DeepCrime: Mutation Testing of Deep Learning Systems Based on Real Faults

Nargiz Humbatova
nargiz.humbatova@usi.ch
Università della Svizzera italiana (USI)
Lugano, Switzerland

Gunel Jahangirova
gunel.jahangirova@usi.ch
Università della Svizzera italiana (USI)
Lugano, Switzerland

Paolo Tonella
paolo.tonella@usi.ch
Università della Svizzera italiana (USI)
Lugano, Switzerland

ABSTRACT

Deep Learning (DL) solutions are increasingly adopted, but how to test them remains a major open research problem. Existing and new testing techniques have been proposed for and adapted to DL systems, including mutation testing. However, no approach has investigated the possibility to simulate the effects of *real* DL faults by means of mutation operators.

We have defined 35 DL mutation operators relying on 3 empirical studies about real faults in DL systems. We followed a systematic process to extract the mutation operators from the existing fault taxonomies, with a formal phase of conflict resolution in case of disagreement. We have implemented 24 of these DL mutation operators into DeepCrime, the first source-level pre-training mutation tool based on real DL faults. We have assessed our mutation operators to understand their characteristics: whether they produce interesting, i.e., killable but not trivial, mutations. Then, we have compared the sensitivity of our tool to the changes in the quality of test data with that of DeepMutation++, an existing post-training DL mutation tool.

CCS CONCEPTS

• **Software and its engineering → Software verification and validation**.

KEYWORDS

deep learning, mutation testing, real faults

ACM Reference Format:
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 (ISSTA '21), July 11–17, 2021, Virtual, Denmark.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3460319.3464825

1 INTRODUCTION

The recent success of Deep Learning (DL) in performing complex, human-competitive tasks, such as artificial vision, speech recognition and natural language processing, are making DL based components an integral part of advanced software systems. When such systems involve life, business or ethics critical activities, the quality of the DL components they use becomes a major concern.

Traditional test adequacy criteria (e.g., branch coverage) are not effective with DL models, whose behaviour is determined by factors such as the training data, the model architecture, the training hyper-parameters, and only marginally by the source code, which consists typically of a plain sequence of invocations to the APIs of the DL framework in use. Hence, researchers have proposed several novel test adequacy criteria tailored for DL systems. These include neuron coverage [28, 31], surprise adequacy [24] and mutation adequacy [29].

Mutation testing relies on the assumption that test suites achieving a high mutation score are also very likely to be able to expose the faults that affect the original, un-mutated program, while test suites achieving a low mutation score need to be improved. This assumption is more likely to hold if mutation operators being used are based on real faults, rather than on arbitrary changes to the program under test. Moreover, mutation testing is being applied to various tasks for DL systems such as program repair [34], generation of optimal oracles for autonomous vehicles [17], detection of adversarial inputs [35], generation of adversarial code snippets for deep neural networks of source code embedding [32], prioritisation of test inputs for the labelling [36]. Availability of a mutation tool that can inject changes imitating real faults would be extremely useful also for these approaches. There exists a number of DL specific mutation operators proposed in the literature [29, 33] and 8 of them are implemented in the tool DeepMutation++ [18], which manipulates a pre-trained model to produce its mutant versions. However, none of the existing DL mutation operators, including those in DeepMutation++, are based on real faults that affect DL systems.

We have analysed 3 classifications [20, 21, 40] of real DL faults, as well as the associated replication packages, to extract a set of mutation operators based on real faults. We have followed a systematic procedure, in which different types of artefacts (e.g., StackOverflow discussions, GitHub issues and interviews) have been inspected by the two assessors, who agreed on the final list of extracted operators via consensus meetings. 24 of the 35 resulting operators are implemented in DeepCrime, the first open source DL mutation tool based on real faults, which operates at the Python-code level.

We have evaluated the properties of DeepCrime's mutation operators in an empirical study on 5 diverse DL systems. One goal of the experimentation was to understand if our operators enjoy some desirable properties, such as being killable and non trivial. We also investigated whether some mutation operators dominate some of the others, because of a subsumption relation between them. The dominating operators are of course those that deserve the highest

ISSTA '21, July 11–17, 2021, Virtual, Denmark

ACM ISBN 978-1-4503-8459-9/21/07…$15.00
https://doi.org/10.1145/3460319.3464825

priority when conducting DL mutation testing in time constrained conditions. Another goal was to assess the discriminative power of our operators in comparison to those available in DeepMutation++. In fact, the core requirement for a test adequacy criterion is that it reacts to a quality change of the test set, by recognising those test sets that are weak and need to be improved. Results confirm that DeepCrime produces mostly killable and non trivial mutants, that it can discriminate very effectively a weak from a strong test set and that it significantly outperforms the DeepMutation++ tool on these grounds. Our main contributions to the state of the art in DL testing are:

- A set of 35 DL mutation operators systematically extracted from 3 existing studies dedicated to DL faults;
- DeepCrime, the first DL mutation tool based on real faults, available as open source code;
- An empirical assessment of the properties of DeepCrime's operators and of their discriminative power, in comparison with DeepMutation++.

2 DEFINITIONS

2.1 Mutation Killing

The output $y = N(x)$ of a neural network N for an input x can be viewed as a particular instance of a random variable Y that represents the distribution of output values obtained from N with input x when taking into account the randomness associated with the training process (e.g., the randomness of initial weights, the possible randomness of the optimisers used to learn the weights from the training set, the randomness of network layers such as the dropout layer, etc.). Techniques that apply Bayesian inference to neural networks [15] aim precisely at estimating the probability distribution of Y, rather than just computing a single realisation of y.

In traditional mutation analysis, a test input kills a mutant if it gets a different output when executed against the original program and against the mutant. In DL mutation analysis, a mutant M of a neural network N could produce an output $M(x) \neq N(x)$ just because of the randomness of the training process, not because the mutant M is actually discriminated from N by the input x. In fact, even the same neural network N could produce different outputs if trained multiple times ($N_1(x) \neq N_2(x)$). Hence, as proposed originally by Jahangirova & Tonella [22], the notion of mutation killing must necessarily be defined using a statistical comparison between the distribution of the random variable Y_N that represents the output of the original network N and the distribution of Y_M that represents the output of the mutant M.

The statistical notion of DL mutation killing [22] requires that the training process is repeated n times for both the original network $N = \langle N_1, ..., N_n \rangle$ and its mutation $M = \langle M_1, ..., M_n \rangle$. The mutation is considered *killed* if for a given test set *TestS* the difference between the accuracies (or other output/quality metrics) of the original and mutated models, respectively $A_N(TestS) = \langle A_{N_1}, \ldots, A_{N_n} \rangle$ and $A_M(TestS) = \langle A_{M_1}, \ldots, A_{M_n} \rangle$, is statistically significant with non-negligible and non-small effect size. The predicate *isKilled* is defined as:

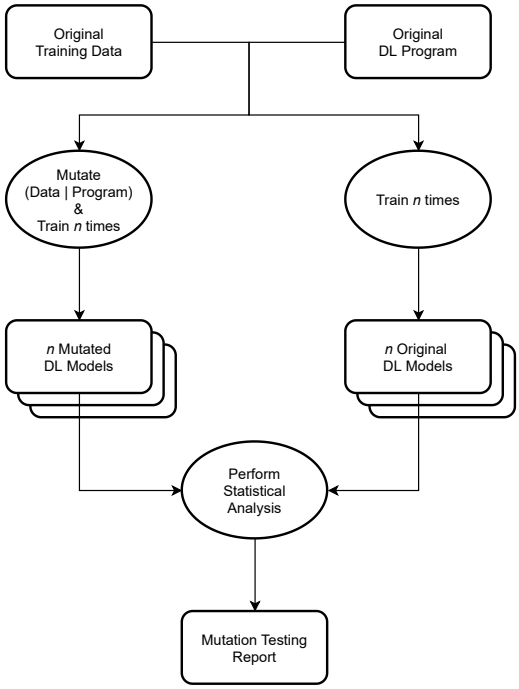

Figure 1: Mutation Testing of DL Systems

$$isKilled(N, M, TestS) = \begin{cases} true & \text{if } effectSize(A_N(TestS), A_M(TestS)) \geq \beta \\ & \text{and } p\_value(A_N(TestS), A_M(TestS)) < \alpha \\ false & \text{otherwise} \end{cases}$$

We have used generalised linear model (GLM) [30] for the calculation of statistical significance (with $\alpha = 0.05$), Cohen's d [23] for the effect size (with $\beta = 0.5$) and $n = 20$ training runs of both original and mutated models in our experiments. The Figure 1 shows the workflow of the application of the mutation testing technique to a DL system.

2.2 Mutation Score

Let $MO(p_1, \ldots, p_k)$ be a mutation operator (e.g., "delete training data") with k parameters (e.g., "delete percentage") the values of which belong to a configuration space $C = C_1 \times \ldots \times C_k$ (e.g., 0% to 99% for "delete percentage"). When applied to a subject study with concrete values assigned to its parameters, an instance of the mutation operator for the chosen parameter configuration is produced (e.g., "delete 50% of training data"). Let *TrainS* denote the training set that is used to train a subject system n times. Let $K(MO, TestS) \subseteq C$ and $K(MO, TrainS) \subseteq C$ denote the subspaces of C containing the configurations of MO that are killed by *TestS* and *TrainS*, respectively.

Definition 2.1 (Mutation Score). *Given a training set TrainS, the mutation score $MS(MO, TestS)$ of a test set TestS for mutation operator MO is defined as the proportion of mutation operator instances that are killed by both training and test set over all those that are*

killed by the train set:

$$MS(MO, TestS) = \frac{|K(MO, TestS) \cap K(MO, TrainS))|}{|K(MO, TrainS)|}$$

For example, for the mutation operator "delete training data" the training set might be able to kill all configurations where the delete percentage is greater than 5%, while the smallest delete percentage might be 25% for the test set. The mutation score will be computed as: $MS = |[0.25 : 0.99]|/|[0.05 : 0.99]| = 0.74 / 0.94 = 0.79$. The overall mutation score of *TestS* is computed as the average of mutation scores across all operators.

We use the training data as a reference for the mutation killing capability of a given test set because the training data contains the set of inputs to which the model is mostly sensitive, since the model was trained on them. A test set *TestS* that kills the same configurations killed by the training set *TrainS* is as sensitive to mutations as the training data and it achieves a mutation score equal to 1.

2.3 Properties of Mutation Operators

We aim for mutation operators that are killable (i.e., likely non-equivalent) and non-trivial (i.e., not killable by any arbitrary input).

Definition 2.2 (Killable Mutation Operator). *We consider a mutation operator killable if there exists at least one configuration that is killed by the training data (i.e., by the data that is expected to have the highest killing capability).*

To formalise the notion of *trivial* mutation operator, we first introduce the concept of a contributing input for a trained model and mutant pair.

Definition 2.3 (Contributing Input for Trained Model and Mutant). *Given a trained model N_i and a trained mutant M_i, an input x with the correct value of prediction y, contributes to killing the mutant M_i if the original trained model N_i produces a correct output $N_i(x)$ for the input x, while the mutated model produces a different output: $N_i(x) = y$ and $M_i(x) \neq y$ if the output is categorical or $\|N_i(x) - y\| < \delta$ and $\|M_i(x) - y\| >= \delta$ if the output is continuous, where δ is a predefined threshold.*

Across the n trainings of the original model N and its mutant M, an input x might be a contributing one only for some, but not for all pairs of the original and mutated models. Hence, we define the contributing input property as a statistical one:

Definition 2.4 (Killing Probability). *Given a model N and a mutant M, the probability p_K that an input x contributes to killing the mutant is the probability that random variables $Y_N = N(x)$ and $Y_M = M(x)$ differ: $p_K = P(Y_N \neq Y_M)$.*

The killing probability p_K can be estimated as:

$$p_K(x) = \frac{|\{i \in [1:n] : diff(M_i(x), N_i(x))\}|}{n},$$

where $diff(M_i(x), N_i(x))$ means that the input x is a contributing one according to the Definition 2.3.

Since the value $p_K(x)$ is estimated from a sample of size n, it is affected by an estimation error that can be measured by Wilson's method [37]. For a binomial variable (in our case, deciding whether the mutant is killed or not), Wilson's method reports its confidence interval, i.e., an interval $[l(p_K(x)) : h(p_K(x))]$ that contains the true value of p_K with some probability (which we set to 0.9). When comparing two killing probabilities p_{K1} and p_{K2} for a given input x, we can reliably say that $p_{K1}(x) < p_{K2}(x)$ (or $p_{K1}(x) > p_{K2}(x)$) only if the associated Wilson's confidence intervals $[l_1 : h_1]$, $[l_2 : h_2]$ are disjoint and $h_1 < l_2$ (or $l_1 > h_2$, to state that $p_{K1}(x) > p_{K2}(x)$).

As inputs from a test set are not just killing or non-killing inputs, but rather have a degree (probability) of contribution to killing, instead of the classical set theory we apply fuzzy set theory [39] to our analysis.

Definition 2.5 (Fuzzy Killing Set). *Given a set of inputs X and a mutation operator MO, the fuzzy killing set of the latter is defined as the fuzzy set $\langle X, \mu \rangle$, whose membership function μ consists of the contributing input probability of each input: $\mu(x).p = p_K(x), \forall x \in X$, as well as the bounds of its confidence interval $\mu(x).l = l(p_K(x)), \mu(x).h = h(p_K(x))$.*

Definition 2.6 (Trivial Mutant). *A mutant is considered trivial if it has a high **triviality score**, measured as the expected number of contributing inputs over the total number of inputs in the training set: $\mathbb{E}[|\langle TrainS, \mu \rangle|]/|TrainS| \geq \gamma$.*

In our experiments we used $\gamma = 0.90$. The expected number of contributing inputs is computed as $\sum_{x \in TrainS} \mu(x).p$.

2.4 Redundancy between Mutants

In traditional mutation analysis, a mutant M_1 is said to *subsume* (or to *dominate*) another mutant M_2 (which is hence *redundant*) if the inputs that kill M_1 are also capable of killing M_2: $K(M_1) \subseteq K(M_2)$, where K gives the killing inputs [25]. In fact, the dominating mutant M_1 is enough to select inputs that kill the subsumed mutant M_2. The problem of redundant mutants is well-known in mutation testing for traditional systems [16, 25, 26]. Such mutants inflate the mutation score and make it hard to interpret as they do not contribute to the selection of the killing inputs. Moreover, they increase the overall execution time of mutation analysis.

For DL systems, the killing power of a test input is expressed as a probability $p_K(x)$ associated with a confidence interval $[l:h]$. We therefore adapt the existing definition of redundancy to DL based on the fuzzy killing set of mutation operators.

Definition 2.7 (Subsuming Mutant). *Given a set of inputs X, a mutant M_1 subsumes another mutant M_2 if the fuzzy subset condition holds between their respective fuzzy killing sets: $p_S(\langle X, \mu_1 \rangle \subseteq \langle X, \mu_2 \rangle)$.*

The subsumption probability p_S is estimated only on non intersecting confidence intervals:

$$p_S = \frac{|\{x : \mu_1(x).h < \mu_2(x).l\}|}{|\{x : \mu_1(x).h < \mu_2(x).l \vee \mu_1(x).l > \mu_2(x).h\}|}$$

To ensure that this does not leave us with a too small set of inputs, that cannot provide conclusive results, we calculate Wilson's confidence interval of the subsumption probability p_S itself and deem our estimate of p_S as reliable only if its error rate (half the size of the confidence interval) is small enough (we set a threshold to 0.05).

3 MUTATION OPERATORS

3.1 Methodology

There are three large scale studies that have investigated real faults in DL systems [20, 21, 40]. To define DL-specific mutation operators based on real faults, two of the authors of the present paper performed a meticulous inspection of the replication packages available for these 3 publications.

We have started our analysis from the work by Humbatova and Jahangirova *et al.* [20] as it presents the most comprehensive classification available in the literature. We have studied all of the 92 unique fault types presented in this taxonomy through the analysis of the corresponding issues, available in the replication package of the study [19]. We have organised the extraction of mutation operators into two stages. First, the issues contributing to the taxonomy that originated from StackOverflow (SO) and GitHub were equally divided between two of the authors (one has analysed 74 issues and another 75). They separately studied the contents of the SO posts and GitHub issues to understand the nature of the faults and filter out those that inevitably lead to a crash. We excluded faults causing crashes as the mutation operators based on them would produce mutants that are easily killed and do not provide any information about the quality of the test set. We then analysed the remaining issues and for each produced a short text description that captured the essence of a fault and proposed a mutation operator that mimics it. The cases where either of the authors was not confident about the proposed operator were discussed and resolved.

Initially, the authors agreed on the proposed mutation operators for 42 out of 49 tags (86% of cases). For 5 tags one of the authors proposed a mutation operator, while the other did not believe that the interview tag describes a fault that can be mimicked with a mutation operator. For the remaining 2 tags the proposed operators varied very slightly (for example, "*delete layer*" vs. "*delete dropout layer*"). All of these disagreements were resolved in a consensus meeting.

We then proceeded with the analysis of two remaining works to study the possibility to propose any additional mutation operators w.r.t. the ones extracted from the taxonomy by Humbatova and Jahangirova *et al.* [20].

For what concerns the dataset by Islam *et al.* [21], we carefully studied bug types, root causes and effects reported for each bug. We excluded generic programming issues, non-DL-specific issues and issues pointing to incorrect documentation of frameworks and libraries. We filtered out all of the bug reports that lead to a crash and divided the remaining issues between the two authors for a deeper analysis. As opposed to the first study, this dataset contained several general *'how to'* questions and unresolved issues. After excluding such instances, we processed the faults relevant for our purposes and introduced 5 additional mutation operators.

The dataset by Zhang *et al.* [40], also built on issues collected from GitHub and SO, is organised as a set of reproducible bugs, consisting of two versions of the same DL program: the correct and the buggy one. For the bugs that were collected from SO we analysed corresponding SO posts and related discussions. A closer inspection revealed that some of these issues contained unresolved issues and *'how to'* questions. For the bugs reproduced from GitHub commits and issues, we manually inspected differences the in code using a publicly available tool for visual comparison and merge of text files [1].

As a result, we extracted 35 unique mutation operators with 27 of them obtained from the taxonomy by Humbatova and Jahangirova *et al.* [20], 5 from the work by Islam *et al.* [21] and 3 from the study of Zhang *et al.* [40].

3.2 List of Mutation Operators

In this section we present the extracted mutation operators, which are shown in Tables 1 (implemented in DeepCrime) and 2 (not yet implemented), where they are grouped by area of application (*Group*). The ID of each operator consists of the first letter that identifies the group (e.g., T = "Training Data") followed by two letters that identify the operator (e.g., CL = "Change Labels").

We have not implemented a portion of the proposed operators because their application is tricky as they trigger a cascade of changes to neural networks' structure (e.g., operator that removes/adds a layer). The usage of such operators might be limited depending on the structure of the network under test as some of the generated mutants might be causing crashes, which is not particularly useful for the mutation testing. Our results show that already implemented operators are enough to outperform the existing, DL-specific mutation testing tools, and the rest will be part of future work. Due to the space limitations, we do not include the descriptions for non-implemented operators, but provide them in our replication package [3].

3.2.1 Training Data Operators. Training data operators manipulate the training set so as to mimic the issues possibly affecting the dataset used to train a DL system. Some operators (e.g., TCL, TUD) assume that the ground truth label y of the training data is a class. For regression problems, where the predicted value is a continuous variable, and not a class, we perform binning to partition such continuous values into a finite set of classes. In particular, by default we create three classes [22]: $y \leq -\sigma_y$; $-\sigma_y < y < \sigma_y$; $y \geq \sigma_y$, where σ_y is the standard deviation of the continuous label y.

Change Labels of Training Data. This mutation operator mimics situations when wrong labels are provided for the training data. Its parameters, *label* and *percentage*, represent the label to be replaced by another, randomly selected label, and the percentage of data with the given label to be changed. If the *label* parameter is not provided by the user, DeepCrime chooses the label appearing most frequently in the training set.

Remove Portion of Training Data. This mutation operator mimics situations where there is not enough training data available. Its parameter *percentage* indicates what percentage of the training data should be removed. DeepCrime removes parts of training data from each class in a proportional manner.

Unbalance Training Data. This mutation operators mimics the case when the data available for training is unbalanced. To achieve this, DeepCrime first calculates the average occurrence of each class in the training data. DeepCrime then identifies the classes that appear less frequently than the average and removes a percentage of their data (specified via the *percentage* parameter), such that the imbalance of the data becomes even more severe.

Make Output Classes Overlap. The mutation process starts by identifying two dominating classes in the training dataset. This

Table 1: Mutation operators implemented in DeepCrime. Column "ST" indicates the type of search used to find killable configurations (B = binary; EL = exhaustive on list; EU = exhaustive on user provided values).)

Group	Operator	ID	Mutation Parameters	ST
Training Data	Change labels of training data	TCL	*label* — a particular label to mutate *percentage* — a percentage of data for the given label to mutate	- B
	Remove portion of training data	TRD	*percentage* — a percentage of training data to delete	B
	Unbalance training data	TUD	*percentage* — a percentage of training data of underrepresented/selected labels to remove in order to unbalance the training data	B
	Add noise to training data	TAN	*percentage* — a percentage of training data to mutate	B
	Make output classes overlap	TCO	*percentage* — a percentage of training data to mutate	B
Hyperparameters	Change batch size	HBS	*new batch size* — new batch size to be used to train the system under test	EU
	Decrease learning rate	HLR	*new learning rate* — new learning rate to be used to train the system under test	B
	Change number of epochs	HNE	*new number of epochs* — new number of epochs to be used to train the system under test	B
	Disable data batching	HDB	—	-
Activation Function	Change activation function	ACH	*layer* — the number of a layer with non-linear activation function to mutate *new activation function* — new activation function for the layer under mutation	EL EL
	Remove activation function	ARM	*layer* — the number of a layer to mutate	EL
	Add activation function to layer	AAL	*layer* — the number of a layer with linear activation function to mutate *new activation function* — new activation function for the layer under mutation	EL EL
Regularisation	Add weights regularisation	RAW	*layer* — the number of a layer with no weights regularisation to mutate *new weights regularisation* — the type of weights regularisation to be added for the layer under mutation	EL EL
	Change weights regularisation	RCW	*layer* — the number of a layer with existing weights regularisation to mutate *new weights regularisation* — the type of weights regularisation to be added for the layer under mutation	EL EL
	Remove weights regularisation	RRW	*layer* — the number of a layer with existing weights regularisation to mutate	EL
	Change dropout rate	RCD	*layer* — the number of a dropout layer *new dropout rate* — new dropout rate for the layer under mutation	EL EU
	Change patience parameter	RCP	*new patience value* — new value for the patience parameter of the *EarlyStopping* callback	B
Weights	Change weights initialisation	WCI	*layer* — the number of a layer to mutate *new weights initialisation* — new type of kernel initialiser for the layer under mutation	EL EL
	Add bias to a layer	WAB	*layer* — the number of a layer with no bias to mutate	EL
	Remove bias from a layer	WRB	*layer* — the number of a layer with bias to mutate	EL
Loss function	Change loss function	LCH	*new loss function* — new loss function to be used to train the system under test	EL
Optimisation Function	Change optimisation function	OCH	*new optimisation function* — new optimisation function to be used to train the system under test	EL
	Change gradient clipping	OCG	*new gradient clipping* — new value to be used for gradients clipping	EU
Validation	Remove validation set	VRM	—	—

Table 2: Operators not yet implemented in DeepCrime

Group	Operator	ID
Training Data	Remove data augmentation	TRA
Layers	Change pooling amount	LCP
	Change the filter size of a convolutional layer	LCF
	Change the padding for a convolutional layer	LCD
	Change the stride for a convolutional layer	LCS
	Change the number of neurons in a layer	LCN
	Remove layer	LRM
	Add layer	LAD
	Change layer type	LCT
	Change output shape of a layer	LCO
	Change skip connections	LCC

operator duplicates the amount of data specified by the parameter *percentage*, taking it from the largest of the two classes, while using the second class as the label. This recreates the situation where the same or very similar training data elements have different labels assigned to them.

Add Noise to Training Data. This mutation operator introduces low quality training data by adding some noise to the original data. The mutation operator uses two parameters: *standard deviation percentage* and *percentage*. DeepCrime takes the vector representation of the training data and calculates the standard deviations of its components. It multiplies these standard deviations by the value of the parameter *standard deviation percentage* and obtains a new value for the standard deviations. Using this value and a mean of zero, DeepCrime generates (by default Gaussian) noise to be added to the training input vectors. The parameter *percentage* identifies what percentage of the inputs will be mutated.

3.2.2 Hyperparameters Operators. This group of operators simulates the choice of suboptimal values for the hyperparameters.

Decrease Learning Rate. This mutation operator investigates the consequences of a too small learning rate to train a model.

Change Number of Epochs. This operator changes the number of epochs for which a model is trained.

Change Batch Size. This operator changes the number of samples presented to a network for a single update of its weights.

Disable Data Batching. This operator mimics the setting where no mini-batching is used to train the model under test.

3.2.3 Activation Function Operators. Mutation operators from this group imitate wrong choices of activation function for specific layers in a model.

Add Activation Function. This mutation operator operates on layers with *linear* activation function and changes it to the one specified by the user. If the user does not specify any preference, the choice is random.

Remove Activation Function. This operator substitutes the activation function of a layer from non-linear one with *linear*.

Change Activation Function. This mutation operator is applicable to layers with *non-linear* activation functions. The new activation (also non-linear) can be provided by the user or is chosen randomly.

3.2.4 Regularisation Operators. The first three operators of this group manipulate the penalties imposed on layers' kernels, while the last two manipulate two regularisation hyperparameters, *dropout rate* and *patience*.

Add Weights Regularisation. This operator adds a regulariser to layers where no regularisation is used.

Remove Weights Regularisation. This operator removes the regularisation of layers where it was originally used.

Change Weights Regularisation. This mutation affects the layers on which a kernel regulariser is applied, by changing the regulariser to the one provided by the user or to another one, selected randomly.

Change Dropout Rate. This operator affects solely dropout layers by changing their dropout rate parameter.

Change Patience Parameter for Early Stopping. Early stopping is an effective way to prevent overfitting. Changing this parameter can produce models that either overfit or even underfit training data.

3.2.5 Weights Operators. Change Weights Initialisation. This mutation operator changes the original kernel initialiser, where possible, to a randomly chosen or user specified one.

Add Bias to a Layer. The bias vector contains an additional set of weights, not connected to any input. This mutation operator creates a bias vector if none is there.

Remove Bias from a Layer. On the contrary, this mutation removes the bias vector of a layer.

3.2.6 Loss Function Operators. Change Loss Function. This mutation operator changes the loss function used to train a model to the one chosen by the user or selected randomly.

3.2.7 Optimisation Operators. There exist a number of optimisation algorithms that can be used while training a DNN model and its choice can affect the performance of the model to a large extent.

Change Optimisation Function. This mutation operator changes the optimiser originally selected to train the model to a new one.

Change Gradient Clipping. Gradient clipping is a technique to avoid exploding gradients while training a DNN. It operates by clipping or normalising the gradients to a predefined value. The aim of this mutation operator is to alter this value.

3.2.8 Validation Operators. Remove Validation Set. This mutation operator disables the usage of validation data during training.

3.3 Search for a Killable Configuration

As shown in Table 1, the majority of the mutation operators provided by DeepCrime have to be supplied some parameter values. When a user does not provide any specific value for these parameters, instead of letting DeepCrime choose them randomly, it is possible to activate an automated search for the killable configurations of the mutation operator. The goal of the search procedure is to find a configuration that is killable but not trivial to kill.

The type of search algorithm used to find a killable configuration is determined by the domain of the mutation operator's parameters, which we classify into three groups: list-based, range-based and user-specified. A parameter is *list-based* if its validity domain is limited to a predefined list of values, such as the list of the layers in the model under test or the list of loss/optimisation/activation functions available in the DL framework (e.g., Keras). For the *range-based* parameters, the validity domain is a continuous range of values on which the mutation impact is increasingly higher or lower depending on the operator. For example, the *percentage* parameter of TRD can take a value in the range (0, 100), and the higher value we pick, the stronger is the effect of the mutation. *User-specified* parameters are the ones that do not fall into the previous two categories and their values need to be specified in an ad-hoc manner. For example, the *batch size* takes values in the range [1, length of training data], but it does not satisfy the condition of an increasing impact of the mutation throughout the range.

For the range-based parameters, DeepCrime uses binary search to find a killable parameter configuration that is likely to be non trivial. DeepCrime first checks if the operator is killable in the most aggressive configuration, it then finds the middle point of the range and checks the killability at this value of the parameter. If so, it reapplies the search to the first half of the range. If not, it moves to the second half instead. This process repeats recursively until the size of the range becomes smaller than or equal to a predefined *precision* ϵ, which gives the granularity of the smallest change percentage to be performed. For the list-based and user-specified parameters, DeepCrime implements an *exhaustive search* for the killable configurations. Column "*ST*" (Search Type) in Table 1 indicates which search type has been used for each mutation operator.

4 IMPLEMENTATION

DeepCrime was developed in Python 3.8 and is applicable to sequential and functional models implemented with Keras. The tool is publicly available [14].

4.1 Injecting Mutations into Python Code

We use Python's Abstract Syntax Tree (AST) module to parse and seed faults into the code that builds and trains the DL model under test. Specifically, while traversing the nodes of the parse tree, DeepCrime identifies the places in the code (target nodes) where each mutation can be injected. For example, DeepCrime looks for calls to specific model training APIs, such as `fit` or `compile` when applying the TRD mutation operator. By parsing such nodes' arguments and keywords, DeepCrime can extract the names or the values of variables of interest, such as the constructed model, the training data (to be manipulated by TRD) or various hyperparameters. Then, the tool can proceed and seed the mutation by modifying the target nodes' parameters or by inserting new nodes representing calls to functions provided by DeepCrime and implementing specific mutation operators. For example, in order to change the number of epochs (HNE operator), DeepCrime just replaces the old value with the new one for the corresponding argument of the `fit` call. When a mutation operator manipulates the layers of a model (e.g., ACH operator), it must be applied to the model before it is compiled. The corresponding mutation function must be called with the model as a parameter just before the invocation to `compile`, so as to return a new model containing the mutated layers, which can now be compiled. After all the necessary modifications to inject the mutations are performed, DeepCrime unparses the modified parse tree back to Python code, hence finalising the creation of a mutant.

4.2 Configuration of DeepCrime

We constructed DeepCrime in a way that it provides its users with high flexibility in configuring each mutation operator, while also minimising the amount of manual work required for a basic usage.

DeepCrime imposes a couple of requirements on the source code of the tested system: (1) the evaluation of the DL model should be enclosed in a method called *main()*, (2) this method should return

the evaluation score. If these requirements are not met by the original code, it is usually quite straightforward to change the code and make it compliant with them.

As the majority of mutation operators come with a set of configurable parameters, we provide users with the functionality to supply specific values for such parameters. An example would be an explicit specification of the percentage of training data to be deleted by the TRD operator or a specific optimisation function to be used by the OCH operator. In case no preferences are specified, DeepCrime would apply a search method appropriate for the parameter in question. Another user option is the selection of a random configuration of the mutation operator.

To determine if a given mutant instance is killed, DeepCrime uses the *isKilled* predicate introduced in Section 2, which requires n retraining of the original model and of the mutant. By default, n is set to 20, but users can change this value.

As output, DeepCrime produces a number of CSV reports, one per applied mutation operator. For each generated mutant, the report contains a record that captures whether the mutation was killed and also reports the *p-value* and *effect size* obtained from the statistical analysis. Additionally, there is a report that includes the mutation score for each of the applied operators and the total mutation score across all the operators.

5 EXPERIMENTAL RESULTS

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

RQ1 [Interesting Mutation Operators]: *What are the interesting (killable, non trivial) mutation operators?*

To investigate this research question we evaluate DeepCrime's mutation operators on the experimental subjects and identify which of them have at least one killable configuration. We then determine the contributing inputs for the killed configurations, and measure the killability score and the average triviality score of the operator. The *Killability Score* (KS) is defined as the ratio of configurations killed by the training set. This is the denominator of the mutation score MS (see Definition 2.1), divided by the size of all configurations: $KS = |K(TrainS)|/|C|$. The *Average Triviality Score* (ATS) is defined as the triviality score (measured according to Definition 2.6) averaged across all configurations sampled during binary or exhaustive search.

For the training operators with continuous range parameters, we performed binary search in the widest range available and set the value of the *precision* parameter to 0.05. For TRD, the range we used was (0, 0.99], as we can not remove the whole training data, while for the remaining operators it was (0, 1.00]. For HNE and RCP, the lower bound of search was always 1 and the upper bound was the number of epochs or the patience in the DL model, respectively. For HLR, the upper bound was the learning rate in the DL model and the lower bound was set to a number very close to 0. For these last three operators we set the *precision* as the size of range divided by 10. For the mutation operators with list-based parameters, we performed an exhaustive search. However, for the operators that have two parameters of this kind and one of them identifies a *layer*, we picked the *layer* randomly and performed the exhaustive search on the other parameters. For example, for the operator ACH we changed the activation function to all the ones available in Keras for one layer picked randomly. For HBS we identified the *batch_size* in the DL model and provided a list of 4 pre-defined values to DeepCrime: $batch\_size \div 4$, $batch\_size \div 2$, $batch\_size \times 2$, $batch\_size \times 4$. For RCD, we again picked the layer randomly and provided 4 pre-defined values for the dropout rate that are different from the one used in the DL model under test and are evenly distributed in the range of (0, 1].

RQ2 [Redundant Mutation Operators]: *What are the mutation operators producing redundant mutants?*

We further evaluate the relative effectiveness of the operators by analysing the results produced by mutations in relation to each other. Before proceeding with redundancy analysis we exclude the non killable mutants, which are likely to be equivalent to the original model.

For all pairs of killable mutants we identify whether they satisfy Definition 2.7. We capture the subsumption relationship in a graph [25]: if a mutant M_1 subsumes a mutant M_2, we add an edge from the node of M_1 to the node of M_2. The mutants, the nodes of which have no incoming edges, are non-redundant (or dominant), while the remaining mutants are redundant.

RQ3 [Comparison with Post-training Mutation]: *How do DeepCrime's pre-training mutation operators discriminate between different qualities of test data in comparison with the post-training operators of the existing tool DeepMutation++?*

In order to assess the performance of the pre-training mutation operators proposed and implemented in this paper, we compare DeepCrime to DeepMutation++ [18], a model-level, post-training mutation testing tool that implements 8 operators from the ones introduced by Ma *et al.* [29] and 9 new operators designed for stateful recurrent neural networks. We measure the *sensitivity* of DeepCrime and DeepMutation++, defined as the relative variation of the mutation score when moving from a weak to a strong test set: $Sens = |MS(Strong) - MS(Weak)|/MS(Strong)$. This metric determines which of the two tools is more sensitive when reacting to a change in the test set quality.

As opposed to DeepCrime, the mutation operators of DeepMutation++ change the weights or the structure of an already trained network. The operators of DeepMutation++ are not based on real faults and implement post-training changes to a model that are unlikely to happen in a real-world setting. DeepMutation++ was implemented in older versions of Python and is based on Keras 2.2.4 with TensorFlow 1.13. To be able to apply this tool to the recently developed case studies used in this work, we had to upgrade it to be compatible with Python 3.8, Keras 2.4.3 and TensorFlow 2.3.0. We also extended the applicability of the tool from *Sequential* Keras models to *Functional* ones, by re-implementing the *Layer Remove* and *Layer Duplication* operators.

To answer RQ3 we need two test sets: a strong and a weak one. For the former, we use just the test data provided with the experimental subjects, while the latter is constructed artificially. In case of classification systems, the weak test set is obtained by removing the test inputs on which the model under test has least confidence. In fact, low confidence inputs are those that are closer to the boundaries between classes and are useful to test a model in corner cases that the model finds difficult to handle. By removing such corner cases we expect to get a less discriminative test set,

which is less effective in assessing the quality of the model under test. This approach has previously been used for a similar task by Jahangirova & Tonella [22] and for test input prioritisation by Byun et al. [13]. In our experiment, for classification systems we build a weak test set by keeping only the test inputs that are predicted with a confidence equal to 1, where confidence is measured as the highest softmax output value.

With regression systems, the most discriminative inputs are those with minimal mean loss and minimal standard deviation of loss in multiple evaluations, since detrimental changes in the model are expected to be reflected first of all on the lowest error values and are not caught by the inputs whose loss is already high. Thus, we constructed the weak test sets for regression systems by removing the inputs with low mean loss or low standard deviation of loss observed on 20 instances of the original models.

5.1 Datasets and Subject DL Systems

For the evaluation of mutation operators in DeepCrime we used 5 different subject systems. We selected our subjects so that they represent different types of DL systems (classification vs. regression), with various DL model structure, training data representation, application domain and diverse tasks that they are trying to achieve.

MNIST (MN) [27] is a large dataset of handwritten digits that is widely used in computer vision and deep learning. We adopted a convolutional model with 8 layers [6] to automatically classify the images into 10 digit labels.

Speaker Recognition (SR) dataset from Kaggle [9] contains 1,500 audio files for 5 different prominent leaders. The dataset also contains background noise audio files such as audience laughing or clapping. The subject model we have used for this dataset [10] recognizes the speakers from the frequency domain representation of speech recordings.

Movie Recommender (MR) is a DL model based on collaborative filtering [7] that uses the MovieLens ratings dataset [8] to recommend movies to users. The MovieLens dataset contains 100,836 ratings given by 610 different users to 9,742 movies. The task of the model is to predict ratings for the movies that a user has not watched yet and to recommend the movies with the highest predicted ratings.

Udacity (UD) self-driving car aims to predict the steering angle for the road that the car encounters while driving in a simulation environment. The model needs a dataset of road images labelled with a steering angles to learn the lane-keeping functionality. We used training data available from a previous work [17] and from NVIDIA's Dave-2 model [12] for this case study.

UnityEyes (UE) is a publicly available rendering framework [38] that allows synthesis of large amounts of various eye region images. For this subject, we use a labelled dataset [4] consisting of images synthesised with UE. The model that comes with the dataset learns the mapping from eye image and 2D head angle to 2D eye gaze angle (yaw and pitch).

Table 3 lists details about our subject systems. Column *Metric* indicates the metric that was used to measure the performance of the model; Column *Value* provides the mean value of that metric, when measured on the test set after completing the training process. The DL models for the first three subjects were obtained from the official Keras documentation [5], the fourth subject was obtained from the existing literature [12, 17], while the fifth subject was available in a public GitHub repository [4].

Table 3: Subject DL Systems; MSE = Mean Squared Error

ID	Train Data	Test Data	Epochs	Metric	Value
MN	60,000	10,000	12	Accuracy	99.03%
SR	5,401	1,350	100	Accuracy	98.29%
MR	72,601	18,151	5	MSE	0.047
UD	9,792	2,432	50	MSE	0.014
UE	103,428	25,857	50	Angle between gaze vectors	3°

For our 3 regression subject systems we need to define thresholds, so that if the difference between actual and correct predictions is less than this value, we say that an input is predicted correctly. For MR, we say that the rating prediction is accurate if it differs from the correct prediction by no more than one rating. For UE, we selected a threshold of 5 degrees as it is a step of a change of yaw and pitch angles that was used to create the corresponding dataset. For UD, there was no frame of reference from the dataset itself and therefore we picked the value of 0.3 empirically based on the predictions of the original model.

5.2 Results

5.2.1 RQ1 (Interesting Mutation Operators). Out of 24 operators currently implemented in DeepCrime, 20 were applicable to the subject systems that we analysed (MN: 18; SR: 16; MR: 11; UE: 17; UD:15). Four operators were not applicable, as none of our subjects possess some specific, required property (e.g., weights regularisation in a layer or gradient clipping value for an optimiser). The number of mutation operator configurations ranges between 76 and 217 (MN: 217; SR: 119; MR: 76; UE: 157; UD:101). As we performed 20 runs for each configuration, in total we performed 13400 (670 x 20) re-trainings during our experiments.

Column *KS* in Table 4 reports the killability score. If this value is equal to zero, then this operator is not killable for any of the applied parameter values. There are 10 mutation operators that are killable for all subject systems. Two operators are not killable for any of the subject systems: VRM and RCD.

Once the list of killable operators was identified, we proceeded with triviality analysis for them. Column *ATS* in Table 4 reports the average triviality score for each mutation operator. LCH has the highest triviality score among all other mutation operators for 4 out of 5 subjects. When it comes to the triviality of each mutant instance, for MN, SR, MR there are no trivial configurations (i.e, none with triviality score > 0.9). In contrast, UE has 4 (all LCH) and UD has 9 (8 LCH, 1 OCH) trivial mutant configurations.

RQ1: The mutation operators implemented in DeepCrime generate a large number of killable, non-trivial mutants. VRM and RCD are non killable for all subjects; LCH has the highest triviality score in 4/5 subjects.

Table 4: Killability and triviality of mutation operators; '-' means not applicable to the subject

Op	MN		SR		MR		UE		UD	
	KS	ATS	KS	ATS	KS	ATS	KS	ATS	KS	ATS
TCL	97%	0.034	97%	0.059	94%	0.119	88%	0.111	81%	0.061
TRD	97%	0.036	94%	0.034	94%	0.127	75%	0.131	87%	0.072
TUD	91%	0.009	0%	-	38%	0.103	1%	0.092	0%	-
TAN	56%	0.002	-	-	-	-	50%	0.082	-	-
TCO	75%	0.016	97%	0.078	94%	0.108	0%	-	97%	0.133
HBS	75%	0.002	-	-	75%	0.149	50%	0.077	-	-
HLR	81%	0.011	0%	-	69%	0.100	50%	0.076	56%	0.098
HNE	82%	0.005	51%	0.029	75%	0.108	76%	0.137	88%	0.108
HDB	100%	0.003	-	-	100%	0.261	-	-	-	-
ACH	56%	0.005	11%	0.005	-	-	56%	0.137	0%	-
ARM	50%	0.223	0%	-	-	-	33%	0.124	0%	-
AAL	-	-	0%	-	-	-	70%	0.380	-	-
RAW	100%	0.003	0%	-	-	-	100%	0.218	0%	-
RCD	0%	-	-	-	-	-	-	-	0%	-
RCP	-	-	78%	0.016	-	-	-	-	-	-
WCI	31%	0.191	8%	0.005	-	-	58%	0.225	0%	-
WRB	0%	-	33%	0.005	-	-	0%	-	0%	-
LCH	77%	0.010	100%	0.794	50%	0.261	82%	0.485	75%	0.654
OCH	50%	0.006	67%	0.018	100%	0.187	33%	0.171	33%	0.345
VRM	0%	-	0%	-	0%	-	0%	-	0%	-

5.2.2 RQ2 (Redundant Mutation Operators). For each subject, we excluded the non killable mutants. Table 5 provides information about the overall number of killable configurations (Column *Killable Confs.*), the number of redundant (Column *Redundant*) and the number of non-redundant mutants (Column *Non Redundant*) for each of the subject systems. As we can see from the results, the number of redundant mutants is quite high (>=50%) for MN and SR, both of which are classification systems. Out of 10 mutation operators that are killable for SR, six (all except ACH, WCI, OCH, WRB) have configurations that lead to redundant mutants. For MN this number is even higher, with 14 out of 15 (all except TCO) killable operators producing redundant mutants. A closer analysis of the parameter values that lead to redundancy reveals that across all operators this is often caused by the usage of extreme values of the parameters. For example, for TRD such a parameter value is removing 99% of the training data and for HNE it is changing the number of epochs from the original number to a value as low as 1. This trend is easily explainable, as the mutants applied with extreme parameter values are easier to kill and therefore their fuzzy killing sets are supersets of the harder to kill mutants.

Table 5: Redundancy Analysis

ID	Killable Confs.	Redundant	Non Redundant
MN	75	38	37
SR	56	28	28
MR	43	4	39
UD	40	37	3
UE	66	6	60

In contrast to SR and MN, the number of redundant mutants is low for MR and UE, both of which are regression systems. More specifically, for MR all 4 and for UE 4 out of 6 redundant mutants are instances of LCH operator. The remaining two redundant mutants of UE are instances of WCI operator. The results for UD are quite peculiar, as only 3 out of its 40 killable configurations are non-redundant. A closer investigation on this case shows that there is a mutant generated by applying the TCL operator with parameter value equal to 18.75%, for which the inputs have very low killability probabilities (hence, this mutant is very hard to kill). As a result, almost all the other mutants are redundant with respect to this one. If we remove this mutant from the subsumption graph, the results become very similar to the ones of MR and UE, i.e. UD gets to have only 7 redundant mutants, 6 of which are instances of LCH operator and 1 is instance of OCH operator.

Overall, we can see that the results of redundancy analysis are closely linked to the results of triviality analysis. LCH operator has the highest triviality score across all subjects, i.e. it produces easy to kill mutants, and as a result these mutants are also redundant across all subjects.

> **RQ2**: The only operator that produces redundant mutants for 5/5 subjects is LCH. Moreover, mutation operators applied with extreme parameter values lead to the generation of redundant mutants, which further justifies the use of binary search to find hard to kill mutant configurations.

5.2.3 RQ3 (Comparison with Post-training Mutation). For the two classification systems MN and SR, we extracted the weak test sets. For MN, the size of the original, strong test set is 10,000; that of the weak test set is 4,813. For SR, the original test set size is 1,350, while the weak, reduced test set has size of 686. MR could not be used for RQ3 because DeepMutation++ is not applicable to this system.

For the regressions systems UD and UE, we considered the inputs with the highest standard deviation of loss observed across evaluations of 20 instances of the original model. For UD, the size of the original test set is 2,432 and the size of the artificially constructed weaker set is 1,000 inputs. For UE, the respective sizes are 25,857 and 4,000. We validated our artificially constructed weak test sets by measuring their adequacy level according to the k-multisection neuron coverage criterion [28] and to surprise adequacy [24]. The resulting adequacy scores are consistently lower for the weaker test sets than for the stronger ones.

Before calculating the mutation score for the strong and weak test suites using DeepCrime, we performed power analysis and excluded all mutation operators whose output data (accuracy, regression prediction) have too low statistical power to decide reliably if the mutant is killed or not (we adopted the conventional threshold $\beta \geq 0.8$, where β is statistical power). This led us to the exclusion of 8 mutation operators on MN, 2 on SR, 4 on UE, and 4 on UD.

The authors of DeepMutation++ define the mutation score [29] for an input t as the ratio of the number of killed mutants m' by t to the total number of mutants m: $MS = m'/m$. We then average this value across all inputs in the test suite, to get a single mutation score value for the test suite. When running DeepMutation++, we used default values of its parameters, when available. We set the mutation ratio parameter (the one that controls the "aggressiveness" of the mutation) to 0.05, which appears to be the best performing parameter configuration among those reported in DeepMutation++'s tool paper [18]. To get relatively stable mutation scores, we generated 400 mutants for each mutation operator that was applicable to a subject study and verified that the standard error of the mean mutation score was always lower than 5%, for each of the evaluated test sets.

Table 6: Mutation scores of weak/strong test sets; Sensitivity

Subject	DeepMutation++			DeepCrime		
	Weak	Strong	Sens	Weak	Strong	Sens
MN	0.0507	0.0591	14.21%	0.1186	1.0000	88.14%
SR	0.0618	0.0846	26.95%	0.4675	0.8500	45.00%
UD	0.1914	0.1976	3.14%	0.0000	0.7700	100.00%
UE	0.2768	0.3341	17.15%	0.6220	0.9250	32.76%

With respect to DeepMutation++, the results provided in Table 6 show that DeepCrime can better discriminate the quality of the test data for all 4 subjects. In particular, on MN, DeepCrime achieves 88.14% of sensitivity, while DeepMutation++'s sensitivity is 14.21%. On SR, DeepCrime outperforms DeepMutation++ by 18.05%, with a sensitivity to the change in the test suite quality equal to 45%. Similarly, DeepCrime outperforms the DeepMutation++ on both regression subjects, namely, UD with a difference of 96.86% and UE with a difference of 15.61%.

There could be a number of reasons behind the higher sensitivity of DeepCrime. The first one is that DeepCrime's operators are based on real faults that are found to affect the performance of DL systems, as observed by practitioners. Another one is that our mutation operators are pre-training, which means that the changes are imposed to a system before the training process commences and are potentially affecting it. Post-training operators, while being much cheaper w.r.t. time and resources, are mostly modifying a randomly chosen, small portion of weights of an already trained network.

> **RQ3**: The pre-training mutation operators based on real faults that are implemented in DeepCrime have substantially higher sensitivity to changes of the test set quality than the post-training mutation operators implemented in DeepMutation++.

5.3 Threats to Validity

Construct. One threat to the construct validity is our own definition of mutation score. While we defined MS based on the statistical comparison of performance metrics proposed by Jahangirova and Tonella [22], we had to take into account the configuration space of each mutation operator and find a way to determine the killed subspace. The search used to measure the volume of the killed subspace (we used both binary and exhaustive search, depending on the mutation parameter) might have affected the accuracy of the reported MS value.

Internal. We propose our mutation operators based on real faults reported in three large scale existing studies on DL bugs. However, there is no up-to-date exhaustive list that would cover all possible instances of DL-related faults, especially because this domain is rapidly evolving.

External. We evaluated DeepCrime's mutation operators on 5 case studies and therefore we cannot guarantee generalisation to other subjects. We carefully selected our models to represent various tasks, architectures and areas of application. The fact that all of our subjects are implemented using Keras can pose additional limitations to the generalisation of our results. We made this choice based on the popularity of the framework [2].

6 RELATED LITERATURE

6.1 Real Faults

A number of recently published studies provide an insight into the faults specific of DL systems and thus are potential sources for the definition of DL specific mutation operators based on real faults.

Humbatova and Jahangirova *et al.* [20] define a real fault in DL systems as a case when a human-made mistake during the development or training of a DL component, leads to functionally insufficient performance. In their work, the authors build a comprehensive taxonomy of real faults in DL systems based on manual analysis of unstructured sources such as GitHub and StackOverflow (SO) and semi-structured interviews with DL practitioners. The final version of the taxonomy consists of 92 unique types of faults that are classified into 5 top-level categories with 3 of them further branching into inner subcategories. The final taxonomy was validated through a survey with a set of 21 practitioners and researchers.

Among other related publications is the work by Zhang *et al.* [40] who studied a set of applications developed using the TensorFlow framework. By manual examination of 175 generic programming and DL specific bugs collected from StackOverflow posts and 11 DL-related tutorial projects from GitHub, the authors focused on the

exploration of strategies that developers employ while localising faults in DL systems and on the associated challenges. As a result, the authors systematised the acquired information on root causes of bugs into seven broad types. They also classified the way such bugs can affect the system behaviour into four categories.

Another publication that investigated general patterns of DL bugs and their evolution over time is the work by Islam *et al.* [21]. Similarly to the previously discussed studies, the authors used Github and SO to construct their dataset of 3,216 potential bug reports. The obtained artefacts are related to the Theano, Caffe, Keras, TensorFlow, and PyTorch frameworks and include problems in the documentation of the frameworks as well as DL-specific and generic programming bugs. As building a novel fault taxonomy was not the main focus of their study, the authors adapted the existing classification schemes by Beizer [11] and Zhang *et al.* [40] to discover the patterns and classify the bugs by causes and impacts.

6.2 Mutation Testing of DL Systems

The works by Ma et al. [29] and Shen et al. [33] are the most related to ours as they also propose mutation operators specific to DL systems. The work by Ma et al. [29] was later extended into a mutation testing tool for DL systems named *DeepMutation++* [18]. However, as the authors of these works note themselves, none of the proposed mutation operators are based on real faults. Moreover, the mutation killing criteria they use do not take into account the stochastic nature of DL systems. DeepCrime is the first tool implementing a set of DL mutation operators rooted on real DL faults.

The work by Jahangirova & Tonella [22] introduced a statistical definition of mutation killing (discussed in Section 2) and performed an empirical evaluation of the mutation operators proposed by Ma et al. [29] and Shen et al. [33]. The authors investigated whether the mutation operators are killable and non trivial. For the definition of killable mutation operator they used the largest test set available and checked whether it kills the mutant. In our work, we use the training set to analyse killability, as this is the set to which the DL system is most sensitive. To filter out trivial mutations, the authors investigated whether the mutant is getting killed by a very "weak" test suite (1% of the available test set). In contrast, we used fuzzy set theory for triviality analysis and do not rely on the hypothesis that the strength of a test set depends on its size. DeepCrime is the first DL mutation tool that implements stochastic measures for key mutation analysis indicators, such as the mutation score, and for key properties, such as killability, triviality and redundancy.

7 CONCLUSION AND FUTURE WORK

We have proposed 35 and implemented 24 DL-specific mutation operators based on real faults that have been reported in 3 existing DL fault taxonomies. The empirical assessment of the implemented operators shows that most of these operators are killable and non trivial. There is only one mutation operator that produces configurations which are redundant for all subject studies, while there is not a case when all of the configurations for this operator are redundant. The evaluation and the comparison with the existing DL mutation tool DeepMutation++ showed that our operators can discriminate more effectively a weaker from a stronger test set.

In our future work, we plan to implement the remaining 11 operators that were extracted from the fault taxonomies. We intend to extend the applicability of DeepCrime beyond models written for Keras framework. We think DeepCrime will enable a number of applications to existing software engineering problems. In particular, we are interested in the possible connection between automated test input generation for DL systems and mutation adequacy. We think that DeepCrime can provide a solid framework for the empirical assessment of alternative test generators, since its mutations simulate real DL faults, which are the target of automatically generated test cases.

ACKNOWLEDGEMENTS

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

REFERENCES

[1] 2013. DiffMerge: an application to visually compare and merge files on Windows, OS X and Linux. https://sourcegear.com/diffmerge/.
[2] 2019. FrameworkData. https://towardsdatascience.com/deep-learning-framework-power-scores-2018-23607ddf297a
[3] 2020. DeepCrime Replication Package. https://zenodo.org/record/4772465
[4] 2020. An implementation of a multimodal CNN for appearance-based gaze estimation. https://github.com/dlsuroviki/UnityEyesModel.
[5] 2020. Keras Code Examples. Available at https://keras.io/examples/.
[6] 2020. Keras MNIST CNN Model. Available at https://keras.io/examples/vision/mnist_convnet/.
[7] 2020. Keras Movie Recommender Model. Available at https://keras.io/examples/structured_data/collaborative_filtering_movielens/.
[8] 2020. Movie Recommender Dataset. Available at http://files.grouplens.org/datasets/movielens/ml-latest-small.zip.
[9] 2020. Speaker Recognition Dataset. Available at https://www.kaggle.com/kongaevans/speaker-recognition-dataset.
[10] 2020. Speaker Recognition Model. Available at https://keras.io/examples/audio/speaker_recognition_using_cnn/.
[11] Boris Beizer. 1984. *Software System Testing and Quality Assurance.* Van Nostrand Reinhold Co., New York, NY, USA.
[12] 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). http://arxiv.org/abs/1604.07316
[13] 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
[14] DeepCrime 2020. DeepCrime. https://github.com/dlfaults/deepcrime.
[15] Yarin Gal and Zoubin Ghahramani. 2016. A Theoretically Grounded Application of Dropout in Recurrent Neural Networks. In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems.* 1019–1027.
[16] Marcio Augusto Guimarães, Leo Fernandes, Márcio Ribeiro, Marcelo d'Amorim, and Rohit Gheyi. 2020. Optimizing Mutation Testing by Discovering Dynamic Mutant Subsumption Relations. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST).* IEEE, 198–208. https://doi.org/10.1109/ICST46399.2020.00029
[17] 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. https://doi.org/10.1109/ICST49551.2021.00030
[18] 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. https://doi.org/10.1109/ASE.2019.00126
[19] Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. [n.d.]. Dataset of Real Faults in Deep Learning Systems. https://zenodo.org/record/3667541#.Xzmily2B3zs. https://doi.org/10.5281/zenodo.3667541
[20] Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. 2020. Taxonomy of Real Faults in Deep Learning Systems. In *Proceedings of the ACM/IEEE 42nd International Conference on Software*

Engineering (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1110–1121. https://doi.org/10.1145/3377811.3380395

[21] 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

[22] 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. https://doi.org/10.1109/ICST46399.2020.00018

[23] Ken Kelley and Kristopher J Preacher. 2012. On effect size. *Psychological methods* 17, 2 (2012), 137. https://doi.org/10.1037/a0028086

[24] 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. https://doi.org/10.1109/ICSE.2019.00108

[25] Bob Kurtz, Paul Ammann, Marcio E Delamaro, Jeff Offutt, and Lin Deng. 2014. Mutant subsumption graphs. In *IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 176–185. https://doi.org/10.1109/ICSTW.2014.20

[26] Bob Kurtz, Paul Ammann, Jeff Offutt, Márcio E. Delamaro, Mariet Kurtz, and Nida Gökçe. 2016. Analyzing the validity of selective mutation with dominator mutants. In *ACM Sigsoft International Symposium on Foundations of Software Engineering*. https://doi.org/10.1145/2950290.2950322

[27] Yann LeCun. 1998. The MNIST Database of Handwritten Digits. (1998). Available at http://yann.lecun.com/exdb/mnist/.

[28] 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

[29] 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

[30] John Ashworth Nelder and Robert WM Wedderburn. 1972. Generalized linear models. *Journal of the Royal Statistical Society: Series A (General)* 135, 3 (1972), 370–384. https://doi.org/10.2307/2344614

[31] 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. https://doi.org/10.1145/3132747.3132785

[32] 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. https://arxiv.org/pdf/2101.07910.pdf

[33] 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

[34] Jeongju Sohn, Sungmin Kang, and Shin Yoo. 2019. Search Based Repair of Deep Neural Networks. *arXiv preprint arXiv:1912.12463* (2019). https://arxiv.org/abs/1912.12463

[35] 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. https://doi.org/10.1109/ICSE.2019.00126

[36] Zan Wang, Hanmo You, Junjie Chen, Yingyi Zhang, Xuyuan Dong, and Wenbin Zhang. 2021. Prioritizing Test Inputs for Deep Neural Networks via Mutation Analysis. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 397–409. https://doi.org/10.1109/ICSE43902.2021.00046

[37] Edwin B Wilson. 1927. Probable inference, the law of succession, and statistical inference. *J. Amer. Statist. Assoc.* 22, 158 (1927), 209–212. https://doi.org/10.1080/01621459.1927.10502953

[38] Erroll Wood, Tadas Baltrušaitis, Louis-Philippe Morency, Peter Robinson, and Andreas Bulling. 2016. Learning an Appearance-Based Gaze Estimator from One Million Synthesised Images *(ETRA '16)*. Association for Computing Machinery, New York, NY, USA, 131–138. https://doi.org/10.1145/2857491.2857492

[39] Lotfi A Zadeh. 1965. Fuzzy sets. *Information and control* 8, 3 (1965), 338–353. https://doi.org/10.1016/S0019-9958(65)90241-X

[40] 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