploit domain-experts' knowledge to create and assign output labels to the training set elements. Unlike these approaches, our open coding identifies input features that can be quantified by metrics, without considering their relationship with the network's expected output (i.e., the labels).

7 CONCLUSIONS AND FUTURE WORK

Most testing approaches for DL systems can successfully find misbehaviour-inducing inputs but fail to clearly explain what are the structural and behavioural features that negatively influenced the system's behaviour. DeepHyperion is the first test generator that overcomes this limitation by providing an interpretable characterisation of DL systems' quality through maps which represent the generated inputs in the space of their relevant features (i.e., the feature space). DeepHyperion has demonstrated to be able to explore the DL systems' feature space at large and trigger diverse misbehaviours thanks to its illumination search based algorithm.

In this work, we proposed DeepHyperion-CS, a novel test generator for DL systems which enhances DeepHyperion by promoting the inputs that contribute more to the feature map exploration during the search. Our empirical study showed that the contribution-based guidance implemented within DeepHyperion-CS significantly improves the efficiency and the effectiveness of DeepHyperion in finding misbehaviour-inducing inputs and exploring the feature space. Moreover, we provided evidence that the inputs generated by DeepHyperion-CS can be also useful for characterising and expanding the datasets used to train the DL system.

Despite the remarkable results achieved so far, our work on testing DL systems using feature maps and illumination search is at its dawn and opens many interesting directions for future research. As an example, we will design efficient search strategies specific to higher dimensionality feature spaces, i.e., maps with more than two dimensions. We also plan to apply the feature maps produced by DeepHyperion-CS to practical DL software development tasks. In fact, we believe they are an intuitive approach for evaluating test set adequacy or guiding test selection. To extend the applicability of DeepHyperion-CS to domains where an input model is not available, e.g., object detection with complex real-world images, we will investigate GAN-based input representations and mutation operators.

Finally, since DeepHyperion-CS is designed to help DL developers interpret their software, we plan to generalise our results to industrial DL systems and assess with practitioners the understandability and usefulness of feature maps.

ACKNOWLEDGMENTS

We thank the editor and the referees for their helpful comments during the review process. This work was partially supported by the H2020 project PRECRIME, funded under the ERC Advanced Grant 2017 Program (ERC Grant Agreement n. 787703), and the DFG project STUNT (DFG Grant Agreement n. FR 2955/4-1). The driving simulator has been kindly provided by BeamNG GmbH.

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* (Gothenburg, Sweden) *(ICSE '18)*. ACM, 1016–1026. https://doi.org/10.1145/3180155.3180160

[3] Nadia Alshahwan and Mark Harman. 2012. Augmenting test suites effectiveness by increasing output diversity. In *2012 34th International Conference on Software Engineering (ICSE)*. 1345–1348. https://doi.org/10.1109/ICSE.2012.6227083

[4] 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

[5] Andrea Arcuri and Gordon Fraser. 2011. On Parameter Tuning in Search Based Software Engineering. In *Search Based Software Engineering - Third International Symposium, SSBSE 2011, Szeged, Hungary, September 10-12, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6956)*, Myra B. Cohen and Mel Ó Cinnéide (Eds.). Springer, 33–47. https://doi.org/10.1007/978-3-642-23716-4_6

[6] T. Back. 1994. Selective pressure in evolutionary algorithms: a characterization of selection mechanisms. In *Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence*. 57–62 vol.1. https://doi.org/10.1109/ICEC.1994.350042

[7] BeamNG GmbH. 2018. *BeamNG.research*. BeamNG GmbH. https://www.beamng.gmbh/research

[8] Matteo Biagiola, Andrea Stocco, Filippo Ricca, and Paolo Tonella. 2019. Diversity-Based Web Test Generation. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 142–153. https://doi.org/10.1145/3338906.3338970

[9] Battista Biggio and Fabio Roli. 2018. Wild patterns: Ten years after the rise of adversarial machine learning. *Pattern Recognition* 84 (2018), 317–331.

[10] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. 2016. End to End Learning for Self-Driving Cars. *CoRR* abs/1604.07316 (2016), 1–9. arXiv:1604.07316 http://arxiv.org/abs/1604.07316

[11] Paulo M. S. Bueno, W. Eric Wong, and Mario Jino. 2007. Improving Random Test Sets Using the Diversity Oriented Test Data Generation. In *Proceedings of the 2nd International Workshop on Random Testing: Co-Located with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)* (Atlanta, Georgia) *(RT '07)*. Association for Computing Machinery, New York, NY, USA, 10–17. https://doi.org/10.1145/1292414.1292419

[12] N. Carlini and D. Wagner. 2017. Towards Evaluating the Robustness of Neural Networks. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 39–57. https://doi.org/10.1109/SP.2017.49

[13] Edwin Catmull and Raphael Rom. 1974. A Class of Local Interpolating Splines. In *Computer Aided Geometric Design*, R. E. Barnhill and R. F. Riesenfeld (Eds.). Academic Press, 317 – 326. https://doi.org/10.1016/B978-0-12-079050-0.50020-5

[14] François Chollet. 2020. Simple MNIST convnet. https://github.com/keras-team/keras-io/blob/master/examples/vision/mnist_convnet.py.

[15] Jeff Clune, Jean-Baptiste Mouret, and Hod Lipson. 2012. The evolutionary origins of modularity. *CoRR* abs/1207.2743 (2012). arXiv:1207.2743 http://arxiv.org/abs/1207.2743

[16] 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

[17] 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

[18] 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

[19] Robert Feldt, Simon Poulding, David Clark, and Shin Yoo. 2016. Test Set Diameter: Quantifying the Diversity of Sets of Test Cases. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 223–233. https://doi.org/10.1109/ICST.2016.33

[20] Gordon Fraser and Andrea Arcuri. 2013. Whole Test Suite Generation. *IEEE Transactions on Software Engineering* 39, 2 (2013), 276–291. https://doi.org/10.1109/TSE.2012.14

[21] Gordon Fraser, Andrea Arcuri, and Phil McMinn. 2015. A Memetic Algorithm for whole test suite generation. *Journal of Systems and Software* 103 (2015), 311–327. https://doi.org/10.1016/j.jss.2014.05.032

[22] Alessio Gambi, Marc Müller, and Gordon Fraser. 2019. Automatically testing self-driving cars with search-based procedural content generation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*. ACM, 318–328. https://doi.org/10.1145/3293882.3330566

[23] Carlo Ghezzi. 2020. *Being a Researcher - An Informatics Perspective*. Springer. https://doi.org/10.1007/978-3-030-45157-8

[24] David E. Goldberg and Kalyanmoy Deb. 1991. A Comparative Analysis of Selection Schemes Used in Genetic Algorithms. Foundations of Genetic Algorithms, Vol. 1. Elsevier, 69–93. https://doi.org/10.1016/B978-0-08-050684-5.50008-2

[25] Ian J. Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

[26] 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

[27] Fitash Ul Haq, Donghwan Shin, Lionel C. Briand, Thomas Stifter, and Jun Wang. 2020. Automatic Test Suite Generation for Key-points Detection DNNs Using Many-Objective Search. arXiv:2012.06511 [cs.CV]

[28] Mark Harman and Phil McMinn. 2010. A Theoretical and Empirical Study of Search-Based Testing: Local, Global, and Hybrid Search. *IEEE Transactions on Software Engineering* 36, 2 (2010), 226–247. https://doi.org/10.1109/TSE.2009.71

[29] Mark Harman, Phil McMinn, Jerffeson Teixeira de Souza, and Shin Yoo. 2010. Search Based Software Engineering: Techniques, Taxonomy, Tutorial. In *Empirical Software Engineering and Verification - International Summer Schools, LASER 2008-2010, Elba Island, Italy, Revised Tutorial Lectures (Lecture Notes in Computer Science, Vol. 7007)*, Bertrand Meyer and Martin Nordio (Eds.). Springer, 1–59. https://doi.org/10.1007/978-3-642-25231-0_1

[30] Florian Hauer, Alexander Pretschner, and Bernd Holzmüller. 2019. Fitness Functions for Testing Automated and Autonomous Driving Systems. In *Computer Safety, Reliability, and Security - 38th International Conference, SAFECOMP 2019, Turku, Finland, September 11-13, 2019, Proceedings*, Vol. 11698. Springer, 69–84. https://doi.org/10.1007/978-3-030-26601-1_5

[31] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[32] 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

[33] 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* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, 1110–1121. https://doi.org/10.1145/3377811.3380395

[34] 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*.

[35] Gunel Jahangirova, Andrea Stocco, and Paolo Tonella. 2021. Quality Metrics and Oracles for Autonomous Vehicles Testing. In *Proceedings of 14th IEEE International Conference on Software Testing, Verification and Validation (ICST '21)*. IEEE, 194–204.

[36] 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.

[37] Sungmin Kang, Robert Feldt, and Shin Yoo. 2020. SINVAD: Search-based Image Space Navigation for DNN Image Classifier Test Input Generation. In *ICSE '20: 42nd International Conference on Software Engineering, Workshops, Seoul, Republic of Korea, 27 June - 19 July, 2020*. ACM, 521–528. https://doi.org/10.1145/3387940.3391456

[38] 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

[39] Wilhelm Kirch (Ed.). 2008. *Pearson's Correlation Coefficient*. Springer Netherlands, 1090–1091. https://doi.org/10.1007/978-1-4020-5614-7_2569

[40] Eugene F Krause. 1986. *Taxicab geometry: An adventure in non-Euclidean geometry*. Courier Corporation.

[41] Craig Larman. 1997. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*. Prentice Hall.

[42] Yann LeCun, Léon Bottou, Yoshua Bengio, Patrick Haffner, et al. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.

[43] Joel Lehman and Kenneth O. Stanley. 2011. Abandoning Objectives: Evolution Through the Search for Novelty Alone. *Evolutionary Computation* 19, 2 (2011), 189–223. https://doi.org/10.1162/EVCO_a_00025

[44] Joel Lehman and Kenneth O. Stanley. 2011. Evolving a Diversity of Virtual Creatures Through Novelty Search and Local Competition. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation* (Dublin, Ireland) *(GECCO '11)*. ACM, 211–218. https://doi.org/10.1145/2001576.2001606

[45] Joel Lehman and Kenneth O. Stanley. 2011. Evolving a Diversity of Virtual Creatures through Novelty Search and Local Competition. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation* (Dublin, Ireland) *(GECCO '11)*. Association for Computing Machinery, New York, NY, USA, 211–218. https://doi.org/10.1145/2001576.2001606

[46] 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

[47] 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* (Montpellier, France) *(ASE 2018)*. ACM, 120–131. https://doi.org/10.1145/3238147.3238202

[48] Andrea Maesani, Giovanni Iacca, and Dario Floreano. 2016. Memetic Viability Evolution for Constrained Optimization. *IEEE Transactions on Evolutionary Computation* 20, 1 (2016), 125–144. https://doi.org/10.1109/TEVC.2015.2428292

[49] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. 2008. *Introduction to Information Retrieval*. Cambridge University Press.

[50] B. Marculescu, R. Feldt, and R. Torkar. 2016. Using Exploration Focused Techniques to Augment Search-Based Software Testing: An Experimental Evaluation. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 69–79. https://doi.org/10.1109/ICST.2016.26

[51] Phil McMinn. 2004. Search-based software test data generation: a survey. *Software testing, Verification and reliability* 14, 2 (2004), 105–156.

[52] Jean-Baptiste Mouret and Jeff Clune. 2015. Illuminating search spaces by mapping elites. arXiv:1504.04909 [cs.AI]

[53] Sebastiano Panichella, Alessio Gambi, Fiorella Zampetti, and Vincenzo Riccio. 2021. SBST Tool Competition 2021. In *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)*. 20–27. https://doi.org/10.1109/SBST52555.2021.00011

[54] 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

[55] 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.

[56] Alexander Ratner, Christopher De Sa, Sen Wu, Daniel Selsam, and Christopher Ré. 2016. Data Programming: Creating Large Training Sets, Quickly. In *Proceedings of the 30th International Conference on Neural Information Processing Systems* (Barcelona, Spain) *(NIPS'16)*. Curran Associates Inc., 3574–3582.

[57] Vincenzo Riccio, Nargiz Humbatova, Gunel Jahangirova, and Paolo Tonella. 2021. DeepMetis: Augmenting a Deep Learning Test Set to Increase its Mutation Score. *arXiv preprint arXiv:2109.07514* (2021).

[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] Carolyn B. Seaman. 1999. Qualitative Methods in Empirical Studies of Software Engineering. *IEEE Transactions on Software Engineering* 25 (1999), 557–572.

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

[62] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[63]]StoccoJSEP21 Andrea Stocco and Paolo Tonella. [n. d.]. Confidence-driven weighted retraining for predicting safety-critical failures in autonomous driving systems. *Journal of Software: Evolution and Process* n/a, n/a ([n. d.]), e2386. https://doi.org/10.1002/smr.2386 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.2386

[64] 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* (Gothenburg, Sweden) *(ICSE '18)*. ACM, 303–314. https://doi.org/10.1145/3180155.3180220

[65] 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.

[66] Thomas Vogel, Chinh Tran, and Lars Grunske. 2019. Does Diversity Improve the Test Suite Generation for Mobile Applications?. In *Search-Based Software Engineering - 11th International Symposium, SSBSE 2019, Tallinn, Estonia, August 31 - September 1, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11664)*. Springer, 58–74. https://doi.org/10.1007/978-3-030-27455-9_5

[67] L. Darrell Whitley. 1989. The GENITOR Algorithm and Selection Pressure: Why Rank-Based Allocation of Reproductive Trials is Best. In *Proceedings of the 3rd International Conference on Genetic Algorithms, George Mason University, Fairfax, Virginia, USA, June 1989*, J. David Schaffer (Ed.). Morgan Kaufmann, 116–123.

[68] 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* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, 146–157. https://doi.org/10.1145/3293882.3330579

[69] 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

[70] Mengshi Zhang, Yuqun Zhang, Lingming Zhang, Cong Liu, and Sarfraz Khurshid. 2018. DeepRoad: GAN-based metamorphic testing and input validation framework for autonomous driving systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*. ACM, 132–142. https://doi.org/10.1145/3238147.3238187

[71] 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.

[72] Zhi-Hua Zhou. 2017. A brief introduction to weakly supervised learning. *National Science Review* 5, 1 (08 2017), 44–53. https://doi.org/10.1093/nsr/nwx106 arXiv:https://academic.oup.com/nsr/article-pdf/5/1/44/31567770/nwx106.pdf

[73] 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*. 79–90.