DeepHyperion: Exploring the Feature Space of Deep Learning-Based Systems through Illumination Search

Tahereh Zohdinasab
Software Institute - USI
Lugano, Switzerland
tahereh.zohdinasab@usi.ch

Vincenzo Riccio
Software Institute - USI
Lugano, Switzerland
vincenzo.riccio@usi.ch

Alessio Gambi
University of Passau
Passau, Germany
alessio.gambi@uni-passau.de

Paolo Tonella
Software Institute - USI
Lugano, Switzerland
paolo.tonella@usi.ch

ABSTRACT

Deep Learning (DL) has been successfully applied to a wide range of application domains, including safety-critical ones. Several DL testing approaches have been recently proposed in the literature but none of them aims to assess how different interpretable features of the generated inputs affect the system's behaviour.

In this paper, we resort to Illumination Search to find the highest-performing test cases (i.e., misbehaving and closest to misbehaving), spread across the cells of a map representing the feature space of the system. We introduce a methodology that guides the users of our approach in the tasks of identifying and quantifying the dimensions of the feature space for a given domain. We developed DeepHyperion, a search-based tool for DL systems that illuminates, i.e., explores at large, the feature space, by providing developers with an interpretable feature map where automatically generated inputs are placed along with information about the exposed behaviours.

CCS CONCEPTS

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

KEYWORDS

software testing, deep learning, search based software engineering

ACM Reference Format:
Tahereh Zohdinasab, Vincenzo Riccio, Alessio Gambi, and Paolo Tonella. 2021. DeepHyperion: Exploring the Feature Space of Deep Learning-Based Systems through Illumination Search. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2021).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

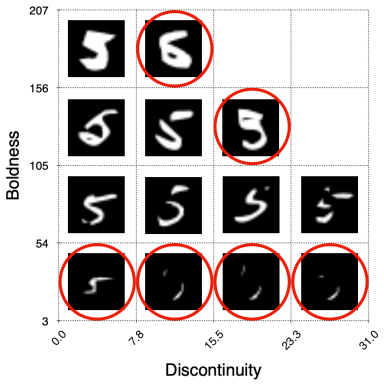

Figure 1: Feature map produced by DeepHyperion for a handwritten digit classifier. The two axes quantify two features: *discontinuity* and *boldness*. Cells show inputs that are either misclassified (marked with a circle) or close to being misclassified.

1 INTRODUCTION

Deep Learning (DL) has become an essential component of complex software systems, including autonomous vehicles and medical diagnosis systems. As a consequence, the problem of ensuring the dependability of DL systems is critical.

Unlike traditional software, in which developers explicitly program the system's behaviour, one peculiarity of DL systems is that they mimic the human ability to learn how to perform a task from training examples [22]. Therefore, it is essential to understand to what extent they can be trusted in response to the diversity of inputs they will process once deployed in the real world, as they could face scenarios that might be not sufficiently represented in the data from which they have learned.

As discussed in the comprehensive work by Riccio et al. [25] and by Zhang et al. [32], the Software Engineering research community is working hard at adequately testing the functionality of DL systems by proposing a steadily growing number of approaches. Since part of the program logic of these systems is determined by the training data, traditional code coverage metrics are not effective in determining whether their logic has been adequately exercised. Therefore, recent testing solutions aim at maximising ad hoc white-box adequacy metrics, such as neuron [13, 24, 29, 31] or surprise coverage [15], or at exposing misbehaviours [1, 11, 33]. A limitation of these approaches is that their output cannot be directly used to explain the behaviour of the DL system under test, e.g. coverage reports do not provide enough information to understand what input features might have caused misbehaviours. Consequently, the usefulness of these approaches for the developers is strongly limited in practice.

Few approaches [2, 26] use behavioural properties during test generation, but none of them considers the combination of interpretable features of the DL system under test as the target of test

generation. This hinders them from exploring the feature space at large and providing a detailed explanation on how the system behaves for qualitatively different inputs.

In this paper, we introduce a novel way to assess the quality of DL systems by automatically generating a large, diverse set of high-performing (i.e., misbehaving or near-misbehaving), but qualitatively different test inputs that provide developers with a human-interpretable picture of the system's quality. With our approach, developers can understand how different structural and behavioural features of the inputs combine to affect the system's performance. To this aim we developed DeepHyperion, an open source automated test input generator for DL systems that leverages the key advantages of Illumination Search, i.e. a family of search algorithms that "illuminate" the input space by returning the highest-performing solution at each point of the search space defined by features of interest to the user [23].

DeepHyperion is the first approach to apply Illumination Search to DL system testing and to provide developers with a feature map, where the automatically generated inputs are positioned based on their characteristics and where the misbehaviours they expose can be interpreted (see Figure 1 for an example).

A crucial element of our approach is the choice of the dimensions that define the feature space of interest. In particular, the features should represent meaningful properties of the test scenarios, i.e. discriminative and interpretable properties of the inputs, or behavioural properties manifested by the DL system when exercised by the test inputs. To this aim, we propose a novel systematic methodology that can be used in conjunction with DeepHyperion to define the feature dimensions in a domain of interest, making it possible to generate test cases that illuminate the associated map in such domain. This methodology supports the identification of the features that better characterise the generated inputs and the definition of metrics that quantify the selected features.

We evaluated the proposed technique on both a classification problem (handwritten digit recognition) and a regression problem (steering angle prediction in a self-driving car). Results show that, for both problems, DeepHyperion is effective in generating failure-inducing inputs that are structurally or behaviourally different among them, as they cover different regions of the feature space. We compared DeepHyperion with state-of-the-art test input generators. Our results show that DeepHyperion can explore the feature space at large, whereas existing tools ignore parts of the feature space and expose only misbehaviours that belong to a narrow region of such space.

To foster research and replication, we release the code implementing DeepHyperion, the dataset, and all the scripts to replicate the experimental evaluation [4].

2 THE DEEPHYPERION TECHNIQUE

DeepHyperion aims to explore extensively the feature space of a DL system to find the most misbehaving solutions (i.e., those that deviate the most from the expected behaviour) with diverse characteristics. DeepHyperion implements the Illumination Search algorithm proposed by J.B. Mouret and J. Clune, named Multi-dimensional Archive of Phenotypic Elites (MAP-Elites) [23]. Given N dimensions of variation of interest, which define the feature space, DeepHyperion looks for the most misbehaving solution at each point in the space defined by those dimensions, trying to fill the entire feature map. The degree of misbehaviour exhibited by a solution is measured by a properly defined *fitness function*. For example, in a grey-scale digit classification problem, two dimensions of interest could be the boldness and discontinuity of the image, whereas the fitness function could be the misclassification probability computed from the softmax layer output of the Deep Neural Network (DNN) [12]. In such a case, the output of DeepHyperion would be a map where each cell is associated with a specific level of boldness and discontinuity, while the entry of each cell would provide an input image with such boldness and discontinuity, which is either misclassified or close to being misclassified – i.e., corner case inputs with the given features (see Figure 1).

Algorithm 1: DeepHyperion's Illumination Search

```
Input  : B: execution budget
         seedsize: seed pool size
         popsize: population size
Output : M: feature map
1  /* Initialise the map of elites */
2  map M ← ∅;
3  /* Generate initial population */
4  seeds S ← GenerateSeeds(seedsize);
5  foreach s ∈ S do
6  |  Evaluate(s);
7  end
8  population P ← InitialisePopulation(S, popsize);
9  /* Populate map with initial population */
10 foreach ind ∈ P do
11 |  M ← UpdateMap(ind) ;
12 end
13 /* Iteratively update the map */
14 while elapsedBudget < B do
15 |  ind ← RandomSelection(M);
16 |  ind_μ ← Mutate(ind);
17 |  Evaluate(ind_μ);
18 |  M ← UpdateMap(ind_μ);
19 end
20 return (M)
```

Algorithm 1 outlines the top level steps of the Illumination Search implemented in DeepHyperion. The map M to be generated is initially empty (line 2). Then a pool S of candidate valid inputs (seeds) is generated (line 4) and evaluated (lines 5-7), which means each seed is associated with its feature values and its fitness function value. The resulting pool of seeds is used to initialise the population P to be evolved by the algorithm (line 8). The position of each initial individual in the map is determined and the highest fitness individual is added to the map in the corresponding position (lines 10-12). Then, the main evolutionary loop is executed, with a termination condition determined by the execution budget (lines 14-19). At each loop iteration, an individual randomly selected from the current map is mutated and evaluated (lines 15-17) and 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). In the next sections, we describe the key design

choices behind DeepHyperion's algorithm and how we applied it to the chosen application domains.

2.1 Model-Based Input Representation

DeepHyperion belongs to the family of the model-based input generation techniques [30]. It does not directly modify raw input data (e.g., pixels) but it manipulates a *model* of the input that is later used to derive the actual raw input data. This enables DeepHyperion to generate more realistic inputs, belonging to the input validity domain [26]. This implies that DeepHyperion is applicable to a given domain if we have a generative model of the input data processed in such domain. The development of input models is standard practice in several domains, among which safety-critical ones such as automotive [18]. Generative input models are largely domain-specific. In the following, we present two examples of such models for the domains we considered in our experimental evaluation: handwritten digits, processed by a digit classifier, and driving scenarios for self-driving cars.

As regards the handwritten digits, the test inputs are images in the MNIST [19] format. MNIST is a database of 70 000 handwritten digits, originally encoded as 28 x 28 images with greyscale levels that range from 0 to 255. We model them as combinations of Bézier curves, adopting the Scalable Vector Graphics (SVG)[1] representation. The control parameters that determine the shape of the modelled digit are: the start point, the end point and the control points that define each Bézier segment. This representation ensures that the realism of handwritten shapes is preserved even after minor manipulation of the Bézier curve parameters [26]. We use the Potrace algorithm [28] to transform an MNIST input into its SVG model representation. To transform an SVG model back to a 28×28 grayscale image, we perform a rasterisation operation by means of the functionalities offered by two popular open source graphic libraries (i.e. LibRsvg[2] and Cairo[3]).

In the autonomous driving domain, the test input is the scenario in which the car drives. A simulated scenario can be modelled as the composition of the roads, the driving task (i.e., start point, end point and lane to keep), and the environment, which includes the weather and lightness conditions. We consider input scenarios similar to the those generated by the state-of-the-art testing tool DeepJanus [26]. These scenarios consist of plain asphalt roads surrounded by green grass on which the car has to drive keeping the right lane. The environment is set to a clear day without fog. The roads are composed of two lanes with fixed width in which there is a yellow center line plus two white lines that separate each lane from the non-drivable area. Our model of a road is a sequence of consecutive points in a bi-dimensional space. To produce a smooth and realistic shape for the road being modelled, we use Catmull-Rom cubic splines [8]. The control parameters that determine the shape of the splines are the coordinates of the control points of the center line spline. To transform the model into a road to be rendered in the simulator, we calculate the road points by means of the recursive algorithm for the evaluation of Catmull-Rom cubic splines proposed by Barry and Goldman [5] and the functionality offered by the Shapely library, for the manipulation and analysis of planar geometric objects[4]. We also enforce the following constraints to ensure that we generate a valid road: (1) the start point and the end point of the road should be different, (2) the road should fall within a square bounding box of fixed size, and (3) a road should not self-intersect.

2.2 Fitness Function

The Evaluate function (lines 6 and 17 in Algorithm 1) evaluates an individual *ind* by determining the values of its features $\{ind.f_1, ..., ind.f_n\}$ and of its fitness function *ind.fitness*, both of which are domain/problem specific.

For what concerns the definition of the relevant input features in a given domain, we propose a novel, systematic methodology, described in detail in Section 3. For what concerns the fitness function, the general idea is that it should quantify how close the DL system is to a misbehaviour. In the following, we illustrate the definition of a sensible fitness function for each of the two domains of handwritten digit recognition and autonomous driving.

For the *digit classification problem*, we exploit the activation levels available in the output softmax layer of the DNN that classifies the input image. In fact, the softmax output can be interpreted as a confidence level assigned to each of the possible classes [12], where the selected class for the given input is the one with highest confidence. More specifically, we calculate the difference between the confidence level associated of the expected class and the maximum confidence level associated to any other class. In this way, we get a fitness value that is close to zero when the correct class and the second highest class have similar activation levels, while we get a negative number when the input is misclassified. Hence, this fitness function is to be minimised.

For the *steering angle prediction problem*, we adopt a fitness function similar to that used by DeepJanus [26]. The behaviour of the self-driving system is characterised by the distance of the car from the center of the lane during the simulation. The fitness is calculated as $\min(w/2 - d)$, where w is the width of the lane and d the distance of the 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 center of the lane, whereas it returns a negative number when the car is out of bound. Hence, this fitness function is also to be minimised.

2.3 Feature Map

The feature map represents the feature space defined by N dimensions of variation that characterise the input or the behaviour of the DL system under test. Given the values of an individual's features, $ind.f_i, \forall i \in [1 : N]$, function UpdateMap (lines 11 and 18 in Algorithm 1) computes the individual's coordinates in the map M by applying the following mapping function:

$$x_i = \lfloor \alpha_i \cdot ind.f_i \rfloor \quad (1)$$

The value $ind.f_i$ is converted to an integer x_i after multiplying it by a constant α_i that ensures approximately the desired grid size, given the expected range of each feature $ind.f_i$. For instance, if the desired grid size is 100 and $ind.f_i$ is known to range between 0 and 1, a proper selection of α_i could be $\alpha_i = 100$. The resulting integer

[1] https://www.w3.org/Graphics/SVG/
[2] https://wiki.gnome.org/Projects/LibRsvg
[3] https://www.cairographics.org
[4] https://github.com/Toblerity/Shapely

x_i is used as an index in the map M to get the cell to be assigned to the individual *ind* (in a 2D map, $M[x_1, x_2]$).

During the search, the size of map M grows dynamically as higher/lower index values x_i are discovered. In fact, initially M has zero entries along all its dimensions. As soon as a new index x_i is discovered during the search process, M is extended to accommodate the newly discovered range of each dimension. For instance, if the first mapped individual has indexes $(x_1, x_2) = (2, 3)$, the initial map M will have size 1 in both directions and will contain just one cell, at position $(2, 3)$. If later another individual is mapped to $(x_1, x_2) = (5, 1)$, the map M will be extended to cover the integer range [2:5] along its first dimension and [1:3] along the second dimension. Hence, at this point of the search the map will cover the rectangle [2:5] × [1:3], which means it will contain 12 cells.

At the end of the search, the final map can be adjusted to allow the user to define a granularity different from the dynamically discovered one. This is particularly useful if users want to compare maps produced in different runs/configurations or by different algorithms. The final remapping is a linear rescaling function:

$$x'_i = \left\lfloor GS \frac{ind.f_i - min_i}{max_i - min_i} \right\rfloor \quad (2)$$

where GS is the desired grid size, while min_i, max_i are the minimum/maximum value of the i-th feature observed across all maps being rescaled to the new grid.

2.4 Initial Population Generation

The generation of the initial population consists of choosing an initial set of diverse individuals from the feature space, given a set of seeds of size *seedsize* and the population size *popsize*. Function GenerateSeeds (line 4 in Algorithm 1) generates seeds that are valid inputs for the system under test. More specifically, for digit classification, seeds are randomly chosen inputs from the MNIST database and converted to SVG. For steering angle prediction, we randomly generate valid roads.

DeepHyperion evaluates the fitness and the feature values of the generated seeds and then it finds the map cells to which they belong (lines 5-7 in Algorithm 1). Then, function InitialisePopulation (line 8 in Algorithm 1) selects as initial population the most diverse inputs among the available seeds, by computing the pairwise Manhattan distance [17] (sum of the absolute differences of the map coordinates) and greedily constructing the set of most diverse seeds, starting from a randomly selected first seed, up to the desired population size.

2.5 Selection and Mutation

Function RandomSelection (line 15 in Algorithm 1) randomly chooses an already filled cell in the map [23]. The individual that occupies the chosen cell is selected for the next genetic evolution. In this way, DeepHyperion is not biased towards the solution with highest fitness, like classic evolutionary algorithms, and it can explore the feature space at large, with the ultimate goal of "illuminating" it as completely as possible.

The selected individual is mutated by the Mutate operator (line 16 in Algorithm 1). This operator manipulates the input model's control parameters by applying a small perturbation to them. After applying the operator, DeepHyperion verifies that the mutant complies with the constraints of the input domain. Moreover, it also verifies that, once concretised into an actual input for the DL system, the mutant is different from its parent. If any of these checks fails, the operator is applied repeatedly, until a valid input is obtained.

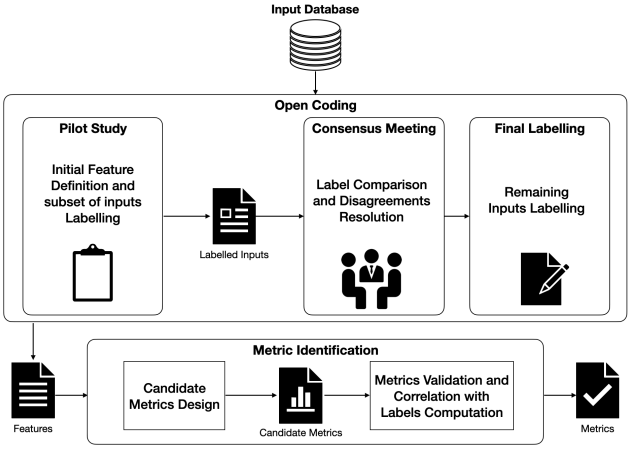

Figure 2: Feature Selection Methodology

3 DEFINITION OF THE MAP DIMENSIONS

A crucial element of our approach is the choice of the dimensions of variation of the automatically generated test cases. Such dimensions define both the feature space of interest to the user [23] and the search space of DeepHyperion. In the case of DL testing, they should represent meaningful properties of the test scenarios: either discriminative and interpretable *structural features* of the inputs, or *behavioural features* observed as the DL system processes the input and produces its output.

We propose an empirical methodology that can be used to define the feature dimensions in a new domain of interest. Our methodology consists of two macro-steps (see Figure 2): (1) *open coding*: select the features that better characterise the generated inputs, and (2) *metric identification*: quantify the selected features. The second step is needed to provide DeepHyperion with quantitative feature values to position the generated tests in the feature map.

3.1 Open Coding

The first step entails an *open coding procedure* [27] in which a set of existing inputs is manually analysed by human assessors to select the relevant features in a given domain. Since we are interested in both structural and behavioural features, the information provided to the human assessors is not restricted to the bare inputs (i.e. digit images and roads): it also includes the output of the DL system when processing the given existing inputs (e.g., the class predicted by an image classifier), as well as any relevant behavioural data (e.g., the trajectory of the car driving on the input road). The assessors independently tag the inputs assigned to them by either reusing an existing feature label or defining a new one. Each feature label is composed of a feature name, paired with the corresponding feature value, chosen from a rating scale, usually with five levels.

For instance, a hypothetical *speed of a self-driving car* label will have values that range between -2 and +2, where -2 means "very low", while +2 means "very high". 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, and supports conflict resolution when assessors evaluate the same input differently.

In our methodology, it is strongly advised to run a preliminary pilot study on a subset of inputs to gain confidence in the labelling procedure and, more importantly, agree on the meaning of the features and on the interpretation of the corresponding values. The pilot is concluded with a consensus meeting in which,the disagreements are solved either by consensus among the assignees or arbitration by the other assessors. In our experience, a disagreement is worth being discussed in the consensus meeting when the assigned values differ by more than 1 position in the rating scale (e.g., a disagreement between "very low" and "low" speed can be just ignored, while one between "low" and "high" is worth being discussed and solved). It might happen that the assessors realise through the discussion that some important features have been overlooked. Therefore, as part of the consensus meeting, assessors are allowed to agree upon additional features to be considered during the labelling procedure. Only when a common understanding of the features and of their possible values is reached, we suggest that it is possible to switch from the pilot study mode to the final study mode. In the final study, it is usually enough that each remaining unlabelled input is evaluated by a single assessor. In fact, while during the pilot study the number of inputs being labeled is kept small, in the final labelling phase we typically want to label as many inputs as possible.

3.2 Metric Identification

The second step of our methodology aims to define a set of candidate metrics that can accurately quantify the features that have been identified in the previous step. The candidate metrics are then applied to the inputs considered in the first step. To select the most accurate metrics for the given features, we compute the Pearson correlation coefficient [16] and the associated p-value, between the manually defined feature values, converted from the rating scale to a numeric scale (e.g., in the range [1:5]), and the values returned by the candidate metrics. The metrics with highest, statistically significant (p-value < 0.05) correlation are chosen to quantify the selected features. In the following, we provide the details about how this methodology was applied to each of our case studies, i.e. digit recognition and autonomous driving.

3.3 Dimensions for Digit Recognition

3.3.1 Open Coding. In this phase, three authors acted as assessors. In the pilot, we randomly selected 30 images from the MNIST database and each assessor was assigned 20 images, such that each input was evaluated by two assessors. The assessors identified the following features, to which they assigned values within a range from -2 to 2:

- **Boldness**, indicates how strong the stroke of the handwriting is. It ranges from very thin (−2) to very thick line (2).
- **Smoothness**, indicates the absence of sharp angles in the digit. It ranges from sharp angles (−2) to smooth angles (2).
- **Discontinuity**, indicates how continuous the stroke of the handwriting is. It ranges from continuous line (−2) to digits made of multiple disconnected segments (2).
- **Rotation** with respect to the vertical axis. It ranges from strongly tilted to the left (−2) to strongly tilted to the right (2).

Table 1: Correlation between features and metrics

Case Study	Feature	Metric	Agree	Correl	p-val
MNIST	Boldness	Lum	100%	0.67	<0.002
	Smoothness	AvgAng	66%	0.05	0.241
	Discontinuity	Mov	100%	0.90	<0.002
	Rotation	Or	-	0.43	<0.002
BeamNG	Smoothness	MinRad	95.8%	0.58	<0.002
	Complexity	TurCnt	87.5%	0.63	<0.002
	Orientation	DirCov	89.5%	0.66	<0.002

Examples of images of handwritten digits at various levels of Boldness and Discontinuity can be found in Figure 1. The inter-rater agreement during the pilot study, measured as the percentage of inputs that were assigned the same feature value or feature values with a difference of 1, is reported in Table 1 under *Agree*. We observed that assessors strongly agreed over Boldness and Discontinuity (i.e., no conflicts have been registered). Noticeably, Table 1 does not report any agreement value for Rotation, as the assessors introduced this feature during the consensus meeting, i.e., after the data collection for the pilot study ended. In the final phase, we randomly selected 600 images from MNIST and each of the three assessors labelled 200 images.

3.3.2 Metric Identification. To measure each feature resulting from the labelling procedure, we designed several candidate metrics and applied them to the 630 images labelled by the assessors. Table 1 (top) shows the metric with highest correlation for each MNIST feature, together with the corresponding correlation and p-value:

- **Luminosity** (Lum): number of light pixels of the image, i.e., pixels whose value is above 127.
- **Average Angle** (AvgAng) the average angle of the Bezier curves in the SVG representation of the digit.
- **Moves** (Mov): sum of the Euclidean distances between pairs of consecutive sections of the digit. To obtain the sections of a digit, we convert its bitmap to SVG.
- **Orientation** (Or): vertical orientation of the digit, obtained by computing the angular coefficient of the linear regression of the non-black pixels, i.e., pixels with value greater than 0.

As shown in Table 1, for Boldness, Discontinuity and Rotation we were able to define metrics that significantly correlate with the human assessment, whereas this was not possible for Smoothness, which turned out to be both difficult to evaluate for humans (see low inter-rater agreement) and difficult to quantify precisely. Hence, this feature was not included among those used by DeepHyperion for input generation.

3.4 Dimensions for Autonomous Driving

3.4.1 Open Coding. In this phase, all the authors acted as assessors. In the pilot, we randomly generated 40 virtual roads according to our model representation. Each assessor was assigned 20 images representing roads, so that each road was evaluated by two assessors. To simplify the job of the assessors, the web application supporting the labelling procedure provides some interaction facilities for the inspection of the road, such as: (1) zoom in/out; (2) selection of specific road segments; (3) navigation along the road; (4) toggling the visualisation of the car. The images abstract the roads over a two-dimensional plane but retain their geometrical properties and the proportions to the vehicle. In the images we draw boxes that represent the vehicle and cones that represent its field of view, to give assessors more context.

The assessors identified the following features, to which they assigned values within a range from 0 to 5:

- **Smoothness**, indicates how smooth the turns of the road are. It ranges from sharp turns (0) to gentle turns (5).
- **Complexity**, indicates how complex the road's shape is. It ranges from almost straight roads (0) to roads with many turns (5).
- **Orientation**, indicates how many directions (i.e., N, NE, E, SE, S, SW, W, NW) the road covers. It ranges from straight road which is oriented to one direction only (0) to road that covers the whole spectrum of directions (5).

As reported at the bottom of Table 1, during the pilot, we observed that assessors generally agreed upon all the features. In the final phase, we randomly generated 400 roads and each assessor tagged 100 of them.

3.4.2 Metric Identification. We designed a set of candidate metrics and applied them to the 440 images labelled by the assessors. We eventually selected the following 3 metrics that best correlate with the corresponding features, as reported in Table 1 (bottom):

- **Minimum radius of curvature** (MinRad): minimum value of the radius for the circles passing through triplets of consecutive road waypoints.
- **Turn Count** (TurCnt): number of turns in the road, where a turn is a change of direction between consecutive road segments by more than 5°.
- **Direction Coverage** (DirCov): number of different angular sectors covered by the directions of the road segment. In particular, we consider 36 sectors, each spanning 10°.

In addition to the features that characterise the structure of the test input, we considered further features to capture the behaviour of the car during the simulation. In particular, we used the following metrics that have been proposed as quality metrics for self-driving cars [14] to measure the *quality of driving*:

- **Standard deviation of the steering angle** (StdSA): standard deviation of the sequence of steering angles collected along the road during self-driving.
- **Mean lateral position of the car** (MLP): mean distance between the center of the car and the center of the driving lane, where the mean is computed across all car positions observed along the road.

4 EXPERIMENTAL EVALUATION

4.1 Subject Systems

We evaluate DeepHyperion on two DL systems which address different tasks and belong to different domains. Moreover, they have been widely used in the literature to evaluate testing techniques for DL systems [25, 32]. Hereafter, we refer to these systems as MNIST and BeamNG, respectively.

The MNIST system performs a classification task, which consists of recognising handwritten digits from the MNIST dataset [19]. It is a DNN that predicts which digit is represented in a greyscale image. We considered the DNN instance provided by Keras, [9] because of its popularity. It has 99.8% test accuracy, obtained after training it ourselves on the MNIST training set with its default configuration, i.e. 12 epochs, batches of size 128, and a learning rate equal to 1.

The BeamNG system is a self-driving car equipped with a Lane Keeping Assist System (LKAS). The DL component solves a regression problem, i.e., it predicts the steering angle of the car given the image of its onboard cameras. We tested the whole DL system which includes the LKAS by using the BeamNG.research driving simulator [6], a freely available research-oriented version of the commercial game BeamNG.drive. The DL component driving the car utilises the DAVE-2 architecture designed by Bojarski et al. at NVIDIA [7], consisting of three convolutional layers, followed by five fully-connected layers. The DNN was trained with images captured by the camera sensors of the car, paired with the steering angles provided by the simulator's autopilot, which takes advantage of global knowledge and computes the optimal steering angle geometrically. We trained the model for 4, 600 epochs, with batches of size 128 and a learning rate equal to 0.001. We used a training dataset obtained by letting the autopilot drive up to 15 mph on 30 randomly generated roads.

4.2 Research Questions

The goal of our evaluation is to understand whether coupling feature maps and automated test generation is an effective technique for DL testing, which (1) can thoroughly stress the DL system under different conditions, and (2) can provide information useful to characterise problems in DL systems. Therefore, we seek to answer the following research questions:

RQ1 (Failure Diversity): *How effective is DeepHyperion in generating test inputs that expose diverse failures?*

Generating tests that trigger failures is more useful when these failures are diverse. Whereas, a test generator that repeatedly exposes the same problem is not desirable, as it wastes computational resources.

Metrics: To assess how many different failures are triggered during a run, we measure the number of *Mapped Misbehaviours (MM)*, i.e. how many cells of the feature map M contain at least one failure-inducing input.

To measure how the mapped misbehaviours of M are diverse, we compute the *Misbehaviour Sparseness (MS)*, defined as the average maximum Manhattan distance between cells containing misbehaviours ($MM \subseteq M$):

$$\mathrm{MS}(M) = \frac{\sum_{i \in MM} \max_{j \in MM} \mathrm{dist}(i, j)}{|MM|} \quad (3)$$

RQ2 (Search Exploration): *How extensively does DeepHyperion explore the feature space?*

Effective test generation should exercise different behaviours of the systems under test. This can be achieved by exploring the feature space extensively, at large.

Metrics: We measure the map coverage as the number of *Filled Cells* in the map ($FC \subseteq M$). Moreover, to measure how broadly our tool explores the feature space, generating inputs in diverse cells, we use the *Coverage Sparseness (CS)*, defined as the average maximum Manhattan distance between filled cells:

$$\mathrm{CS}(M) = \frac{\sum_{i \in FC} \max_{j \in FC} \mathrm{dist}(i, j)}{|FC|} \quad (4)$$

RQ3 (Feature Discrimination): *How strongly do different combinations of features characterise the failure-inducing inputs?*

The existence of regions of the map where the probability of misbehaviours is very high indicates that the corresponding feature value combinations are very likely to induce a failure, since most of the times when the combination was generated by DeepHyperion, a misbehaviour was observed. This could provide developers with a powerful tool to understand the conditions responsible for misbehaviours (a form of root-cause analysis).

Metrics: To answer this research question, we compute the *Misbehaviour Probability (MP)* for each cell of a map as the ratio between the number of failure-inducing inputs and the total number of inputs generated by DeepHyperion during the search process for that cell. Since occasionally only a small number of inputs may be generated by DeepHyperion for a given cell during the search, our estimate of MP might be affected by a large error. Hence, we also compute the confidence interval of MP. In particular, we use Wilson's confidence interval estimator for binomial random variables: in our case, such binomial variable indicates whether a misbehaviour is induced or not. We consider a combination of feature values with a high probability of failure if its MP value is greater than 0.8 and the lower bound of its confidence interval is above 0.65.

4.3 Experimental Procedure

To answer our research questions, we ran DeepHyperion with different combinations of the features we identified following our methodology (see Section 3). We limited DeepHyperion to use only pairwise combinations of features to ease the visualisation of the maps and the discussion of the results; however, the algorithm is general and works also with maps that have more than two dimensions. For MNIST, we report the results achieved by considering all the three features that significantly correlate with the corresponding metrics (see Table 1). As regards BeamNG, we conducted our experiments on five feature combinations that cover the three combinations types: two structural features, two behavioural features, and a combination of a structural and a behavioural feature.

We compared DeepHyperion with state-of-the-art testing approaches for DL systems, i.e. DeepJanus [26] and DLFuzz [13].

DLFuzz is representative of approaches that generate test inputs for image classifiers by applying perturbations to the raw input (i.e., pixels), often used to generate adversarial examples and test the robustness of DL components. It has been applied to the MNIST system. However, it cannot be applied to BeamNG since it could only manipulate individual camera inputs, without affecting the road shape.

Table 2: DeepHyperion Configurations

Parameter	MNIST	BeamNG
seed pool size	900	40
population size	800	24
time budget (s)	3600	36000
mutation lower bound	0.01	1
mutation upper bound	0.6	6

DeepJanus is a search-based tool that generates inputs at the frontier of behaviours of DL systems, i.e. similar inputs that trigger different system behaviours. It is a model-based approach that can be applied to both BeamNG and MNIST systems. Moreover, it shares with DeepHyperion the same input representation which guarantees a consistent measurement of the features and, thus, a fair comparison of the approaches.

We ran each tool the same number of times on each subject system, i.e. 30 times on MNIST and 10 times on BeamNG, respectively. To ensure a fair comparison, we ran them on the same computing nodes and used the same time budget for each tool, i.e. 3600 seconds for MNIST and 36000 seconds for BeamNG, respectively. The reason for the different time budgets is that testing MNIST requires only to feed it an image and get the corresponding prediction, which usually is a matter of milliseconds, while testing BeamNG requires to execute driving simulations that take several minutes to complete.

The configurations of DeepHyperion were obtained in a few preliminary runs and are reported in Table 2. With the other tools, we either used the configuration reported as the one achieving the best performance or directly contacted the tools' corresponding authors when some details about the configuration were missing.

The initial seeds for MNIST were obtained by randomly selecting 900 inputs from the MNIST test set, all belonging to the class "5". We obtained similar results for other digit classes, but we do not report them for space reasons. For BeamNG, the seeds were defined by 10 control points in which the initial point was always at a fixed position, whereas the others were placed at a random position 25 meters away from the previous one.

At the end of the runs, we used the inputs generated by each tool and the outputs generated by the subjects to compute the feature map of each run. All the maps were generated with the same number of cells for each feature, i.e. up to 25 cells, by using the rescaling function described in Equation 2. The min/max values defining the range for each feature were the ones observed across the runs of all the tools. We used these feature maps to compute the metrics associated with each research question. To assess the statistical significance of the comparisons between DeepHyperion and the considered state-of-the-art tools, we performed the Mann-Whitney U-test and measured the effect size by means of the Vargha-Delaney's $\hat{A}_{12}$ statistic [3].

5 RESULTS

5.1 RQ1: Failure Diversity

Figure 3 shows the number of diverse misbehaviours found by each tool in the MNIST system and their sparseness on the map.

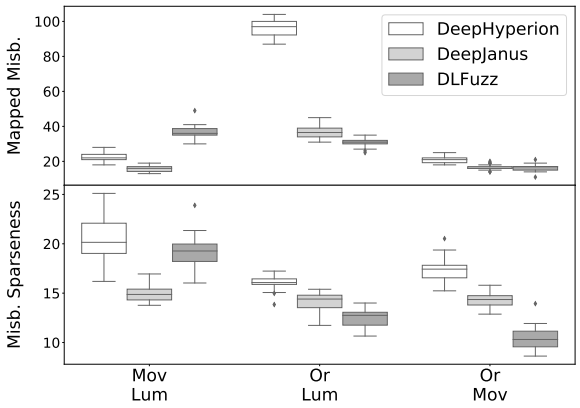

Figure 3: RQ1: Misbehaviours found by DeepHyperion, DeepJanus and DLFuzz on MNIST

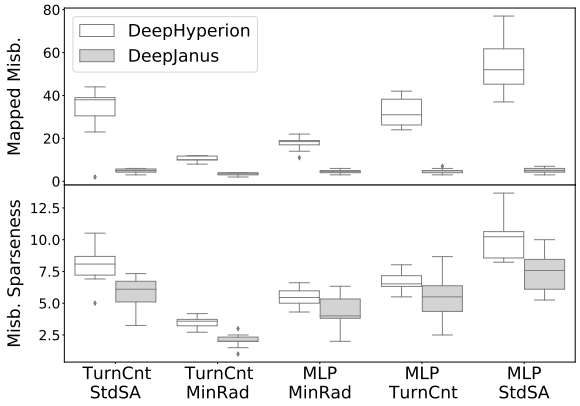

Figure 4: RQ1: Misbehaviours found by DeepHyperion and DeepJanus on BeamNG

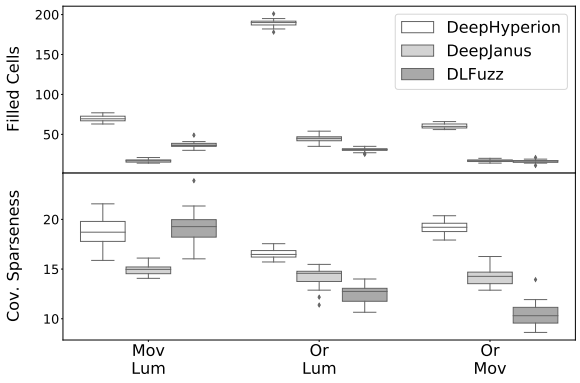

Figure 5: RQ2: Map cells filled by DeepHyperion, DeepJanus and DLFuzz on MNIST

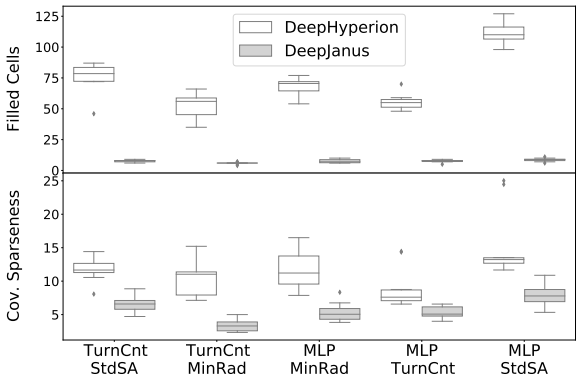

Figure 6: RQ2: Map cells filled by DeepHyperion and DeepJanus on BeamNG

As shown in Figure 3 (top), DeepHyperion found more than 20 diverse misbehaviours for each feature combination. DeepHyperion outperformed the other tools by generating a higher number of diverse failures for two out of the three feature combinations (p-values < 0.05; large effect size). In particular, for the Or-Lum feature combination DeepHyperion produced a number of mapped misbehaviours that is remarkably above all the others (exceeding the second best by > 50). As regards the Mov-Lum combination, DLFuzz produced maps containing a significantly higher number of cells with at least one misbehaviour. It should be noted that DLFuzz operates directly on the image pixels. Therefore, this competitor is able to produce less continuous images with more light pixels. However, these inputs are supposedly less realistic and less representative than the DeepHyperion's ones [26]. Moreover, although more numerous (for this feature combination), the misbehaviours exposed by DLFuzz are less sparse than DeepHyperion's (see bottom of Figure 3), which indicates that DLFuzz tends to repeatedly expose similar misbehaviours.

Figure 3 (bottom) shows that DeepHyperion produced failure-inducing inputs that are more sparse on the feature map for all feature combinations, as its Misbehaviour Sparseness (MS) metric is always significantly higher than the compared tools with p-values < 0.05 (effect size is always large with the exception of Mov-Lum vs DLFuzz, for which the effect size is medium). This result is achieved despite DeepJanus explicitly rewards diversity, having a fitness function that promotes the euclidean distance among solutions. DeepHyperion can expose a large number of misbehaviours Figure 3 (top), and, more importantly, it can reveal highly *diverse misbehaviours*, associated with very distant feature combinations (Figure 3, bottom).

Figure 4 shows Mapped Misbehaviours and Misbehaviour Sparseness of the tools that have been applied to the BeamNG system. Figure 4 (top) shows that DeepHyperion was always able to expose several diverse failures of the BeamNG system (at least 10.5 on average). In comparison with DeepJanus, it produced significantly more misbehaviours for all the feature combinations (p-values < 0.05, large effect size). In particular, MLP-StdSA produced an impressive number of mapped misbehaviours (53.4 on average) which is remarkably above the competitor (exceeding it by 48.3). The goodness of the combination of behavioural features is confirmed also on the sparseness of the misbehaviours, as shown in Figure 4 (bottom),

since it performed better than the other combinations, i.e. 10.14 on average. With respect to the competitor, DeepHyperion generated significantly sparser inputs for four out of five combinations. Only for MLP-TurnCnt, the sparseness values of the inputs generated by the two tools do not show any statistically significant difference (p-value = 0.06, slightly above the conventional threshold of 0.05; medium effect size in favour of DeepHyperion).

> **Summary**: *DeepHyperion can find diverse failure-inducing inputs for all feature combinations. It can detect up to 10X more than the competitor in BeamNG.*

5.2 RQ2: Search Exploration

Figure 5 shows the number and the sparseness of the filled cells in the maps produced by each tool for the MNIST system. Figure 5 (top) shows that DeepHyperion covered all feature maps more extensively than the other tools (with large effect size and p-value always < 0.05). Similarly to RQ1, the Or-Lum combination shows dramatically better results of DeepHyperion in comparison to the other tools, i.e. 130 additional cells filled by DeepHyperion.

Figure 5 (bottom) shows that DeepHyperion produced more sparse inputs for two out of three feature combinations, i.e. Or-Lum and Or-Mov. As regards the Mov-Lum combination, DeepHyperion has significantly better Coverage Sparseness than DeepJanus (p-value < 0.05, large effect size), whereas it does not show any statistically significant difference with DLFuzz (p-value > 0.05).

For the BeamNG system, Figure 6 reports Filled Cells and Coverage Sparseness achieved by the tools applied to the BeamNG system. Figure 6 (top) confirms that DeepHyperion is particularly good in covering the map corresponding to the combination of behavioural features MLP-StdSA (111.8 filled cells on average). In comparison with DeepJanus, it always filled significantly more cells (at least 47.7 cells more). Figure 6 (bottom) shows that DeepHyperion produced inputs that are always significantly sparser than DeepJanus (p-values < 0.05, large effect size).

> **Summary**: *DeepHyperion can always explore the feature space more extensively than the other tools (up to 4X for MNIST).*

5.3 RQ3: Feature Discrimination

In Figures 7 and 8, we report feature maps where the cell colour indicates the average Misbehaviour Probability (MP) across the runs for the corresponding feature combination (darker colour indicates higher probability). Blank cells are combinations that have never been explored by DeepHyperion. We highlight with a dark border the cells for which there is a MP > 0.8 and the lower bound of its confidence interval is > 0.65, which means that whenever an input is placed in these cells it is very likely to trigger a misbehaviour.

As regards MNIST, Figure 7 shows that each feature map produced by DeepHyperion has multiple regions where the probability of failure is high. For instance, Figure 7 (center) suggests that left-oriented and thin digits are very likely to cause a classification failure. DeepHyperion can further help the user to interpret its results by showing the most representative inputs for each cell. As an example, in Figure 1 we show the actual inputs generated by DeepHyperion representing the map in Figure 7 (left). We can see that bold are hard to recognise as "5" when part of the figure forms a circle, as they can be considered as "6" or "9". The bottom of the map shows that thin and discontinuous figures are hard to classify.

As regards BeamNG, Figure 8 shows that each of the maps corresponding to the combinations MLP-StdSA and MLP-MinRad has a clear region where the probability of failure is high. Figure 8 (left) suggests that the car is likely to go astray in roads that cause it to drive closer to the lane margins (lower MLP) and change often the steering angle direction (higher StdSA). Figure 8 (right) suggests that roads with at least a very sharp turn that cause the car to drive close to the lane margins are likely to cause a failure. The absence of high failure probability regions for the combination of the *structural* features TurnCnt-MinRad (see Figure 8 (center)) may indicate that *behavioural* features are more useful for characterising the conditions that trigger a misbehaviour. Developers can use these maps to improve the LKAS component of a self-driving car by augmenting the training set with driving scenarios that exhibit the troublesome feature combinations.

> **Summary**: *For both subjects, DeepHyperion can detect well-characterised regions of the feature space that are likely to expose failures.*

5.4 Threats to Validity

Construct Validity: DeepHyperion highly depends on map dimensions corresponding to measurable features. There is the risk that the selected features are not accurately quantified by the adopted metrics. To mitigate this threat, we (1) developed an empirical methodology to define the features of interest and the associated metrics, and (2) used metrics widely adopted in the literature.
External Validity: The choice of subject DL systems is a possible threat to the *external validity*. To mitigate this threat, we chose two diverse DL systems. One solves a classification problem, while the other is a self-driving car software that solves a regression problem. However, further studies with a wider set of DL systems should be carried out to fully assess the generalisability of our findings.

6 RELATED WORK

DL systems' quality has been mainly assessed by generating new inputs that expose misbehaviours [11, 13, 21, 24, 29] and by proposing novel adequacy criteria that guide the input generation process [15, 24, 31]. To the best of our knowledge, no technique aims at covering the feature space of DL systems and few works [2, 26] make use of interpretable properties for test generation.

6.1 Input Generation and Adequacy

DeepXplore [24] is a testing technique to detect behaviour inconsistencies among different DNNs. It is guided by neuron coverage, i.e., the percentage of neurons whose activation level is above a certain threshold. DLFuzz [13] is also a test input generator guided by neuron coverage. DeepTest [29] maximises the neuron coverage of a DNN-based steering angle predictor by applying different image

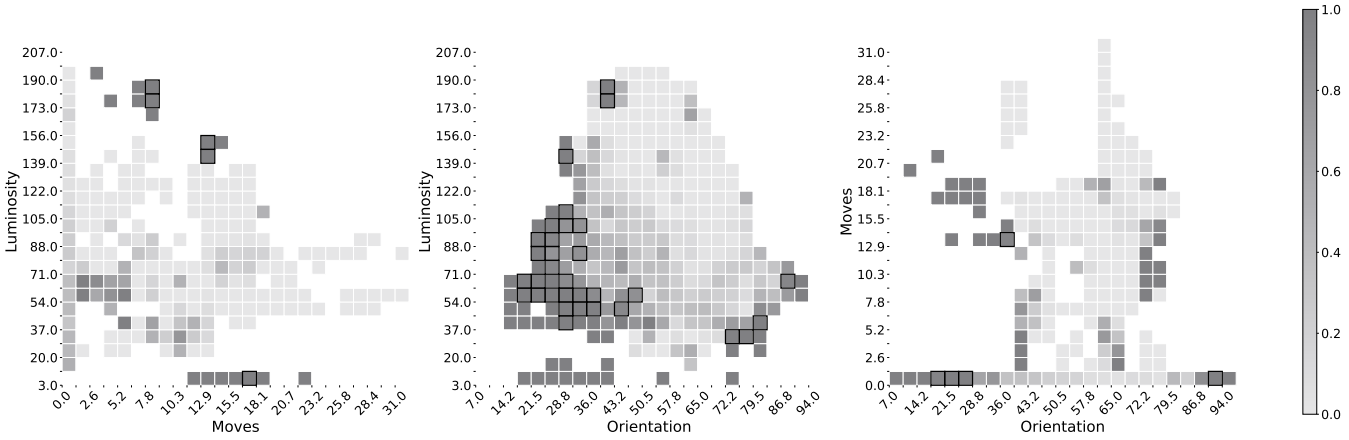

Figure 7: RQ3: Probability maps and feature discrimination for MNIST

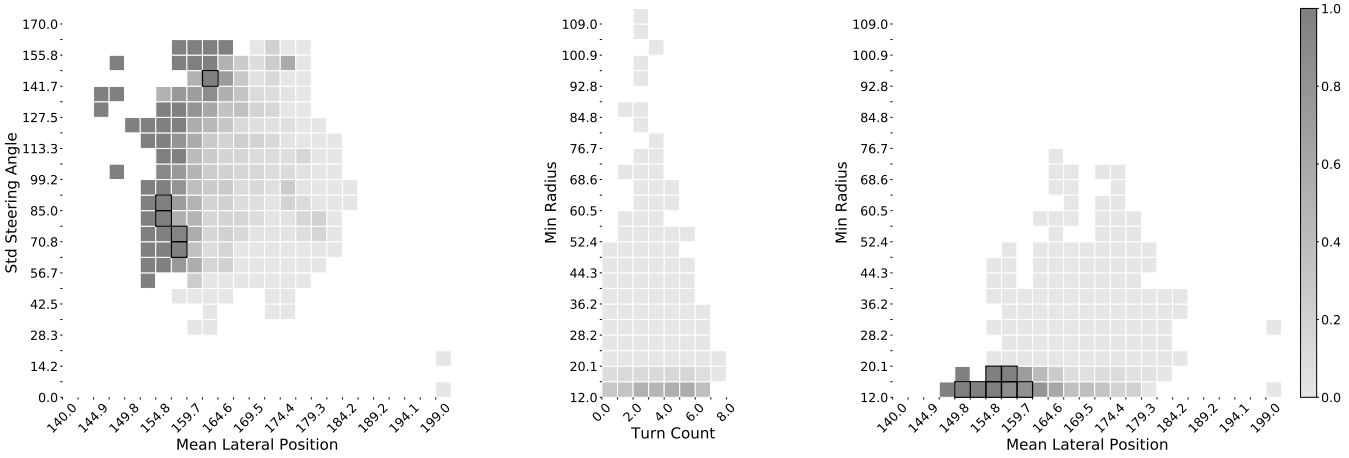

Figure 8: RQ3: Probability maps and feature discrimination for BeamNG

transformations to images captured by the on-board camera of an autonomous car. DeepGauge [21] uses a set of coverage criteria that extend neuron coverage by taking the distribution of training data into consideration. DeepCT [20] uses a set of combinatorial testing criteria for DNNs based on the interactions between neurons. DeepHunter [31] leverages multiple coverage criteria originally proposed by Ma et al. [21] as feedback to guide test generation. DeepSmartFuzzer [10] uses Monte Carlo Tree Search (MCTS) to exploit the coverage increase patterns. The degree of "surprise" of an input was measured by means of the two metrics proposed by Kim et al. [15], associated with a *surprise adequacy* coverage criterion.

The above test input generation techniques focus on generating adversarial inputs by adding perturbations to the original inputs. Other input generators [2, 11, 26] are instead based on the manipulation of a model of the inputs, which ensure more control on the validity and realism of the generated inputs, going beyond adversarial attacks that expose security vulnerabilities. For instance, DeepJanus [26] manipulates the way-points that define the shape of a road within a self-driving car simulator. AsFault [11] is a model-based approach that applies a search-based algorithm to test the lane-keeping system of self-driving cars. DeepHyperion belongs to this category of test generators.

All test generators mentioned above aim at maximising some internal adequacy metric, such as neuron or surprise 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. DeepHyperion is the first tool to provide developers with a map of such features, where the automatically generated inputs, as well as the exposed misbehaviours, are positioned and can be interpreted. Hence, existing test generators might completely ignore parts of a feature map or might expose only misbehaviours that belong to a narrow map region.

6.2 Behavioural Properties

NSGAII-DT [2] is a model-based approach for testing 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 multidimensional 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. While decision trees provide interpretable information to developers as DeepHyperion does with its feature maps, the variables that appear in decision nodes are limited to the control parameters of the input model, which might not be fully representative of all relevant behavioural features of the system under test. Moreover, decision trees are used to focus the search on critical scenarios (collisions or near-collision at high speed with pedestrians), so as to increase the search efficiency, while DeepHyperion aims 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.

DeepJanus [26] characterises the quality of a DL system as its frontier of behaviours, i.e., pairs of similar inputs that trigger different (expected vs failing) behaviours of the system. The output of DeepJanus provides users with a a set of system's frontier inputs, but it does not explicitly characterise them based on structural or behavioural features. Instead, DeepHyperion's maps allow developers to interpret the inputs that trigger a misbehaviour in terms of their feature values.

7 CONCLUSIONS AND FUTURE WORK

DeepHyperion provides a unique characterisation of a DL system's quality through an interpretable map which represents the highest-performing (i.e., misbehaving or closest to misbehaving) inputs in the space of the relevant, domain-specific features.

Our empirical study shows that DeepHyperion is more effective than state-of-the-art DL testing tools in generating failure-inducing inputs associated with highly diverse features. In the reverse direction, we showed that DeepHyperion is useful to detect the feature combinations that are most likely to induce a system misbehaviour. In our future work, we plan to generalise our results to a wider sample of DL systems, including industrial ones.

ACKNOWLEDGMENTS

This work was partially supported by the H2020 project PRECRIME, funded under the ERC Advanced Grant 2017 Program (ERC Grant Agreement n. 787703). The driving simulator has been 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*, David Lo, Sven Apel, and Sarfraz Khurshid (Eds.). 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, New York, NY, USA, 1016–1026. https://doi.org/10.1145/3180155.3180160

[3] 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

[4] Anonymous Author(s). 2021. DeepHyperion:Replication Package. https://github.com/anonuser-1234/anon_ISSTA_2021.

[5] Phillip J. Barry and Ronald N. Goldman. 1988. A Recursive Evaluation Algorithm for a Class of Catmull-Rom Splines. *SIGGRAPH Comput. Graph.* 22, 4 (June 1988), 199–204. https://doi.org/10.1145/378456.378511

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

[7] 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

[8] 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

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

[10] 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 co-located with the 29th International Joint Conference on Artificial Intelligence and the 17th Pacific Rim International Conference on Artificial Intelligence (IJCAI-PRICAI 2020), Yokohama, Japan, January, 2021 (CEUR Workshop Proceedings, Vol. 2640)*, Huáscar Espinoza, John McDermid, Xiaowei Huang, Mauricio Castillo-Effen, Xin Cynthia Chen, José Hernández-Orallo, Seán Ó hÉigeartaigh, and Richard Mallah (Eds.). CEUR-WS.org, 134–140. http://ceur-ws.org/Vol-2640/paper_19.pdf

[11] 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*, Dongmei Zhang and Anders Møller (Eds.). ACM, 318–328. https://doi.org/10.1145/3293882.3330566

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

[13] 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*, Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu (Eds.). ACM, 739–743. https://doi.org/10.1145/3236024.3264835

[14] Gunel Jahangirova, Andrea Stocco, and Paolo Tonella. 2021 - to appear. Quality Metrics and Oracles for Autonomous Vehicles Testing. In *Proceedings of 14th IEEE International Conference on Software Testing, Verification and Validation (ICST '21)*. IEEE, 1–12.

[15] 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*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 1039–1049. https://doi.org/10.1109/ICSE.2019.00108

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

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

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

[19] 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.

[20] 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*, Xinyu Wang, David Lo, and Emad Shihab (Eds.). IEEE, 614–618. https://doi.org/10.1109/SANER.2019.8668044

[21] 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, New York, NY, USA, 120–131. https://doi.org/10.1145/3238147.3238202

[22] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. 2008. *Introduction to Information Retrieval.* Cambridge University Press, New York, NY, USA.

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

[24] 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

[25] 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

[26] 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

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

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

[29] 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, New York, NY, USA, 303–314. https://doi.org/10.1145/3180155.3180220

[30] 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.

[31] 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, New York, NY, USA, 146–157. https://doi.org/10.1145/3293882.3330579

[32] 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

[33] 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*, Marianne Huchard, Christian Kästner, and Gordon Fraser (Eds.). ACM, 132–142. https://doi.org/10.1145/3238147.3238187