DeepMetis: Augmenting a Deep Learning Test Set to Increase its Mutation Score

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

Nargiz Humbatova
Università della Svizzera Italiana
Lugano, Switzerland
nargiz.humbatova@usi.ch

Gunel Jahangirova
Università della Svizzera Italiana
Lugano, Switzerland
gunel.jahangirova@usi.ch

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

ABSTRACT

Deep Learning (DL) components are routinely integrated into software systems that need to perform complex tasks such as image or natural language processing. The adequacy of the test data used to test such systems can be assessed by their ability to expose artificially injected faults (mutations) that simulate real DL faults.

In this paper, we describe an approach to automatically generate new test inputs that can be used to augment the existing test set so that its capability to detect DL mutations increases. Our tool DeepMetis implements a search based input generation strategy. To account for the non-determinism of the training and the mutation processes, our fitness function involves multiple instances of the DL model under test. Experimental results show that DeepMetis is effective at augmenting the given test set, increasing its capability to detect mutants by 63% on average. A leave-one-out experiment shows that the augmented test set is capable to expose unseen mutants, which simulate the occurrence of yet undetected faults.

1 INTRODUCTION

Deep Learning (DL) based software is widespread and has been successfully applied to complex tasks such as image processing and speech recognition. Systems including DL components are employed also in safety and business critical domains, e.g. autonomous driving and financial trading. DL systems possess the human-like ability to learn how to perform a task from experience, i.e., the inputs seen during training [28], but such ability comes with the possibility to make errors when presented with new inputs. Therefore, it is crucial for DL software developers and manufacturers to assess to what extent these systems can be trusted in response to real-world inputs, as they could face scenarios that might be not sufficiently represented in the data from which they have learned.

Traditional test adequacy criteria, like code coverage, fail to determine whether DL systems are adequately exercised by a test set, since most of the DL systems' behaviour depends on their training data, not the code. Recent research defined ad-hoc white-box adequacy metrics, based on DL software's internal architecture, e.g., neuron [12, 33, 40, 45] or surprise coverage [20]. A limitation of these approaches is that their output cannot be directly associated to a root cause of a DL system's failure, i.e., a DL fault [13].

On the other hand, mutation testing approaches evaluate a test set against faults that are artificially injected into the system under test. So, the inability of a test set to expose injected faults (*kill mutants* in the mutation testing jargon) can be interpreted as its inability to properly exercise the mutated code [19]. The tool *DeepCrime* generates mutants of DL systems by injecting artificial faults that resemble those described in the taxonomy of real DL faults by Humbatova et al. [15]. In this way, it addresses the challenge of simulating real-world DL faults [48]. Hence, a DL test set that cannot kill a mutant generated by *DeepCrime* is also unlikely to expose any real fault similar to the one injected by *DeepCrime*, in case such a fault affected the DL system under test. In such a situation, the test set should be augmented with additional tests that target the undetected fault.

In this paper, we introduce a novel and automated way to augment existing test sets with inputs that kill mutants generated by *DeepCrime*. Our goal is to increase the mutation score of a test set by generating new inputs that kill the mutants not killed by the original test set. To this aim, we propose DeepMetis, a search-based test generator for DL systems that uses mutation adequacy as guidance. Intuitively, a mutant is killed if the correct behaviour is observed for a DL model under test, while a misbehaviour is observed on its mutated version. However, mutation testing approaches should take into account the stochastic nature of DL (in particular, of its training process) and of mutation generation (some DL mutations are non-deterministic) to properly measure the test set's ability to discriminate the original system from the artificially generated faulty versions [35]. In fact, observing a drop in accuracy between the original and the mutated model is not enough to conclude that the mutant is killed, since such a drop might be due to random fluctuations of accuracy associated with the non-determinism of the training and the mutation process. The mutation killing criterion proposed by Jahangirova and Tonella [18] addresses this DL-specific challenge by evaluating a test set on multiple re-trained instances of the same model and applying statistical tests. DeepMetis adopts the same non-deterministic view on DL systems and correspondingly its generation process is guided by multiple instances of the model being mutated.

Recently, DL-specific mutation operators have been used for different tasks, such as program repair [39], adversarial inputs detection [43], generation of adversarial code snippets [34], and calculation of optimal oracles for autonomous vehicles [11], but no approach leveraged them to generate new inputs which augment an inadequate test set.

We evaluated DeepMetis on both a classification problem and a regression problem, using mutation operators provided by *DeepCrime*. Results show that DeepMetis is effective at generating inputs that improve a test set in terms of its mutation killing ability. We also conducted a leave-one-out experiment to simulate a practical usage scenario where an undetected fault affecting the DL system is unknown. In this experiment setting, one mutant produced by *DeepCrime* is taken apart, while test augmentation is performed by DeepMetis based only on the remaining mutants. In this way, the left out mutants simulate a yet unknown fault. Results show that left out mutants can be killed by the augmented test set on average 82% of the times.

2 BACKGROUND

2.1 Mutation Testing of DL Systems

Mutation testing is a technique that injects artificial faults into a system under test guided by the assumption that the ability to expose such artificial faults translates into the ability to expose also real faults. In traditional software systems, the main decision logic of a program is implemented in its source code and synthetic faults are introduced by applying small syntactic changes to the source code. In contrast, the behaviour of a DL system is determined not only by the source code but also by its training data, the structure of its neural networks or the tuning of various hyperparameters. As syntactic code changes are not sufficient to achieve realistic fault injection, DL mutation operators have a different nature [18].

DeepMutation [27] and MuNN [38] were the first works to recognise the need for mutation operators tailored specifically to DL systems. In DeepMutation (later extended into a tool called *DeepMutation++* [14]), the authors propose a set of operators of two distinct categories: *source level* and *model level* operators. Source level operators apply changes to training data or model structure before training is performed, while model level operators alter weights, biases or the structure of an already trained model. Model level mutation operators tend to be less costly as, unlike source-level operators, they do not require re-training. Mutation operators proposed in MuNN [38], solely belong to the latter category.

Jahangirova & Tonella [18] performed an extensive empirical evaluation of the mutation operators proposed in DeepMutation++ and MuNN and investigated the configuration space of their parameters. For example, for a mutation operator that imitates training with corrupted data by changing the labels of training inputs to incorrect ones, the parameter would be the percentage of mutated labels. According to their results, the choice of the parameter values affects the impact of the mutation to a major extent.

Moreover, the authors propose a novel mutation killing criterion, which takes into account the stochastic nature of DL systems. Their definition requires multiple re-trainings of both the original program and the mutant to obtain n distinct model *instances* of each ($n = 20$ in their experiments). Then, they measure whether the difference between 20 accuracies (or any other quality metrics) obtained on original vs mutated model instances is statistically significant ($p\_value < 0.05$) and whether the effect size is not "negligible". If these conditions hold, the mutation is considered *killed*.

Table 1: Mutation Operators provided by *DeepCrime* [16] and not killed by the initial test sets of our case studies

Group	Mutation Operator	Mutation Parameters
Training Data	Change labels of training data (TCL)	*label* to perform the mutation on *percentage* of data to mutate
	Remove portion of training data (TRD)	*percentage* of data to delete
	Unbalance training data (TUD)	*percentage* of data to remove
	Add noise to training data (TAN)	*percentage* of data to mutate
	Make output classes overlap (TCO)	*percentage* of data to mutate
Hyperparams	Decrease learning rate (HLR)	*new learning rate* value
	Change number of epochs (HNE)	*new number of training epochs*
Activation	Change activation function (ACH)	*layer* w/ non-linear activ. function *new activation function*
	Remove activation function (ARM)	*layer* w/ non-linear activ. function
	Add activation function to layer (AAL)	*layer* w/ linear activation function *new activation function*
Regularisation	Add weights regularisation (RAW)	*layer* w/o weights regularisation *new weights regulariser*
Weights	Change weights initialisation (WCI)	*layer* to perform the mutation on *new weights initialiser*
Optimisation	Change optimisation function (OCH)	*new optimisation function*

2.2 DeepCrime

DeepCrime [16] is a mutation testing tool designed for automated seeding of artificial faults (mutations) into DL systems. Its main difference from DeepMutation++ is that *DeepCrime* is based on a set of mutation operators derived from *real faults*. In *DeepCrime* the authors propose 35 and implement 24 *source level* mutation operators that target different aspects of development and training of DL systems. This set of operators was extracted from an existing taxonomy of real faults in deep learning systems [15] and was complemented with the issues found in the replication packages for the studies by Islam et al. [17] and Zhang et al. [52]. To establish whether a mutation is killed or not, *DeepCrime* incorporates the notion of statistical killing proposed by Jahangirova & Tonella [18], using by default 20 re-trainings for the original model and for each of the applied mutations.

The mutation operators in *DeepCrime* have two types of parameters: continuous and non-continuous. For example, the operator that removes part of the training data has the continuous parameter *percentage*, which decides what portion of inputs should be deleted. Its value varies in the range 0% to 99% (as we cannot delete all training data). In contrast, mutation operators that operate on a per-layer basis have a non-continuous parameter *layer*, which determines the *specific layer* of a neural network to mutate.

In case the parameter values are not specified by a user, *DeepCrime* automatically computes the best configuration for the mutation operator. For non-continuous parameters, *DeepCrime* performs an exhaustive search by iterating through all of the possible values for a parameter. In case of continuous parameters, the computation is based on identifying the lowest and the highest possible values and performing a binary search in this range. The aim of the search is to discover the most challenging and yet killable configuration of the mutation operator for a given test suite. For example, for the operator *remove portion of training data* (*TRD* in Table 1) the binary search first checks if the most aggressive configuration (99%) is killed by the test data. If so, *DeepCrime* finds the middle point in the range of possible values (49.5%) and checks it for the killability. If the middle point gets killed, the search continues on the lower part of the range (0% - 49.5%); otherwise, on the upper half of the range (49.5% - 99%). This process is applied in a recursive manner till the

point when the size of a new range becomes smaller than or equal to the desired precision ϵ. The observed value of the percentage parameter that is not killed, which is ϵ-close to the least aggressive killable configuration, is the output of the binary search: this non killed mutant is the target of test generation.

The authors of *DeepCrime* also propose a definition of mutation score per operator. The definition is based on the assumption that training data is a set of inputs to which a trained model is the most sensitive. Given a test set *TS*, its mutation score (*MS*) is the proportion of configurations killed by both test and train set over those killed by the train set. It is calculated as:

$$MS(MO, TS) = \frac{|K(MO, TS) \cap K(MO, TRS))|}{|K(MO, TRS)|} \quad (1)$$

For example, if for the mutation operator *TRD* the least aggressive killed configuration found by binary search is 10% for the training data and 25% for the test data, the mutation score will be computed as: $MS = |[0.25 : 0.99]|/|[0.10 : 0.99]| = 0.74 / 0.89 = 0.83$.

The overall mutation score of the test suite is computed as the average of mutation scores across all operators.

The fact that *DeepCrime* offers a wide selection of mutation operators that are based on real DL faults and that it produces a statistically reliable outcome was the key motivation for us to choose this tool. The list of *DeepCrime*'s mutation operators (with their parameters) that produced mutants that were not killed by the test sets used in our case studies can be found in Table 1 (killed mutants are not the target of DeepMetis's input generation).

3 THE DEEPMETIS TECHNIQUE

DeepMetis aims to augment an existing test set, by extending it with mutant killing inputs that increase its mutation score. The Algorithm 1 describes the main steps implemented in DeepMetis to generate new inputs that kill mutants.

Starting from the original code of a DL model and an existing test set, DeepMetis leverages *DeepCrime* to obtain the configurations for which the considered mutation operator is not killed by the original test set (for continuous operators, this is the most aggressive non killed configuration found by binary search). *DeepCrime* injects the corresponding mutation into the model's code and produces multiple original model instances and mutants by executing n times the training process on the original and the mutated model's code, respectively (line 4). DeepMetis uses evolutionary search to generate new test inputs that can discriminate the original model instances from the mutants. The algorithm is based on NSGA-II [8], a multi-objective evolutionary search algorithm largely used in search-based software testing research [21, 29, 32, 36, 46, 47].

After initialising variables g, A, P_0, P (lines 6-13), the evolutionary steps are repeated for a given number of iterations, g_{max}. In each iteration, a population of individuals, i.e. test inputs, is evolved and their behaviour is evaluated against the original and mutant models. The result of such evaluation (lines 10 and 23) is the assignment of fitness values to individuals. Based on fitness values, the best individuals are identified and sorted by means of *crowding distance sorting* [8], a technique that accounts for both dominance between individuals according to the fitness values as well as the distance between individuals that belong to the same dominance front (lines 13 and 25). Then, we use tournament selection to select

Algorithm 1: Overall algorithm of DeepMetis

Input : ts_o: original test set
C: original DL program code
g_{max}: max number of generations
$popsize$: population size
$mutop$: mutation operator
n: number of re-training runs
o: number of original models
m: number of mutants
Output: ts_a: augmented test set

1 generate original models and mutants using *DeepCrime*
2 *original models* $M_o \leftarrow \emptyset$;
3 *mutant models* $M_m \leftarrow \emptyset$;
4 $M_o, M_m \leftarrow$ DeepCrime $(C, mutop, n, m, o)$;
5 start evolutionary search
6 *generation* $g \leftarrow 0$;
7 *archive* $A \leftarrow \emptyset$;
8 *initial population* $P_0 \leftarrow$ InitPopulation$(M_o, popsize)$;
9 *population* $P \leftarrow P_0$;
10 Evaluate(P, M_o, M_m);
11 $A \leftarrow$ UpdateArchive(P);
12 assign crowding distance to individuals
13 $P \leftarrow$ Select$(P, popsize)$;
14 **while** $g < g_{max}$ **do**
15 $g \leftarrow g + 1$;
16 selection based on dominance/crowding distance
17 *offspring* $Q \leftarrow$ SelTourDCD$(P, popsize)$;
18 substitute the most dominated/misbehaving on original models
19 $P \leftarrow$ Repopulation(P_0, A);
20 **foreach** $q \in Q$ **do**
21 $q \leftarrow$ Mutate(q) ;
22 **end**
23 Evaluate$(P \cup Q, M_o, M_m)$;
24 $A \leftarrow$ UpdateArchive$(P \cup Q)$;
25 $P \leftarrow$ Select$(P \cup Q, popsize)$;
26 **end**
27 augment the test set with the archived inputs
28 $ts \leftarrow ts_o \cup A$;
29 **return** (ts)

the surviving individuals Q (line 17), which are mutated by genetic operators (line 21). The worst individuals are replaced by means of the re-population operator, which re-introduces some of the initial seeds (P_0) into the current population P (line 19). When mutation killing inputs are generated, they are stored into an archive (lines 11 and 24). Finally, the archived solutions are used to augment the initial test suite (line 28). The test set improvement can be assessed by re-running *DeepCrime* to check if the previously non-killed configuration is now killed.

DeepMetis's evolutionary algorithm rewards individuals that behave correctly on original models and misbehave on mutants. DeepMetis is hybridised with novelty search, as it rewards also individuals that exhibit diversity of behaviours [25, 30]. It uses an archive to store the best solutions found during the search, in order to avoid cycling. It also uses re-population, to escape the stagnation in local optima with high basin of attraction.

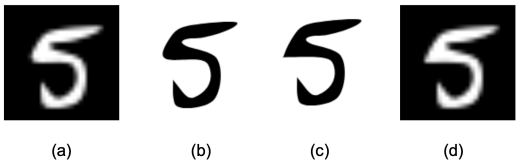

Figure 1: Digit input representation and mutation. (a) original input; (b) original SVG model after vectorization; (c) SVG model mutated by moving a control point; (d) mutated input

3.1 Model-Based Input Representation

DeepMetis belongs to the family of model-based test input generators [42], i.e., tools that manipulate an *input model* instead of directly modifying the input data (e.g., pixels). Input data derived from a model are more likely to be realistic and belong to the input validity domain [36] than data subjected to low level manipulation. This implies that DeepMetis is applicable to problems for which an input model is available. The development of input models is standard practice in several domains, such as cyber-physical systems, including safety-critical ones, such as automotive [22]. Below, we present models for domains we considered in our experimental evaluation: a vector image format for handwritten digit classifiers and a 3D human model for eye gaze predictors.

Digit Classification. We consider handwritten digit samples in the format adopted by the MNIST database [23]. Its inputs are originally encoded as 28×28 images, with greyscale levels that range from 0 to 255. As shown in Figure 1, we model them by adopting Scalable Vector Graphics (SVG)[1] as their representation. SVG is an XML-based vector image format for two-dimensional graphics that defines shapes as combinations of cubic and quadratic Bézier curves. The control parameters determining the shape of a modelled digit are: the start point, the end point and the control points of each Bézier curve. This representation helps in preserving the smoothness and curvature of handwritten shapes after minor manipulations of the curve parameters [36]. We use the Potrace algorithm [37] to transform an MNIST input into its SVG model representation. This algorithm performs a sequence of operations to obtain a smooth vector image starting from a bitmap. To transform an SVG model back into a 28×28 grayscale image, we perform the rasterisation operation by using two popular open source graphic libraries (i.e. LibRsvg[2] and Cairo[3]).

Gaze Prediction. We focus on the input format for the gaze estimator model proposed by Zhang et al. [50], which takes as an input an eye image and a *2D* head rotation angle (*yaw* and *pitch*) and predicts the eye gaze angle. The eye images are generated by exploiting *UnityEyes*, a freely available rendering framework [44]. Our eye model consists of all the independent parameters used by UnityEyes to generate an eye image. They can be divided into two groups: those that cover various aspects related to the eye appearance (head angle, eye angle, pupil size, iris size, iris texture, skin texture) and others that describe the lighting (texture, rotation, ambient intensity, exposure, for image-based lighting, and rotation and intensity, for directional lighting). Only some of these parameters are directly controllable when asking UnityEyes to generate new images, namely head rotation angles and eye rotation angles (the latter providing us the ground-truth for gaze prediction), while the others are decided internally by UnityEyes. All parameters are recorded by UnityEyes in a JSON file that accompanies the generated image. For each pair of head and eye angles (controllable parameters), it is possible to request UnityEyes to generate an arbitrary number of eye images, differing among each other by the remaining, not directly controllable, parameters. When manipulating UnityEyes' parameters for the purpose of test generation, we need to know the range in which each parameter falls, in order to ensure the validity of the manipulated values. Thus, for head and eye angles we use the ranges suggested in the *UnityEyes* interface. To learn the valid ranges for the remaining parameters, we generated a dataset of more than 1 million images and analysed the generated *JSON* files. The identified ranges and the script used for such analysis are available in our replication package [5].

[1]https://www.w3.org/Graphics/SVG/
[2]https://wiki.gnome.org/Projects/LibRsvg
[3]https://www.cairographics.org

3.2 Fitness Functions

DeepMetis optimises two fitness functions, which measure the ability of an individual to kill mutants and its diversity from the solutions already encountered during the search, respectively.

Mutation Killing. The fitness function f_1 measures how close an individual is to misbehave on mutants. In particular, for a given mutant m, its value is negative in the presence of a misbehaviour, while its value is positive and indicates the distance from a misbehaviour when the system behaves correctly. We estimate the distance from a misbehaviour as the model's confidence in the predicted class for classifiers, or the difference between the tolerable error and the actual prediction error for regressors. Hence, the lower the value assumed by such distance to misbehaviour, the higher the mutant's likelihood of misbehaving. To take into account the non-determinism of mutation and training, we generate and train n mutants. Correspondingly, the fitness value of an individual is computed as the sum of our misbehaviour closeness metric over n mutants. The fitness function f_1 has to be minimised:

$$\min \ f_1(x) = \min \sum_{m \in M_m} \text{eval}_m(x) \qquad (2)$$

To compute f_1 for an individual x, DeepMetis executes the n instances of mutant m with x as an input. The definition of function *eval* is clearly problem specific.

Digit Classification. The *eval* function exploits the classifier's output softmax layer, which can be interpreted as the confidence level assigned to each possible class. The predicted class corresponds to the highest confidence level and there is a misbehaviour when the expected class has a confidence level lower than another class. In particular, *eval* is calculated as the difference between the confidence associated with the expected class and the maximum confidence associated with any other class, when the prediction is correct; it is -1 otherwise.

Gaze Prediction. A misbehaviour is detected when the prediction error exceeds the maximum tolerated error. The prediction error is the difference between the model prediction and the expected prediction (provided as ground-truth by UnityEyes). Since predictions consist of a pair of eye rotation angles in radians (pitch and yaw),

the error is calculated as the angle between the expected vector and the predicted one. The maximum tolerated error can be set according to problem-specific requirements. In our study, we set it to 5 degrees, as this is an acceptable error in other gaze prediction applications [16, 50]. The value of f_1 is the difference between such acceptable threshold and the actual gaze prediction error.

Diversity. The fitness function f_2 represents an individual's sparseness with respect to individuals in the archive, and we want to maximise it:

$$\max \ f_2(x) = \max \text{spars}(x, A) \tag{3}$$

where A is the archive of solutions and x is the individual being evaluated. Function *spars* measures the minimum distance of an individual x from the solutions in the archive A: $\min_{y \in A, y \neq x} \text{dist}(x, y)$. The distance function (dist) is computed on pairs of inputs and is domain-specific. For *Digit Classification*, it is computed as the Euclidean distance between pixel vectors. In the *Gaze Prediction* problem, we use the genotypic distance, i.e., the distance between the chromosomes of two individuals, whose genes are the eye parameters used by UnityEyes. As in the chromosome there are float, vectorial and categorical gene values, to obtain an overall distance between chromosomes, we compute the distances between genes of the same type, normalise them separately and return the weighted sum of gene distances. In particular: for float genes we compute the difference d and normalize it as $d/(d+1)$; for the pitch and yaw angles' pairs, we calculate the angle between two vectors in radians (given the natural limits for eye rotation, the difference never exceeds 1 radian); for categorical genes we assign 0 to the distance if the genes contain the same category, 1 otherwise.

3.3 Initial Population

To obtain the initial population, we first gather a set of seeds, i.e., inputs on which the original models behave correctly. Then, we select the most diverse seeds, by computing pairwise distances and greedily constructing the set of most diverse seeds, starting from a randomly selected first seed, up to the desired population size. Then, initial individuals are obtained by applying a mutation genetic operator to each selected seed. We considered as seeds the samples in the training set on which the models behave correctly.

3.4 Archive of Solutions

The best individuals encountered during the search are kept in the archive of solutions [7]. This prevents the search for novelty from *cycling*, a phenomenon where the population moves from one area of the solution space to another and back again, without memory of the areas it has already explored [31]. At the end of the last iteration, the archive will contain the final solutions.

An individual of the population is a solution candidate to be included in the archive if it behaves correctly on at least one of the n original model instances and it triggers a misbehaviour on at least one mutant model instance. When a new candidate solution is found, it competes locally with similar solutions already in the archive so that only the best ones are kept, i.e., those with the lowest value of fitness function f_1.

In the archive used for *Digit Classification*, a solution competes with the archived inputs that are generated from the same MNIST seed. In the archive used for *Gaze Prediction*, we cannot rely on the starting seeds, as UnityEyes generates valid eye images from any random vector of controllable parameters, without requiring to evolve them from an initially valid seed solution. Hence, we had to define a similarity criterion for the archive used for *Gaze Prediction*: if the distance from the nearest neighbour in the archive is higher than a threshold t_a, the new individual is kept in the archive. Otherwise, the new candidate competes locally with its nearest neighbour in the archive. The threshold t_a is a parameter that can be adjusted by a tester to obtain a proper trade off between number of solutions that enter the archive and similarity/diversity of the archive. To empirically choose the value of t_a, we recommend to (1) compute the minimum distance among a randomly selected set of diverse inputs; (2) choose a value greater than this number; (3) iteratively adjust this value based on the corresponding archive size and similarity.

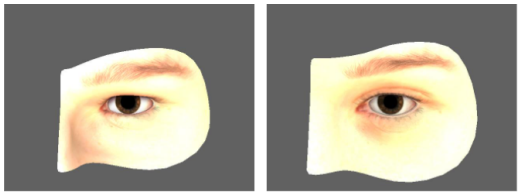

Figure 2: Eye input mutation

3.5 Genetic Operators

In multi-objective evolutionary algorithms there are multiple dimensions (in our case, f_1 and f_2) on which to compare the individuals. We use the Selection operator from NSGA-II [8], which applies Pareto front analysis and promotes individuals that are not dominated by any other individual. This operator favours individuals with smaller non-domination rank and, when the rank is equal (i.e., they belong to the same Pareto front), it encourages diversity by favouring the one in a less dense region. The offspring of the current population is obtained through the tournament selection with the tournament size equal to 2, by choosing the best between each pair of individuals being compared.

Each offspring individual is mutated by the Mutation genetic operator, which is domain specific. For *Digit Classification*, the mutation genetic operator randomly chooses an SVG model's point and applies a displacement to it in one of the four directions in the 2D space. Then, the rasterisation operation is applied to obtain the new digit image. For *Gaze Prediction*, the mutation genetic operator randomly chooses a gene from the individual's chromosome and applies a displacement to its value. Then, an input image that corresponds to the new values of the eye model's parameters is supposed to be generated. However, since only a small subset of parameters can be controlled in UnityEyes, DeepMetis generates a high number of images and JSON file pairs (~200) under the desired controllable parameters. From these pairs, it selects the one that is closest to the desired mutant chromosome, checking that it has never been used before during the search. Figure 2 shows an original eye image (left) and the corresponding mutated image (right) obtained by maintaining the controllable parameters unchanged.

During the search, exploration could get stuck in the local optima, despite the use of fitness function f_2 to promote diversity. To mitigate this situation and further vary the population, DeepMetis uses the Repopulation genetic operator, which replaces the individuals in the population that are behaving incorrectly on all the considered original model's instances. The repopulation operator replaces also a fraction of the most dominated individuals in the current population. The aggressiveness of this operator can be tuned by setting the range from which such fraction is uniformly sampled. The new individuals are generated starting from a randomly chosen seed. Repopulation is applied when the archive is not empty.

4 EXPERIMENTAL EVALUATION

4.1 Subject Systems

We ran our experiments on two subject systems for which a model of the input is available and can be manipulated via our genetic operators: MNIST and UnityEyes.

MNIST is a publicly available dataset consisting of 70,000 images of hand-written digits. Typically, 60,000 images are used for training, and the remaining 10,000 for testing. The DL system consists of a DNN model that predicts which digit is represented by an input image. We considered the deep convolutional neural network (CNN) provided by Keras,[4] because of its popularity, simplicity and effectiveness (99.15% test accuracy).

For the gaze prediction case study based on UnityEyes, we use a multimodal CNN [1], which provides an implementation based on the LeNet network architecture [23] following the approach described in the work by Zhang et al. [50]. The CNN learns the mapping from an eye image and a *2D* head angle (*pitch* and *yaw*) to a *2D* eye gaze angle. The dataset that we used for training and testing is supplied along with the model and consists of 129,285 eye region images (with 103,428 images used for training and 25,857 for testing) synthesised with *UnityEyes* [44]. Each image generated by *UnityEyes* is accompanied by a *JSON* file describing *2D* head angle, eye gaze vector, as well as other parameters used to generate the image, such as skin texture and various lighting features. When presented to a model for training and prediction, images are converted to grayscale and cropped to 60×36 pixels. The head angle and eye angle, which represent the second input to the model and the ground truth, respectively, are converted into radians.

4.2 Research Questions

We have performed a set of experiments to answer the following research questions:

RQ1 (Effectiveness): *Can DeepMetis generate inputs that improve a given test set, in terms of mutation killing capability?*

To answer this research question, for each of our subject systems we need an initial test set that we will then improve with the help of DeepMetis. The original test sets available for these subjects are very large in size and successful in terms of mutation score (100% for MNIST and 92.5% for UnityEyes). We therefore had to artificially construct a weaker test set for our case studies. For MNIST we did so by removing the test inputs that are predicted with low confidence (i.e., confidence less than 1) from the original test set. The elimination of such inputs leads to a test set with smaller discriminative power, as low confidence inputs typically represent difficult, corner cases that are effective at discriminating a mutant from the original model. For UnityEyes, which solves a regression, not a classification problem, we instead removed inputs with the smallest standard deviation of loss measured across 20 instances of the original model. Such inputs are very discriminative, as mutants typically amplify the standard deviation of the error observed for the original model, so the effect is more visible when we start from a small standard deviation. A similar approach to construct weak test sets for both classification and regression systems was adopted in Humbatova et al. [16]. The approach we used for classification systems has also been previously used in the work by Jahangirova and Tonella [18] for weak test set construction and by Byun et al. [6] for test input prioritisation. The size of the weak test set for MNIST is 4,813 elements and for UnityEyes it is 4,000 elements.

We then performed mutation testing of our subject systems considering the constructed weak test sets and using *DeepCrime*. Out of the 24 mutation operators implemented in *DeepCrime*, 18 were applicable to MNIST and 17 to UnityEyes. For operators with non-continuous parameters, we applied every value from the list exhaustively. For operators with continuous parameters, we performed the binary search on the full range of the parameter value space. We adopted the statistical notion of mutation killing [18], using Wilcoxon test to calculate the p-value and Cohen's d to measure the effect size. According to our procedure, statistical significance is reached when $p\_value < 0.05$ and effect size is greater than "small", i.e., Cohen's $d \geq 0.5$. Overall, we got 71 not killed mutants (i.e., instances of *DeepCrime*'s mutation operators) for MNIST and 38 for UnityEyes.

As mutation testing suffers from the problem of equivalent mutants, it is possible that some of the mutants not killed by our weak test set are not killable by any test set, and therefore our attempts for generating inputs that kill these mutants are vain. To avoid this situation, we use the definition of "likely equivalent" mutants proposed by Humbatova et al. [16]. According to this definition, if a mutant is not killed by the training data (i.e. the data the mutant should be most sensitive to, as the mutant was trained on such data) then this mutant is deemed *likely equivalent*. After filtering out the likely equivalent mutants, we were left with 19 mutants for MNIST and 10 for UnityEyes. The 19 MNIST mutants are instances of 12 different mutation operators, while for UnityEyes 10 mutants are produced by 9 mutation operators. To make our experiments feasible, we further reduced the set of MNIST mutants by picking only one instance for each mutation operator.

We applied DeepMetis to each of the 22 mutants. We first ran the initial population generation process 10 times to obtain 10 different populations. We then invoked the input generation process for each pair of mutant and initial population, getting as a result 10 runs of DeepMetis on each mutant, to account for the non-deterministic search-based nature of our tool. In these experiments, DeepMetis is run in the 1vs5 (1 original vs 5 mutants) configuration. This means that the number of mutants used by the fitness function f_1 (see Equation 2) is 5. The next research question investigates other, alternative configurations of our tool.

RQ2 (Fitness Guidance): *How does the fitness function based on a single mutant instance compare to the fitness function based on*

[4] https://keras.io/examples/vision/mnist_convnet/

Table 2: DeepMetis Configurations

Parameter	MNIST	UnityEyes
population size	100	12
generations	1000	100
archive threshold t_a	-	0.55
repopulation upper bound	10	2

multiple mutant instances in guiding DeepMetis towards generation of mutation killing inputs?

The aim of this research question is to identify whether providing more instances of the same mutant to DeepMetis increases its success in generating mutation-killing inputs. For this purpose we ran DeepMetis in 4 different modes by providing it with either 1, 5, 10 or 20 instances of the same mutation (i.e., we configure it as 1vs1, 1vs5, 1vs10 and 1vs20).

Similarly to RQ1, we perform 10 runs using 10 different initial populations. We do not evaluate extensively the effect of increasing the number of instances of the original model (e.g., 5vs5 or 10vs10), as preliminary experiments showed that the effect of such alternative choices is negligible on the effectiveness of the fitness function, while at the same time increasing substantially the overall computation time.

RQ3 (Comparison with other Tools): *Can we use existing DL input generators to achieve comparable improvement in the mutation killing capability of a test set?*

To answer this research question, we compare DeepMetis to two state of the art test input generators for DL systems: *DeepJanus* [36] and *DLFuzz* [12]. *DeepJanus* is a model-based tool that uses a multi-objective evolutionary algorithm to generate frontier inputs for DL systems. The *frontier inputs* are defined as pairs of inputs that are similar to each other but trigger different behaviours of the DL system. The idea is that for a low quality DL system, such a frontier will include pairs that intersect the validity domain, while for a high quality one it will have a small or no intersection at all. In our experiments, we passed *DeepJanus* one instance of the original model and from the generated set of pairs of inputs we use only those inputs that do not trigger any misbehaviour in the original model, as our goal is to obtain inputs that behave correctly on the original models but misbehave on the mutated ones. Another option could be passing *DeepJanus* the mutated model and then using the misbehaving set of inputs. However, some preliminary runs showed that the misbehaving inputs for the mutant almost never behave correctly on the original model. Therefore, we excluded this setup from our comparison study.

DLFuzz is representative of search-based fuzzing testing tools that generate test inputs by applying perturbations to the raw input (i.e., pixels) [9]. It aims to generate adversarial inputs that maximise neuron coverage for the DL system under test. For this purpose, *DLFuzz* iteratively selects neurons, the activation of which would lead to increased neuron coverage, and applies perturbations to test inputs in order to activate those neurons, so guiding the DL systems towards exposing misbehaviours. The publicly available version of *DLFuzz*[5] does not support regression systems, therefore we could not apply it to UnityEyes. Moreover, this implementation does not work with Python versions higher than 2.7.1, so we had to update the code to make it compatible with Python 3.8.

Similarly to DeepMetis, both *DeepJanus* and *DLFuzz* are affected by randomness, so we performed 10 runs of each tool, each run using a different initial population. However, we fixed the same population across runs of different tools, to ensure that the differences in their performance are not due to the different starting points of the respective algorithms. As explained before, *DeepJanus* uses the original model in its generation process, not requiring a re-run for each mutant. In contrast, as *DLFuzz* generates only inputs that get misclassified by the given DL model, we had to use the mutants. As a result, *DLFuzz* had to be re-run for each considered mutant. Overall, we performed 20 runs of *DeepJanus* (10 populations for the original model of both MNIST and UnityEyes), 120 runs of *DLFuzz* (10 populations for 12 MNIST mutants) and 220 runs of DeepMetis (10 populations for 22 MNIST and UnityEyes mutants).

For both tools, we used the configuration reported as the one achieving the best performance by their authors.

RQ4 (Fault Detection): *Can the test set augmented by DeepMetis expose more faults than the original test set?*

This research question analyses whether DeepMetis delivers its promise of improving the test set so that it detects more faults. Since, to the best of our knowledge, there is no publicly available dataset of real faults for DL systems, we use *DeepCrime* mutants as a replacement for real faults, in a cross-validation setup.

Specifically, we perform cross-validation by leaving one of the mutants out and augmenting the test set with all the inputs generated by DeepMetis for the remaining mutants. We ensure that none of the remaining mutants are generated by the same mutation operator as the cross-validation mutant, assuming that mutants produced by the same operator may have similar properties. We then check if the augmented test set is able to kill the cross-validation mutant. This process is repeated separately for the inputs generated in each of the 10 runs of DeepMetis. We added the previously excluded 7 MNIST mutants to this analysis, as, although there are no inputs generated specifically for them, they can still serve as cross-validation mutants. Before proceeding with the experiment, we performed a redundancy analysis [16] among the mutants of each subject, to ensure that inputs generated for one mutant do not kill another mutant just because the latter is redundant with respect to the former. Redundancy analysis showed that all 10 UnityEyes mutants are non-redundant, while for MNIST 6 out of 19 mutants are redundant. We excluded redundant mutants from further analysis, i.e., we did not use them as cross-validation mutants.

4.3 Results

Columns *Subject* and *MO* in Table 3 indicate the DL system and the mutation operator which provided the mutants used by DeepMetis for test input generation. For each operator, we report in brackets the parameter values which were found by binary/exhaustive search and were used to generate the non killed mutant. For mutation operators that manipulate the training data, this value indicates the ratio of the affected data. For example, *MNIST/TRD* removes 89.72% of the training data. For the other operators, parameter values with prefix 'l' followed by a number indicate the layer to which a

[5] https://github.com/turned2670/DLFuzz

Table 3: Results: column *K* (killing probability) reports mutation score, for continuous operators, and binary killed/non-killed outcome, for discrete operators, both averaged across 10 runs

Subject	MO	Weak TS	DeepMetis (1vs1)		DeepMetis (1vs5)		DeepMetis (1vs10)		DeepMetis (1vs20)		DeepJanus		DLFuzz	
		K	Inputs	K	Inputs	K	Inputs	K	Inputs	K	Inputs	K	Inputs	K
MNIST	TCL (84.38%)	13%	21	92%	16	87%	18	90%	20	86%	8	62%	61	91%
	TRD (89.72%)	6%	48	82%	40	89%	45	88%	18	78%	8	60%	119	85%
	TUD (90.62%)	6%	23	78%	17	77%	20	73%	22	73%	8	18%	61	68%
	TAN (100%)	0%	19	63%	19	81%	21	74%	22	79%	8	37%	67	43%
	TCO (96.88%)	0%	14	49%	14	59%	17	69%	50	60%	8	29%	39	48%
	HLR (0.064)	0%	42	85%	26	86%	27	86%	30	86%	8	70%	110	86%
	HNE (1)	0%	47	87%	30	89%	35	90%	40	90%	8	64%	110	96%
	ACH (l6; 'sigmoid')	0%	20	100%	18	100%	23	100%	27	100%	8	100%	136	100%
	ARM (l5)	0%	12	90%	11	100%	12	100%	14	90%	8	10%	91	100%
	RAW (l0; 'l1_l2')	0%	15	100%	11	100%	15	100%	16	100%	8	100%	52	100%
	WCI (l0; 'ones')	0%	21	100%	14	90%	24	100%	28	100%	8	80%	170	100%
	OCH ('rmsprop')	0%	15	100%	13	100%	18	100%	23	100%	8	100%	80	100%
	Average	**2%**	**25**	**86%**	**19**	**89%**	**23**	**89%**	**26**	**87%**	**8**	**61%**	**91**	**85%**
UnityEyes	TCL (21.88%)	86%	477	86%	536	88%	562	86%	604	86%	76	86%	-	-
	TRD (41.66%)	67%	335	79%	515	87%	70	67%	74	67%	76	67%	-	-
	TUD (100%)	0%	496	100%	587	100%	595	100%	662	100%	76	40%	-	-
	TAN (84.38%)	25%	379	15%	546	40%	669	38%	611	63%	76	25%	-	-
	HLR (0.0037)	39%	480	54%	557	61%	563	65%	597	72%	76	41%	-	-
	HNE (32)	70%	318	70%	454	72%	514	69%	528	70%	76	70%	-	-
	AAL (l9; 'signsoft')	0%	40	0%	392	90%	402	60%	533	60%	76	0%	-	-
	RAW (l1; 'l2')	0%	48	0%	480	60%	517	70%	557	60%	76	0%	-	-
	RAW (l3; 'l2')	0%	40	0%	494	40%	549	60%	568	50%	76	0%	-	-
	WCI (l1; 'ones')	0%	444	0%	611	40%	613	60%	123	0%	76	0%	-	-
	Average	**29%**	**306**	**40%**	**517**	**68%**	**505**	**68%**	**486**	**63%**	**76**	**33%**	**-**	**-**

mutation operator was applied. All the other parameters specify the exact value used to inject the fault. For example, *MNIST/ACH (l6; 'sigmoid')* means that the activation function of layer number 6 was changed from the original to the 'sigmoid' one.

In Table 3, the sub-columns *K* indicate the *killing probability*, computed as the mutation score (see Equation (1)) for continuous operators or as the binary killed/non-killed outcome for discrete operator instances (since we did not apply DeepMetis to all instances of discrete operators, Equation (1) cannot be used for them). Column *Weak TS* shows the killing probability *K* of the initial, weak test set. In the following columns, the sub-column *Inputs* shows the average number of inputs generated across 10 runs by each tool/tool configuration, while the sub-column *K* shows the average killing probability of the test set augmented with the generated inputs, computed across 10 runs.

4.3.1 RQ1: Effectiveness. The results for DeepMetis in its best configuration (1vs5) show that for both subjects the augmentation of the initial test set with the DeepMetis-generated inputs leads to a substantial increase of the mutation score. For MNIST, the improvement across the operators varies between 59% and 100%, with the average *K* jumping from 2% to 89%. For UnityEyes, the improvement ranges from 2% to 100% on a per operator basis, and the average *K* rises from 29% to 68%. The number of generated inputs, which would require manual labeling, is 19 on average for MNIST and 517 for UnityEyes. As these numbers constitute only 0.0003% of the training data set size for MNIST and 0.005% for UnityEyes, we consider the labelling effort associated with DeepMetis to be low.

> **RQ1**: DeepMetis is able to achieve a substantial improvement in killing probability on each of the provided mutants. The magnitude of this improvement is 87% for MNIST and 39% for UnityEyes. The manual labeling effort for the newly generated inputs can be deemed acceptable.

4.3.2 RQ2: Fitness Guidance. Columns DeepMetis (1vs1), DeepMetis (1vs5), DeepMetis (1vs10), DeepMetis (1vs20) report the results obtained when the fitness function uses respectively 1, 5, 10 and all 20 instances of a mutant during the input generation process. In the case of MNIST, for 3 mutants out of 12, 1vs1 and 1vs5 provide the same results. For 6 operators 1vs5 performs better; however, for 2 out of those the improvement is marginal (1-3%). For the remaining 3 operators 1vs1 outperforms 1vs5, with the difference for one of the operators being only 2%. When we further compare 1vs5 to 1vs10, the latter exhibits an improvement for 4, equal performance for 5 and deterioration for 3 operators, while being substantially more expensive computationally. Overall, as also reflected in the average *K* across operators, for MNIST the optimal performance is obtained with 1vs5 and 1vs10 settings, which provide slightly better results than 1vs1 and 1vs20.

The results for UnityEyes show that 1vs5 and 1vs10 produce the same average *K* (68%), which is slightly better than 1vs20 (63%), but is definitely superior when compared to 1vs1 (40%). On a closer

inspection, 1vs5 outperforms 1vs10 and 1vs20 on 5 mutants out of 10, with the majority of them being continuous operators, while 1vs10 is the best in 3 cases and 1vs20 in 2. As was noted, 1vs20 on average performs similarly to 1vs5 and 1vs10; however, in one case (*WCI (l1; 'ones')*) it fails to produce any improvement at all.

The reason behind the comparative weakness of 1vs1 w.r.t. the other settings is that its fitness function has a very limited range, because it aggregates the *eval* value of a single mutant, which provides limited guidance to the test generation process.

The input generation for our experiments was performed on various machines. This complicates the comparison of the execution time between different configurations of DeepMetis. However, for each subject we ensured to run all 4 configurations on the *TUD* operator (selected randomly) using the same machine. For MNIST we used a MacBook Pro laptop (2.2 GHz Intel Core i7, 6 cores, 16GB RAM), while for UnityEyes we used Alienware Aurora R8 (3.60 GHz Intel Core i9-9900K, 8 cores, 32GB RAM, NVIDIA GeForce RTX 2080 Ti 11 GB). For MNIST this operator took 6, 22, 47 and 57 minutes on average across 10 runs for 1vs1, 1vs5, 1vs10 and 1vs20, respectively. For UnityEyes the generation of inputs for one run on average lasted 53 (1vs1), 66 (1vs5), 65 (1vs10), and 69 (1vs20) minutes. These results show that 1vs5 is the optimal setting in terms of balancing the improvement in mutation score and the time required to generate the inputs.

> **RQ2**: The 1vs5 configuration of DeepMetis proved to be the optimal one. It outperforms 1vs1 by a substantial margin, as a single mutant (1vs1) cannot provide enough guidance to generate effective inputs. The settings with a higher number of mutants are sometimes comparable, but they might require significantly more computation time.

4.3.3 RQ3: Comparison with other Tools. Columns *DLFuzz* and *DeepJanus* in Table 3 report the results for each of the tools being compared to DeepMetis. In case of MNIST, for 9 out of 12 mutants DeepMetis (1vs5) performs better than *DeepJanus*, while for the remaining mutants they have similar performance. The average K across all mutants for DeepMetis (1vs5) is higher by 28% than for DeepJanus. When it comes to the comparison between DeepMetis and *DLFuzz*, DeepMetis provides better results for 4 mutants, *DLFuzz* for 3 mutants, and the outcome is equal for the remaining 5. The average K across all mutants is 89% for DeepMetis (1vs5) and 85% for *DLFuzz*. However, *DLFuzz* generates 4.8 more inputs than DeepMetis (1vs5) and therefore requires much more manual labelling effort.

As *DLFuzz* is not applicable to regression problems, the comparison for the UnityEyes subject was only possible between DeepMetis and *DeepJanus*. Results show that *DeepJanus* is not able to produce any improvement in the majority of the cases. The only exceptions are *TUD* and *HLR* operators, where for the former the average improvement is 40%, compared to 100% of DeepMetis (1vs5), and for the latter the improvement of *DeepJanus* is limited to 2% vs 22% of DeepMetis.

We performed statistical analysis on the comparison of the results by each tool. For mutants with continuous parameters we used the Wilcoxon statistical test to obtain a p-value and the Vargha-Delanay $\hat{A}_{12}$ to quantify the effect size. For mutants with non-continuous parameters we calculate confidence intervals using Wilson's method. When comparing DeepMetis and *DeepJanus* for MNIST, the difference is statistically significant (p-value < 0.05 or confidence intervals do not intersect) for 7 mutants out of 12. For 5 out of 7 mutants with continuous parameters, the effect size is large, for 1 mutant it is medium and for the remaining one it is small. In case of DeepMetis and *DLFuzz*, there is a statistically significant difference two mutants. The effect size is negligible for 1, small for 3, medium for 2 and large for 1 mutant. The results of the comparison of DeepMetis (1vs5) and *DeepJanus* on the UnityEyes subject are statistically significant for 5 out of 10 applied mutants. For the 6 mutants with continuous parameters, the effects size ranges between large (3), small (2) and negligible (1).

When it comes to execution time comparison (conducted in the same conditions as described for RQ2), for MNIST DeepMetis took on average 22 minutes, *DeepJanus* 9 minutes and DLFuzz 24 minutes. For UnityEyes, DeepMetis took about 66 minutes on average and *DeepJanus* about 86 minutes.

> **RQ3**: DeepMetis outperforms *DLFuzz* and *DeepJanus* in the task of augmenting a test set to improve its mutation score.

4.3.4 RQ4: Fault Detection. Results are presented in Table 4, where column *MO* specifies the cross-validation mutant, used to check the hypothesis that DeepMetis generated inputs are also able to kill other, previously unseen mutants. Column *Inputs* indicates the average number of inputs across 10 runs that were generated by DeepMetis and added to the original weak test set. Column *Killed* reports the proportion of runs (out of 10) in which the augmented the test was able to kill the validation mutant.

For MNIST almost all validation mutants were killed in all 10 runs, with the exception of ARM (l5) and RAW (l0, 'l2') that were killed in 8 runs and WCI (l0; 'random_uniform') that is killed in 1 run. The latter is an almost equivalent mutant, with a very low triviality score [16], which is very difficult to kill for DeepMetis. The results for UnityEyes also indicate that DeepMetis is always able to kill the unseen mutant at least once. For 5 mutants out of 10, the test set augmented with DeepMetis inputs killed the mutant in 100% of the runs. In all other cases except for *TRD (46.41%)* and *HNE (32)*, the augmented test set succeeds in 5 to 6 out of 10 runs.

> **RQ4**: The mutation killing capability of the DeepMetis-generated inputs holds also for previously unseen mutants, with 82% average success rate across our two subjects.

4.4 Threats to Validity

Construct Validity: The choice of the distance metrics may threaten our findings. We chose sound metrics for the considered domains. We used Euclidean distance when comparing matrices of grayscale values (also used in previous studies [36]). When comparing UnityEyes inputs, we used a combination of appropriate distances for each gene type in the chromosome.

Table 4: Fault Detection

Subject	MO	Inputs	Killed
MNIST	TCL (84.38%)	107	10/10
	TUD (90.62%)	106	10/10
	TCO (96.88%)	109	10/10
	HLR (0.064)	98	10/10
	ACH (l6; 'hard_sigmoid')	123	10/10
	ACH (l6; 'softplus')	123	10/10
	ACH (l6; 'softmax')	123	10/10
	ARM (l5)	112	8/10
	RAW (l0; 'l1_l2')	112	10/10
	RAW (l0; 'l2')	112	8/10
	WCI (l0; 'ones')	109	10/10
	WCI (l0; 'random_uniform')	109	1/10
	OCH ('rmsprop')	111	10/10
UnityEyes	TCL (21.88%)	4635	10/10
	TRD (46.41%)	4655	1/10
	TUD (100%)	4584	10/10
	TAN (84.38%)	4625	10/10
	HLR (0.0037)	4613	10/10
	HNE (32)	4717	3/10
	AAL (l9; 'signsoft')	4779	6/10
	RAW (l1; 'l2')	4197	10/10
	RAW (l3; 'l2')	4197	5/10
	WCI (l1; 'ones')	4559	6/10

Internal Validity: The main threat affecting the internal validity of our results is the choice of mutation operators and mutation tool. We use *DeepCrime*, a DL mutation tool that accounts for the stochastic nature of DL systems and DL specific mutation operators by adopting the statistical notion of mutation killing. Moreover, its operators are derived from real DL faults that ensures a higher degree of realism as compared to alternative possible choices.

External Validity: The choice of the subject DL systems is a possible threat to the *external validity*. To mitigate it, we chose two diverse DL systems. One solves a classification problem, while the other solves a regression problem. A wider set of systems (including industrial ones) should be considered in future studies to further generalise our findings.

To ensure **Reproducibility** of our results, we share online the source code of DeepMetis, the considered subjects, and the experimental data [5].

5 RELATED WORK

5.1 Test Generation for DL Systems

Several works in the literature [12, 24, 33, 40, 49] propose techniques that generate test inputs for DL systems by manipulating raw input data, i.e. they apply small perturbations to available real inputs. A limitation of these approaches is the lack of realism of the generated inputs. While these corrupted images are useful for security testing as adversarial attacks, they are not necessarily representative of data captured by sensors of a real DL system.

Another family of testing techniques [2–4, 10, 36, 41] adopts a model-based approach that exploits model manipulation and model-based generation. Differently from raw input manipulation approaches, these techniques tend to generate more realistic inputs if a faithful model of the input domain is adopted, since the generated images are compliant with the constraints of such model. In this work, we adopt a model-based approach to improve the realism of the generated inputs.

Pei et al. propose a raw input manipulation technique aimed at generating inputs that trigger inconsistencies between multiple DL systems [33]. Other techniques manipulate raw images and consider as failures the inconsistent behaviours triggered by the original and a transformed test input [12, 24, 40, 49].

Model-based approaches, proposed by Abdessalem et al. [2–4] and Gambi et al. [10], aim to test advanced driver-assistance systems by generating extreme and challenging scenarios that maximise the number of detected system failures.

Riccio and Tonella proposed a model-based approach that produces test suites made of pairs of inputs that identify the frontier of behaviours of a DL system, i.e. the inputs at which the DL system starts to misbehave [36].

Udeshi et al. generate inputs that highlight fairness violations by perturbing discriminatory parameters, e.g. gender [41].

Vahdat Pour et al. [34] use DL mutation to guide the generation of adversarial code snippets for DL models tailored to the computation of code embeddings.

DeepMetis differs from existing approaches because its goal is to increase the mutation killing ability of a test set. With the advent of DL mutation frameworks, such as DeepMutation [27], MuNN [38] and DeepCrime [16], the problem of achieving a high mutation score is increasingly important, especially when mutants mimic real faults, as is the case of DeepCrime [16].

DeepMetis is the first approach that can assist developers in the challenging task of making a DL test set better at mutation killing.

5.2 Test Adequacy for DL Systems

Several test adequacy criteria have been proposed for DL systems.

Pei et al. [33] use the number of neuron activations of the model to measure test adequacy. In particular, a neuron is considered activated if its output value is higher than a predefined threshold.

Ma et al. [26] propose a set of additional adequacy criteria based on neuron activations. They use activation values obtained from the training data and divide the range of values for each neuron into k buckets.

Kim et al. [20] designed a test adequacy criterion, named *surprise adequacy*, based on the degree of "surprise" of an input for the neural network. Similarly to Ma et al.'s criteria [26], bucketing is used to make the surprise measure an adequacy criterion: all k buckets of surprise ranges must be covered by the test set.

X. Zhang et al. [51] observe how inputs are distributed across different uncertainty patterns, i.e. combinations of alternative uncertainty metrics (e.g., high prediction confidence and low variation ratio). Although they do not define a proper adequacy criterion, they recommend to generate additional test inputs to cover the least covered uncertainty patterns and they show that such inputs evade defenses against adversarial attacks.

We adopt a test set's mutation score as an adequacy criterion. Like other criteria [20, 26, 33], our criterion uses as a reference the training set, since it contains the inputs to which the model

is mostly sensitive: the mutation score of the test set should be as close as possible to the one of the training set.

Jahangirova & Tonella [18] compared mutation score to other adequacy metrics such as neuron coverage [33] and surprise coverage [20], showing that mutation score is more effective in differentiating between weak and strong test sets than the existing alternatives.

DeepMetis is the first tool that uses mutation adequacy as guidance for the generation of inputs that increase the mutation score of an existing weak test set.

6 CONCLUSIONS AND FUTURE WORK

We proposed DeepMetis, the first automated test generator for DL systems that can increase the mutation score of a weak test set, guided by mutation adequacy. Our empirical evaluation shows that our tool outperforms state-of-the-art DL test generators in this task. The test sets generated by DeepMetis can expose unknown faults, simulated in our leave-one-out experiment by means of previously unseen mutants. In our future work, we plan to generalise our results to a wider sample of DL systems, including industrial ones.

REFERENCES

[1] 2020. An implementation of a multimodal CNN for appearance-based gaze estimation. https://github.com/dlsuroviki/UnityEyesModel.

[2] 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*. 63–74.

[3] 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

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

[5] anonymized. 2021. Replication package for DeepMetis. https://drive.google.com/drive/folders/1a2sdImhU0O48DQzn9Fo-eEvU-OTZf96t.

[6] Taejoon Byun, Vaibhav Sharma, Abhishek Vijayakumar, Sanjai Rayadurgam, and Darren Cofer. 2019. Input prioritization for testing neural networks. In *2019 IEEE International Conference On Artificial Intelligence Testing (AITest)*. IEEE, 63–70. https://doi.org/10.1109/AITest.2019.000-6

[7] Edwin D. de Jong. 2004. The Incremental Pareto-Coevolution Archive. In *Genetic and Evolutionary Computation – GECCO 2004*, Kalyanmoy Deb (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 525–536.

[8] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation* 6, 2 (April 2002), 182–197. https://doi.org/10.1109/4235.996017

[9] Swaroopa Dola, Matthew B Dwyer, and Mary Lou Soffa. 2021. Distribution-Aware Testing of Neural Networks Using Generative Models. *arXiv preprint arXiv:2102.13602* (2021).

[10] 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*. 318–328.

[11] Jahangirova Gunel, Stocco Andrea, and Tonella Paolo. 2021. Quality Metrics and Oracles for Autonomous Vehicles Testing. In *2021 IEEE 14th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE.

[12] 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*. 739–743.

[13] Fabrice Harel-Canada, Lingxiao Wang, Muhammad Ali Gulzar, Quanquan Gu, and Miryung Kim. 2020. Is neuron coverage a meaningful measure for testing deep neural networks?. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*. 851–862.

[14] 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)*. IEEE, 1158–1161.

[15] 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 42nd International Conference on Software Engineering (ICSE '20)*. ACM, 12 pages.

[16] 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*.

[17] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A Comprehensive Study on Deep Learning Bug Characteristics. 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)*. ACM, New York, NY, USA, 510–520. https://doi.org/10.1145/3338906.3338955

[18] 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.

[19] Y. Jia and M. Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering* 37, 5 (2011), 649–678. https://doi.org/10.1109/TSE.2010.62

[20] 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*. 1039–1049.

[21] Kiran Lakhotia, Mark Harman, and Phil McMinn. 2007. A Multi-objective Approach to Search-based Test Data Generation. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation* (London, England) *(GECCO '07)*. ACM, New York, NY, USA, 1098–1105. https://doi.org/10.1145/1276958.1277175

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

[23] 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.

[24] Seokhyun Lee, Sooyoung Cha, Dain Lee, and Hakjoo Oh. 2020. Effective White-Box Testing of Deep Neural Networks with Adaptive Neuron-Selection Strategy. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual Event, USA) *(ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 165–176. https://doi.org/10.1145/3395363.3397346

[25] 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

[26] 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

[27] Lei Ma, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Felix Juefei-Xu, Chao Xie, Li Li, Yang Liu, Jianjun Zhao, and Yadong Wang. 2018. DeepMutation: Mutation Testing of Deep Learning Systems. In *29th IEEE International Symposium on Software Reliability Engineering, ISSRE 2018, Memphis, TN, USA, October 15-18, 2018*. 100–111. https://doi.org/10.1109/ISSRE.2018.00021

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

[29] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective Automated Testing for Android Applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. ACM, New York, NY, USA, 94–105. https://doi.org/10.1145/2931037.2931054

[30] 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

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

[32] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2018. Automated Test Case Generation as a Many-Objective Optimisation Problem with Dynamic Selection of the Targets. *IEEE Transactions on Software Engineering* 44, 2 (2018), 122–158.

[33] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. DeepXplore: Automated Whitebox Testing of Deep Learning Systems. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 1–18.

[34] 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 *IEEE International Conference on Software Testing, Verification and Validation (ICST'21)*. IEEE, 11 pages.

[35] 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

[36] Vincenzo Riccio and Paolo Tonella. 2020. Model-Based Exploration of the Frontier of Behaviours for Deep Learning System Testing. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 876–888. https://doi.org/10.1145/3368089.3409730

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

[38] W. Shen, J. Wan, and Z. Chen. 2018. MuNN: Mutation Analysis of Neural Networks. In *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. 108–115. https://doi.org/10.1109/QRS-C.2018.00032

[39] Jeongju Sohn, Sungmin Kang, and Shin Yoo. 2019. Search Based Repair of Deep Neural Networks. *arXiv preprint arXiv:1912.12463* (2019).

[40] 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

[41] Sakshi Udeshi, Pryanshu Arora, and Sudipta Chattopadhyay. 2018. Automated Directed Fairness Testing. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE 2018)*. ACM, New York, NY, USA, 98–108. https://doi.org/10.1145/3238147.3238165

[42] 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.

[43] Jingyi Wang, Guoliang Dong, Jun Sun, Xinyu Wang, and Peixin Zhang. 2019. Adversarial sample detection for deep neural network through model mutation testing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 1245–1256.

[44] Erroll Wood, Tadas Baltrušaitis, Louis-Philippe Morency, Peter Robinson, and Andreas Bulling. 2016. Learning an Appearance-Based Gaze Estimator from One Million Synthesised Images. In *Proceedings of the Ninth Biennial ACM Symposium on Eye Tracking Research & Applications*. 131–138.

[45] 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

[46] Shin Yoo and Mark Harman. 2007. Pareto Efficient Multi-objective Test Case Selection. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis* (London, United Kingdom) *(ISSTA '07)*. ACM, New York, NY, USA, 140–150. https://doi.org/10.1145/1273463.1273483

[47] Shin Yoo and Mark Harman. 2010. Using hybrid algorithm for Pareto efficient multi-objective test suite minimisation. *Journal of Systems and Software* 83, 4 (2010), 689 – 701. https://doi.org/10.1016/j.jss.2009.11.706

[48] 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

[49] 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*. 132–142.

[50] Xucong Zhang, Yusuke Sugano, Mario Fritz, and Andreas Bulling. 2015. Appearance-based gaze estimation in the wild. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4511–4520.

[51] 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.

[52] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. 2018. An Empirical Study on TensorFlow Program Bugs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) *(ISSTA 2018)*. ACM, New York, NY, USA, 129–140. https://doi.org/10.1145/3213846.3213866