Self-assessment Oracles for Anticipatory Testing

TECHNICAL REPORT: TR-Precrime-2019-03

Gunel Jahangirova, Paolo Tonella

An Empirical Evaluation of Mutation Operators for Deep Learning Systems

Project no.: 787703
Funding scheme: ERC-2017-ADG
Start date of the project: January 1, 2019
Duration: 60 months

Technical report num.: TR-Precrime-2019-03
Date: August 2019
Organization: USI Università della Svizzera Italiana
Authors: Gunel Jahangirova, Paolo Tonella
Dissemination level: Public
Revision: 1.0

Università della Svizzera Italiana (USI)

Principal investigator: Prof. Paolo Tonella
E-mail: paolo.tonella@usi.ch
Address: Via Buffi, 13 – 6900 Lugano – Switzerland
Tel: +41 58 666 4848
Project website: https://www.pre-crime.eu/

Abstract

Deep Learning (DL) is increasingly adopted to solve complex tasks such as image recognition or autonomous driving. Companies are considering the inclusion of DL components in production systems, but one of their main concerns is how to assess the quality of such systems. Mutation testing is a technique to inject artificial faults into a system, under the assumption that the capability to expose (*kill*) such artificial faults translates into the capability to expose also real faults.
Researchers have proposed approaches and tools (e.g., *DeepMutation* and *muNN*) that make mutation testing applicable to deep learning systems. However, existing definitions of mutation killing, based on accuracy drop, do not take into account the stochastic nature of the training process (accuracy may drop even when re-training the un-mutated system). Moreover, the same mutation operator might be effective or might be trivial/impossible to kill, depending on its hyper-parameter configuration. We conducted an empirical evaluation of existing operators, showing that mutation killing requires a stochastic definition and identifying the subset of effective mutation operators together with the associated most effective configurations.

Contents

1 Introduction

Recent advancements in Deep Neural Networks (DNNs) have made them capable of dealing with relevant, real world problems in the area of image processing, speech recognition and natural language processing, just to mention a few. Correspondingly, the application scenarios are growing exponentially, ranging from domains such as autonomous vehicles to automated customer assistance and financial bots. However, the quality of systems based on Deep Learning (DL) remains a big challenge. In fact, the behaviour of these systems depends on how they are trained (i.e., the training data, the training process, the adopted model architecture, etc.) and is not programmed explicitly in the code. As a consequence, faults are not necessarily the logical faults that affect traditional software: they can be faults that affect the way these systems are trained.

The existence of DL specific faults has been recognized by researchers [17, 21] who investigated mutation testing for DL systems. They proposed DL specific mutation operators to inject artificial faults that mimic possible issues affecting the training of DL systems. For instance, they proposed mutation operators that shuffle, remove, duplicate (part of) the training data; that add noise to training data or change the labels of training data; that modify the DNN model structure, by adding/removing layers of changing the activation function of neurons. DeepMutation [17] and muNN [21] are two examples of tools that implement two partially overlapping sets of DL mutation operators.

While the usage of mutation testing with DL systems is potentially an effective technique for quality assurance, the familiar notions of mutation killing, equivalent mutant and trivial mutant do not translate to DL systems in a straightforward way. In a traditional (deterministic) software system, a mutant is killed if original program and mutant behave differently in a test scenario. In a DL system it is common to observe behavioural differences even when the system is re-trained on the same dataset, without applying any mutation to it, just because the training process is usually non deterministic. Hence, on the same input a DL system might behave correctly after one training session and incorrectly after another. So, misbehaviour on a single test input is not enough to kill a mutant. Even an aggregate metric, such as accuracy computed on the entire test set, may be not enough, since the accuracy values measured on the test set fluctuate around the mean upon re-training. Hence, accuracy drop is also not a valid mutation killing criterion, unless it is formulated in statistical terms. Moreover, DL mutation operators come with several hyper-parameters to be configured. Depending on the configuration, a given mutation operator might be trivial to kill (any test set will experience huge accuracy drop), impossible to kill (even the weakest test set will not experience any drop) or effective (i.e., it can discriminate strong from weak test data).

In this paper we report our empirical investigation of 20 DL mutation operators proposed in the literature. Results show that the notion of mutation killing is not meaningful without a statistical definition. While results indicate that most (17/20) mutation operators are potentially useful and effective, their configuration must be selected carefully to avoid making them trivial or impossible to kill. We also confirm the need for DL specific mutation operators, by comparing them with syntactic mutations, and we show that some of the proposed DL test adequacy criteria (namely, surprise adequacy [13]) correlate well with mutation score, which provides relevant, mutation-based empirical evidence in support to those approaches.

2 Background

The works by Ma et al. [17] and Shen et al. [21] represent initial attempts to define mutation operators for DL systems. Ma et al. [17] introduce a framework called *DeepMutation*, which includes two families of mutation operators: *source level* and *model level* operators. Source level mutation operators modify the original training data or the model structure *before training*. In the former category, the mutation operators that target training data can be applied at two levels: global

and local. The *global level* operator is applied to all the training data, while the *local level* operator is applied only to specific types of data (e.g., all those with a given label) within the training set. The authors note that source level mutation operators follow the same principle as the mutation operators for traditional software systems, as they introduce faults directly into the programming sources of DL systems, under the assumption that training data are are deemed as the equivalent of the source code for DL systems.

Model level mutation operators change the weights, biases or structure of an *already trained model*. The mutation operators proposed by Shen et al. [21], as part of a framework called *muNN*, also fall into this category. Model level mutation operators are less costly, as unlike source-level operators, they require no re-training.

Table 1 lists all the mutation operators from DeepMutation (DM) and muNN along with their short descriptions. Column *Source* shows the framework that proposed such operator. Column *Type* indicates whether it is a source level (SL) or a model level (ML) mutation operator, while column "*Target*" points to the part of a DL system the operator is applied to.

Table 1: Mutation Operators

Operator Name	Description	Source	Type	Target	Hyper-Parameters
Data Shuffle	Shuffle training data	DM	SL	data	-
Data Missing	Remove part of training data	DM	SL	data	label, percentage
Data Repetition	Duplicate a portion of training data	DM	SL	data	label, percentage
Label Error	Change the label for a data	DM	SL	data	label, percentage, change method
Noise Perturbation	Add noise to training data	DM	SL	data	label, percentage, flip perc. or st. dev.
Layer Removal	Delete a layer of DNN	DM	SL	model structure	which layer
Layer Addition	Add a layer to DNNs structure	DM	SL	model structure	which layer
Activation Function Removal	Remove activation function of a layer	DM	SL	model structure	which layer
Gaussian Fuzzing	Change the weight value using Gaussian distribution	DM	ML	weight	percentage, standard deviation
Weight Shuffle	Shuffle the weights of neuron's connections to the previous layer	DM	ML	weight	neuron percentage
Change Weight Value	Change one or more weight values	muNN	ML	weight	percentage, change percentage
Change Bias Value	Change one or more bias values	muNN	ML	bias	percentage, change percentage
Neuron Effect Blocking	Change the outcoming connection weights of a neuron to zero	DM	ML	neuron	percentage
Neuron Activation Inverse	Changing the sign of the output value of a neuron	DM	ML	neuron	percentage
Neuron Switch	Switch two neurons within a layer	DM	ML	neuron	percentage
Delete Input Neuron	Delete a neuron from input layer	muNN	ML	neuron	which neuron
Delete Hidden Neuron	Delete a neuron from hidden layer	muNN	ML	neuron	which neuron
Layer Deactivation	Deactivate the effects of a layer	DM	ML	layer	which layer
Layer Addition	Add a layer to the DNNs structure	DM	ML	layer	which layer
Activation Function Addition	Add an activation function for a layer	DM	ML	layer	which layer
Change Activation Function	Change activation function of a layer	muNN	ML	layer	which layer

Along with the mutation operators, DeepMutation proposes also some metrics to evaluate them. It defines a mutation $m \in M$ as *killed* by test set T if there exists a test input $t \in T$ that is not handled correctly (e.g., it is misclassified) by m. The *average error rate* (AER) of T on M is measured as the average killing ratios across mutants. For example, if there are 5 generated mutant models and for each of them the percentage of test inputs from T that kill each mutation is respectively 0.2, 0.3, 0.4, 0.5 and 0.6, the average error rate will be $(0.2 + 0.3 + 0.4 + 0.5 + 0.6)/5 = 0.4$.

The *mutation score* metric in DeepMutation is defined only for classification systems. More specifically, in case of a k-classification problem with a set of classes $C = \{c_1, ..., c_k\}$, a test input $t \in T$ *kills* the pair $\langle c, m \rangle$, with class $c \in C$ and mutant $m \in M$, if t is correctly classified as c by the original model and if t is misclassified by the mutant model m. Based on this, the mutation score is calculated as the ratio of killed classes per mutant m over the product of the sizes of M and C. For example, if the training data has 10 classes and there are 5 generated mutant models, and for each of them the test data is able to kill 3, 4, 5, 6 and 7 classes, the mutation score will be $(7 + 5 + 3 + 4 + 6)/(5 \times 10) = 0.48$.

The evaluation of DeepMutation was performed using two image classification datasets MNIST [15] and CIFAR [14]. Two known models from the literature have been used for the training of MNIST; one for CIFAR. A random sample of 5,000 and 1,000 images have been used respectively to train and test these models, out of 70.000 available for MNIST and 60.000 for CIFAR. This sampling was repeated 5 times. Source-level mutation operations targeting training data were applied to 1% of the data. Due to the expensive re-training process, only 20 mutants for each source-level mutation operator (10 for the global level and 10 for the local level) were generated. The source-level mutation operators targeting the model structure were applied whenever they did not lead to a failure during the training process (indeed, random mutations of the model structure

might produce badly-formed models), with a maximum of 20 mutations per operator. Model-level mutation operators were applied to 1% of weights or neurons randomly. At the layer level, this operator was applied only if it did not break the structure of the trained model. As no re-training effort is required for these mutation operators, 50 mutations were generated for each of them.

For muNN, the evaluation was performed on the MNIST dataset paired with two models taken again from the literature. The *Delete Input/Hidden Neurons* mutation operators were applied with the number of deleted neurons ranging from 1 to 20. The mutation operators changing weight and bias were applied using 4 different extents (amounts of weight change) and 5 different ratios (proportions of changed weights), i.e., in 20 distinct configurations. To account for the random nature of the operators, each mutant was run 10 times.

The results of DeepMutation's evaluation are reported as an average mutation score for each of the 6 pairs dataset + model, and for each family of mutation operators (source and model level). The details of the experimental data are not publicly available and such aggregated representation of the results lacks information on the performance of each single mutation operator. Moreover, the proposed operators can be applied with a wide range of their parameters, leading to different results. The authors considered only a small, fixed set of configurations.

When it comes to the mutation score and AER metrics proposed with DeepMutation, we can notice that they are potentially unstable metrics if considered without any statistical analysis. In fact, they might change even for the original model, when it is re-trained, not only for a mutant. Moreover, different executions of the re-training process might lead to different mutation scores and AER values. The authors just computed the mean values of these metrics across a small number of runs. In turn, muNN does not provide any formal definition of mutation score, and the impact of mutation operators is measured as the difference in accuracy of the original and mutated model. However, no threshold value was proposed to decide if a mutant is to be considered killed or not. Without such threshold, small accuracy changes, occurring even when re-training the original model, as well as its mutants, might lead to different results and conclusions.

Overall, the existing work on mutation testing for DL systems lacks a comprehensive analysis of the possible configurations of mutation operators. Moreover, there is no solid definition of mutation killing, that takes into account the non-deterministic nature of DL systems, can be applied to non-classification systems and provides statistically significant results. Whether mutation killing is a good test adequacy criterion for test set selection/creation is another, still open, question.

3 Methodology

3.1 Mutation Operators

The mutation operators from Table 1 can be configured in a number of different ways, which will determine the magnitude of the change that they introduce into the DL system. The existing evaluation of these operators was mostly performed with one fixed configuration, which does not provide enough insight into the operators' capabilities. The same operator might be trivial (resp. impossible) to kill under some configurations, while it might be a useful, discriminative operator under other configurations. To provide a more fine-grained analysis, we have implemented these mutation operators with a set of user-configurable hyper-parameters.

Column *Hyper-Parameters* in Table 1 lists the hyper-parameters that can be configured for each mutation operator. The hyper-parameter *label* is used for mutation operators that target training data. When applied to classification systems (i.e., DL systems whose output is a category), the mutation operator affects only training data with the given label. On regression systems (i.e., DL systems whose output is a continuous variable), we create chunks of training data using domain-specific thresholds and then we apply the mutation operator to the selected chunk.

The hyper-parameter *percentage* indicates the percentage of the mutation operator's target set

(i.e., percentage of training data, weights, neurons) that is affected by the mutation. For example, suppose we are applying the *Data Missing* operator with percentage equal to 5%. As the target of this mutation operator is training data, 5% of the training data will be removed.

The hyper-parameters *which layer* and *which neuron* are used for mutation operators that are applied to a specific layer or a specific neuron respectively. The value of the parameter indicates the specific layer or neuron to be mutated. Let's say we have a DNN model with 6 layers, two of which are dropout layers. If the mutation operator *Layer Removal* with the value of the parameter *which layer* equal to `dropout_1` is applied, the first dropout layer is removed from the DNN's structure.

The mutation operator *Label Error* has a hyper-parameter *change method*, which can have two values: `random` or `predefined`. When the random change method is chosen, the given label of the training data is changed to another randomly selected label, while with the predefined method, the label is changed to a user-supplied alternative value. The mutation operator *Gaussian Fuzzing* has a parameter *standard deviation*, which specifies the standard deviation of the Gaussian noise added to the model's weights.

The mutation operators *Change Bias Value* and *Change Weight Value* use a hyper-parameter *change percentage*. This parameter (not to be confused with the hyper-parameter *percentage*), indicates the magnitude of the relative (percentage) change to be applied. As the mutation operator *Change Weight Value* is similar to *Gaussian Fuzzing*, we exclude it from our analysis. Similarly, *Delete Input Neuron* and *Delete Hidden Neuron* are versions of the mutation operator *Neuron Effect Blocking*, therefore we exclude also these two.

The mutation operator *Noise Perturbation* either flips or adds Gaussian noise to training data. In the former case, input vectors have binary entries that are flipped and the percentage being flipped is the hyper-parameter *flip percentage*. In the latter case, input vectors have numerical entries that are transformed by Gaussian noise addition, the hyper-parameter being the *standard deviation* of the noise. Let us consider a DL system with a training set consisting of 1000 black and white images (i.e., images whose pixels are 0 or 1). If the *percentage* hyper-parameter is equal to 10% and *flip percentage* is equal to 25%, the mutation will be applied to 100 images out of 1000, changing 25% percent of each image's pixels into their binary complement.

3.2 Mutation Killing

Let $P = (TP, TrainD, TestD)$ be our original DL system. It has three components: (1) program TP to train the system, i.e., the source code that defines model structure, training hyper-parameter values, etc.; (1) training data $TrainD$; (3) testing data $TestD$. We combine the training and testing datasets available for P into one set $TrainD \cup TestD$. Out of this combined set, we create n random samples of training and testing data using cross-validation (80% and 20% split), which we denote as $TrainD = (TrainD_1, ..., TrainD_n)$ and $TestD = (TestD_1, ..., TestD_n)$. As a result, we get n DL systems $(P_1, ..., P_n)$, each consisting of the triplets $(TP, TrainD_i, TestD_i)$, $i \in [1 : n]$.

Let $\mu(par_1, .., par_k) \in MO$ be a mutation operator with k hyper-parameters. We apply μ to each $P_i \in P$, generating as a result a set of mutated DL systems $M = (M_1, ..., M_n)$. If μ is a source level mutation operator that targets the training data, then it will change each $TrainD_i \in TrainD$. If instead μ is a source-level mutation operator that targets model structure, it will change TP. In both cases, n re-training will have to be performed to obtain the desired set M. In case μ is a model level mutation, it will be applied to already trained models for each $P_i \in P$ without any need for re-training to get the set M.

Given the original models $P_1, \ldots, P_n$ and the mutated models $M_1 ... M_n$, we can analyse whether the available test suite is able to reveal the differences between these two set of models, i.e., to kill the mutant. We evaluate each couple (P_i, M_i) using test data $TestD_i$, $i \in [1 : n]$. For classification systems, the *accuracy score* measures how well the model is performing for a given test dataset, as the complement of the mis-classification rate. For regression systems such an evaluation requires an oracle that will decide whether the expected value and the output of the model are close enough.

Usually this problem is addressed by introducing thresholds that identify the acceptable deviation between two values. However, in our approach there is no need for such an oracle, as for regression systems we use the output values of both P_i and M_i and compare them directly.

Once we have obtained the accuracy scores or output values produced by P and M, we need to assess the likelihood that the effect observed on the mutated models does not occur by chance. We do this by calculating the p-value using a generalised linear model (GLM) [19]. In our general linear model, the accuracy or output of the model is the *dependent variable*, while whether the model is the original or the mutated one is the *independent variable*. A high p-value means that the linear coefficient of the dependent variable has a high probability of being zero (null hypothesis), which means that the dependent variable is not a useful predictor of the dependent one, while a low p-value suggests that the coefficient has a low probability of being zero. To ensure that the difference between original and mutated models is statistically significant, we require the p-value to be less than or equal to α =0.05. A high p-value might be due to insufficient statistical power. To avoid Type II errors (accepting a false null hypothesis) due to low statistical power, we also compute the statistical power β and make sure we have enough data points to reach $\beta > 0.95$.

A low p-value does not tell us if the accuracy drop of regression deviation is large, medium, small or negligible. To get this information, we compute the *effect size* [12], which is a quantitative representation of the magnitude of difference between two distributions. One of the commonly used measures of the effect size is Cohen's d. This value can be interpreted using the thresholds provided by Cohen [7], i.e. $|d| < 0.2$ – "negligible", $|d| < 0.5$ – "small", $|d| < 0.8$ – "medium", otherwise – "large". To ensure that the size of the difference between the outcome of the original and mutated models is prominent, we require that the value of d is not "negligible", i.e., it must be greater than or equal to 0.2.

We adopt the following definition of mutation killing:

$$isKilled(P, M, TestD) = \begin{cases} true & \text{if } \textit{effectSize} \geq 0.2 \\ & \text{and } \textit{p\_value} < 0.05 \\ false & \text{otherwise} \end{cases}$$

By *mutation score* MS we mean the fraction of mutants killed by $TestD$:

$$MS = \frac{\{\mu \in MO | isKilled(P, \mu(P_1, \ldots, P_n), TestD)\}}{|MO|}$$

where MO is the set of mutation operators being considered.

4 Experimental Results

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

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

In the first research question we investigate whether mutation operators are useful for test suite quality assessment. In case the mutant is not killed by the largest test suite available, we mark it as an *equivalent* mutant. We then filter out the *trivial mutations*, i.e., those that can be killed by all test suites, including the "weakest" ones.

RQ2 [Definition of killed mutant]: *Does the proposed mutation killing criterion differ from the threshold based criterion? How does it relate to hyper-parameter configuration?*

RQ2 aims to check whether the proposed notion of mutation killing exhibits different behaviour than the threshold-based approach used in the existing literature. To perform this comparison, we examine each mutation operator using a wide range of parameters and identify the configurations that are killable based on our definition. We also record the accuracy drop caused by each mutation and check whether it is higher than a predefined threshold. Our approach provides a single

statistical verdict for a given number of runs. However, the threshold-based approach might provide different outcomes for each run, i.e. while the difference in accuracy is above the threshold in one case, it can be below it in the other. Therefore, the cases of special interest for this analysis are not only the ones where the two approaches disagree, but also the ones where the threshold-based approach provides inconsistent results across different runs.

RQ3 [Syntactic mutations]: *Are DL mutation operators needed or can we obtain equivalent mutations by means of standard, syntactic operators?*

In traditional software systems, mutation testing is performed by introducing syntactic changes into the source code of the software under test. In RQ3, we analyse whether these simple mutations are capable of causing the DL system's behaviour to deviate from the original one, and whether the effect produced by them is similar or different from that caused by the DL-specific mutation operators.

RQ4 [Comparison with other adequacy criteria]: *Does the DL mutation score correlate with other DL adequacy criteria, such as neuron coverage and surprise coverage?*

In RQ4 we check whether the mutation score based on our notion of mutation killing correlates with the test adequacy criteria proposed in the literature, such as *neuron coverage* [20] and *surprise adequacy* [13]. We measure each metric on a set of different test suites and analyse whether the metrics are sensitive to the changes in the size and the quality of the test data.

4.1 Datasets and DL Systems

For the evaluation of existing mutation operators we have used both classification and regression systems. For the classification systems we have selected two publicly available datasets: MNIST [15] and CIFAR [14].

MNIST is a dataset of handwritten digits which contains 60,000 images of training and 10,000 images of testing data. We used two models, MNIST-M1 [4] and MNIST-M2 [5], available as part of Keras' documentation on training of models for the classification of MNIST images. MNIST-M1 is a convolutional network with 8 layers. MNIST-M2 is an example of transfer learning, which first trains a simple convolutional network for the first 5 digits. Then it freezes the convolutional layers and it fine-tunes the dense layers for the classification of the remaining digits.

CIFAR10 is an object recognition dataset with 10 different classes composed of 50,000 images for training and 10,000 images for test. Similarly to MNIST, for CIFAR10 we have also used two models CIFAR-M1 [2] and CIFAR-M2 [3], both from Keras' documentation. CIFAR-M1 is a convolutional network with 18 layers, while CIFAR-M2 uses the residual neural network proposed in the work by He et al. [10].

As a representative of regression systems, we have used a self-driving car from the Udacity challenge. To train this model, we used the dataset from the work by Stocco et al. [22]. The dataset contains 4,411 images available for training and 1,103 images for testing. The goal of this DL system is to predict the steering wheel angle from the images.

Table 2: Subject DL Systems

Model	Number of Epochs	Accuracy (MSE)
MNIST-M1	12	99.03%
MNIST-M2	5	99.39% and 98.97%
CIFAR-M1	25	76.04%
CIFAR-M2	10	74.33%
Udacity	100	0.21

Table 2 shows the number of epochs for which we have trained each model and the accuracy

achieved as a result. For MNIST-M2 two accuracies are provided, as it consists of two models, i.e. the one for the first 5 digits and the other for the last 5 digits. As Udacity is not a classification system, instead of reporting its accuracy, we report its Mean Squared Error (MSE), i.e., the average distance between predicted and expected steering angle.

4.2 Mutation Generation

As shown in Table 1, each mutation operator has a number of hyper-parameters, the value of which might affect its performance immensely. To account for this, we analysed each operator considering a large set of fine-grained hyper-parameter combinations.

For the mutation operators that have parameters "*which layer*" or "*which activation*" we have performed an exhaustive analysis and applied them to all available layers and activation functions.

For the source level mutation operators we set the *percentage* hyper-parameter to each of the following values: 5, 10, 25, 50, 75, 80, 85, 90, 95 and 100. The first five numbers in this set increase with bigger steps, while after 75 they are more dense. We have selected this spectrum of values under the assumption that the impact of the mutation operator for small percentage values will not be significant, but once the percentage reaches a higher value, the effect of even small steps becomes meaningful.

For the mutation operator "*Data Missing*" we have considered two levels: the whole training data and a specific label. In the latter case, the mutation operator deletes a percentage of the data that belongs to the selected label. In the former case, it removes part of the overall training data (100% is never applied to this case). For the "*Label Error*" operator we change the label of training data either to any other label randomly or to some predefined label. For Udacity, which is not a classification system, we divide its training data into groups to support the notion of a label. For this, we first calculate the standard deviation σ_θ of the steering angle θ available for the training data images. Then we form three chunks: 1) angles $\theta \leq -\sigma_\theta$; 2) angles $\theta > -\sigma_\theta$ and $\theta < \sigma_\theta$; 3) angles $\theta \geq \sigma_\theta$. Each chunk is interpreted as a label. Since results are consistent across labels, for all five subject systems we report only the results obtained when mutating one particular label. For MNIST-M1 and CIFAR we used label 0, while for Udacity we used the second chunk described above. MNIST-M2 is an exception, as it consists of two models each training for 5 images. We apply all the mutation operators that have *percentage* as a parameter, to two different labels: zero and five.

The "*Noise Perturbation*" mutation operator has a third parameter, which is either *flip percentage* or *standard deviation* depending on the subject system. In MNIST, in order to introduce noise to training data, we flip the pixels of the images from black to white and vice versa, i.e., in this case the flip percentage parameter is used. For CIFAR10 and Udacity we use Gaussian noise with the standard deviation parameter.

As the model level operators are applied to an already trained model, they are more disruptive in nature than the source level ones. Our initial trials with these operators showed that passing big values for the *percentage* parameter generates models with a very low accuracy. Therefore, we used values 0.1, 0.5, 1, 5 and 10 instead.

Our definition of mutation killing requires a large number of runs for each configuration of the mutation operator. To obtain these runs, we generated N datasets for each of our subject systems using N-fold cross-validation. For mutation operators such as "*Delete Layer*", where no random manipulation of training data is involved, we run the mutation operator on each cross-validation dataset, thus obtaining $N = 20$ runs for each mutation (e.g., for each deleted layer). For mutation operators involving random manipulation of training data, such as "*Data Missing*", where we randomly delete a percentage of data, results may change at each application of the operator. Therefore, to account for randomness, we applied the mutation operator 5 times to a smaller number of cross-validation datasets (i.e., with $N = 5$), getting in total 25 runs as a result.

Table 3 shows the number of mutations generated for each subject as part of our experiments. For some mutation operators, the number of configurations is fixed, therefore the number of mutations

is the same for all subjects. For example, the "*Data Missing*" is applied 5 times randomly to 11 different percentages and 5 datasets, resulting in 11 configurations of the same mutation operator and 275 generated mutants. On the other side, the "*Delete Layer*" operator is applied for each layer, therefore the number of generated mutants is different for each model. This is the reason why the number of mutants is different across our subject systems. Overall, our experiments required the generation of 16,300 source level mutations (each of which required retraining) and 7,460 model level mutations.

Table 3: Generated Mutations

Model	Source Level	Model Level	Overall
MNIST-M1	2,815	1,820	4,635
MNIST-M2	4,745	1,380	6,125
CIFAR-M1	3,020	1,580	4,600
CIFAR-M2	2,735	1,420	4,155
Udacity	2,985	1,260	4,245
	16,300	7,460	23,760

Due to our expensive experimental procedure, we have used multiple machines for the generation and evaluation of our mutations. One of them was an Alienware machine, with *i9* processor, *32 GB* of memory, and an *Nvidia GPU 2080 TI* with *11 GB* of memory. In addition to that, we used a number of `p3.2xlarge`, `g3s.xlarge` and `g3.4xlarge` Amazon EC2 instances, with Deep Learning AMI (Ubuntu 16.04) Version 24.2 installed on them.

4.3 Procedure

4.3.1 RQ1 (Mutation Operators)

After generating mutants, we have applied our definition of mutation killing to find out which hyper-parameter values make them killable by the largest available test set. We marked the set of non killed mutants as *likely equivalent*. Out of the 20 mutation operators analysed, for 17 there exists a configuration that is not equivalent. Three mutation operators, "*Data Shuffle*", "*Data Repetition*" and "*Change Bias Value*", produced mutants that were not killed under any analysed combination of the hyper-parameters.

From the set of killable mutants we then filter out the *trivial mutations*, i.e. the ones that get killed by any test suite. To perform this analysis we generated different test suites based on two metrics: *size* of the test suite and *quality* of test data inside the test suite. For the size, we randomly sampled 5% and 50% of the original test suite. Our assumption is that the number of killed mutations will decrease as the size of the test suite becomes smaller.

For the test quality based approach, we use the *confidence* of the original DL system in the classification or value prediction of a given test input. We select the confidence threshold so that it divides the test data into two groups of comparable size. We then group the elements that are misclassified or classified correctly with a confidence less than the threshold into the *difficult* test set. The remaining inputs form the *easy* test set. Note that misclassification is taken into account only for classification systems, for regression systems the division into two sets is based only on the confidence.

We expect that the *difficult* test set will have a higher mutation score, as these inputs are already challenging for the original system, and therefore their classification/prediction is more likely to be affected by the faults injected into the DL system. We keep the size of *easy* and *difficult* the same, to ensure that the mutation score for these test suites is not affected by the size, but only by the quality of test data. For this, across all 20 test sets obtained using cross-validation, we determine the size of

the smallest easy or difficult test set. Then we use this value to randomly sample that number of elements from each test set. As a result of this process, we generate 20 test sets of the same size.

Table 4 shows the selected threshold and the sizes of test suites for each subject program. The sizes for MNIST-M2 are smaller than for MNIST-M1, as we evaluate the second model which classifies only the last 5 digits of the dataset.

Table 4: Test Suites

Dataset	TS100	TS50	TS5	Conf.	TSEasy	TSDiff
MNIST-M1	14,000	7,000	700	1.0	3,644	3,644
MNIST-M2	6,805	3,408	1990	1.0	1,672	1,672
CIFAR	12,000	6,000	600	0.7	4,922	4,922
Udacity	1,103	550	55	0.7	460	460

Table 5: Hyper-parameter *percentage* for Non-Equivalent and Trivial Mutations (non equiv param, trivial param)

Operator Name	MNIST-M1	MNIST-M2	CIFAR-M1	CIFAR-M2	Udacity
Data Missing: Label	75,90	50,95	75,80	75,85	50,80
Data Missing: Whole Data	25,99	10,95	5,75	50,90	25,90
Label Error: Random	50,75	25,75	25,50	25,75	25,50
Label Error: Predefined	10,50	25,50	25,75	25,80	25,50
Noise Perturbation	75, 85	25,90	25,75	25, 80	25,75
Gaussian Fuzzing	25,75	25,50	25,75	25,100	25,50
Weight Shuffle	5,10	5,10	0.1,0.5	0.1,0.5	0.1,0.5
Neuron Effect Blocking	5,10	5,10	0.5,1	1,5	0.5,5
Neuron Activation Inverse	5,10	5,10	0.1,0.5	0.1,0.5	0.5,1.0
Neuron Switch	0.1,0.5	5,10	1,5	5,10	5,10

4.3.2 RQ2 (Definition of Mutation Killing)

To identify whether our definition of mutation killing differs from the threshold-based we conducted a detailed comparison between them. We use three different thresholds for the classification systems: 1%, 5% and 10%. Our definition of mutation killing gives a single boolean verdict for n runs of the mutation operator. This is different from the threshold-based approach, for which we get n outcomes on whether the mutation is killed or not. We introduce the *agreement rate* metric to measure how consistent is the threshold-based approach across n runs in regarding the mutant as *killed*. If in m out of n cases the accuracy drop is greater than the selected threshold, the consistency rate is equal to m/n. Therefore, if the outcome of the threshold-based approach is always *killed*, then its agreement rate will be equal to 1.

4.3.3 RQ3 (Syntactic Mutations)

The only part of a DL system to which syntactic mutations can be applied is the implementation of the training program. Usually the training program contains just the model structure, values of hyper-parameters and calls to libraries that support the training process. Therefore, the number of lines of code available for mutation is low, as demonstrated in the column "*LOC*" of Table 8.

For syntactic mutant generation we used MutPy [6], a mutation tool for Python 3.3+ code that applies mutation operators to the abstract syntax tree. It supports 27 different mutation operators.

Column "*#Mutations*" shows the overall number of syntactic mutations generated by MutPy for each DL system. From the initial set of mutations, we excluded the ones that remove statements such as `model.compile()`,`model.fit()`,`model.save()` or alter the statements that calculate and save the accuracy of the trained model. They are not meaningful for the evaluation of DL systems, as they force the source code not to train or save the model, or provide wrong values for the evaluated metric. We report the number of these mutants in column "*Excluded*".

Column "*Failures*" shows the number of mutants that cause compilation errors. Inspection of the error messages showed that the most common reason is the change in the shape of the input data or of the layer. The second common reason is mutants that change the name of the activation function, loss function or optimiser to an empty string or to some predefined string which is not a valid name for any of these functions/parameters.

4.3.4 RQ4 (Adequacy Criteria)

We compare our approach with two adequacy criteria proposed in the literature: neuron coverage [20] and surprise adequacy [13]. Neuron coverage of a set of test inputs is defined as the proportion of activated neurons over all neurons when all available test inputs are supplied to the DNN.

The surprise adequacy of an input is measured as the difference in DL system's behaviour between the input and the training data. According to this criterion, a good test input should be sufficiently but not overly surprising compared to training data. Surprise adequacy comes in two versions: likelihood-based and distance-based. The Distance-based Surprise Adequacy (DSA) is measured using the Euclidean distance between the activation traces of a new input x and the activation traces observed during training. The Likelihood-based Surprise Adequacy (LSA) uses kernel density estimation to estimate the probability density of each activation value, and obtains the surprise of a new input as its (log-inverse) probability, computed using the estimated density.

Both works proposing neuron coverage and surprise adequacy come with a replication packages that make their tools publicly available. We reused their code to measure neuron coverage, LSA and DSA for each of the test suites described in Table 4 and to contrast them with the mutation score. However, as the tool measuring surprise adequacy is implemented only for classification systems, we had to exclude Udacity from this analysis.

Table 6: Layer and Activation Function Mutations (mutations/killable mutations/trivial mutations)

Operator Name	Type	MNIST-M1	MNIST-M2	CIFAR-M1	CIFAR-M2	Udacity
Layer Removal	SL	6/4/2	10/3/0	9/3/1	7/4/1	13/4/2
Layer Addition	SL	5/4/1	9/1/1	8/1/0	6/1/0	11/5/4
Activation Function Removal	SL	2/1/0	3/2/0	4/3/1	3/2/1	6/2/1
Layer Deactivation	ML	0/0/0	2/0/0	0/0/0	3/2/1	0/0/0
Layer Removal	ML	0/0/0	0/0/0	3/0/0	5/2/1	0/0/0
Activation Function Addition	ML	5/2/1	0/0/0	0/0/0	4/2/1	8/2/1
Change Activation Function	ML	1/1/1	0/0/0	0/0/0	4/1/1	6/2/2

4.4 Results

All our experimental data is publicly available as part of our replication package [**?**].

4.4.1 RQ1 (Mutation Operators)

After running each non-equivalent mutation operator on each of the considered test suites, we identify the ones that are killed by any test suite. Table 5 provides a summary of these data for the mutations that change the training data before training or the weights/neurons after the

Table 7: Statistical vs. Threshold-Based Mutation Killing (proportion of runs in which thresholds, equal to 1%, 5% or 10%, agree with the statistical verdict stating that the mutation is *killed*)

Operator Name	MNIST-M1			MNIST-M2			CIFAR-M1			CIFAR-M2			Udacity		
	1%	5%	10%	1%	5%	10%	1%	5%	10%	1%	5%	10%	0.1	0.25	0.5
Data Missing: Label	0	0	0	0	0	0	0.32	0	0	0.24	0	0	1	1	1
Data Missing: Whole Data	0	0	0	0	0	0	0.64	0.16	0	0.32	0.16	0	1	1	1
Label Error: Random	0	0	0	0	0	0	0.24	0	0	0.32	0	0	1	1	1
Label Error: Predefined	1	0.6	0	0	0	0	0.44	0	0	0.16	0	0	1	1	1
Noise Perturbation	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1
Layer Removal	0	0	0	0	0	0	0.8	0.56	0	0.64	0.56	0	1	1	1
Layer Addition	0	0	0	0	0	0	0.64	0.2	0	0.32	0.16	0	1	1	1
Activation Function Removal	0	0	0	1	1	1	0.8	0.8	0.8	0	0	0	1	1	1
Gaussian Fuzzing	0	0	0	0	0	0	0.56	0	0	0.32	0	0	1	1	1
Weight Shuffle	0.56	0.16	0.16	0.28	0.28	0.2	0.32	0	0	0.28	0	0	1	1	1
Neuron Effect Blocking	0.36	0.2	0.08	0.28	0.24	0.16	0.12	0	0	0.36	0.16	0	1	1	1
Neuron Activation Inverse	0.84	0.36	0.32	0.32	0.24	0.24	0.6	0.32	0.2	0.84	0.32	0.2	1	1	1
Neuron Switch	0.16	0.16	0.16	0.24	0.24	0.24	0.64	0.2	0.16	0.24	0.2	0.16	1	1	1
Layer Deactivation	-	-	-	0	0	0	-	-	-	0	0	0	-	-	-
Layer Removal	-	-	-	0.8	0.48	0.08	0.8	0.8	0.8	0.8	0.48	0.08	-	-	-
Activation Function Addition	0.48	0.48	0.48	-	-	-	-	-	-	0.36	0.32	0	1	1	1
Change Activation Function	0.8	0.8	0.8	-	-	-	-	-	-	0.8	0.48	0.2	1	1	1

training. For each subject we report two numbers. The first number shows the hyper-parameter configuration of each mutation operator for which the mutation gets killed. For example, the "*Data Missing*" operator for MNIST-M1 kills the mutant once the value of parameter *percentage* becomes 75% or higher. The second number denotes the configuration from which the mutation operator becomes trivial. In case of the "*Data Missing*" operator for MNIST-M1 this number is 90%.

Overall, in most of the cases the source level mutation operators applied to training data become killable when the *percentage* hyper-parameter is equal to or higher than 25%, which shows that small changes to the training data do not affect the behaviour of the model. The value of "percentage" parameters that make the mutation killable or trivial varies across subjects and mutation operators.

For the mutations changing the layers and activation functions of the model, the specific layer or the specific activation function that makes a mutation killable depends on the model structure. Therefore, instead of reporting such model-dependent hyper-parameters, in Table 6 we indicate the mutation type (model level or source level), as some of the operators are very similar and only differ in wether they are applied before or after training, and we show the following triples of numbers: *number of applications of the mutation operator/number of killable mutations/number of trivial mutations*. The first number indicates how many layers/activations functions could be mutated by the mutation operator without breaking the model structure. The second number shows how many of these were killable, while the third denotes how many, out of the killable ones, were trivial.

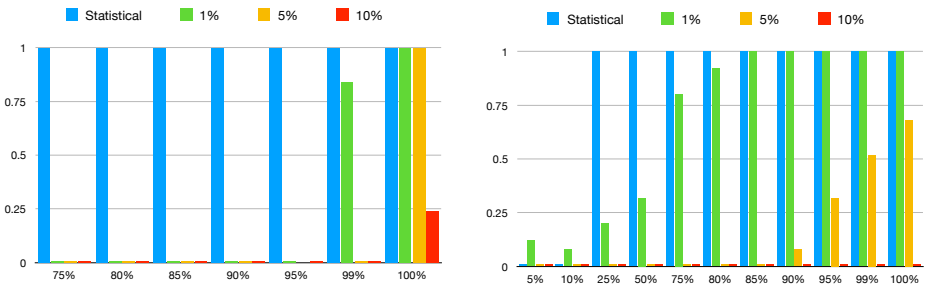
Summary: *Out of the 20 proposed ones, we identified 17 "interesting" (non equivalent, non trivial) DL mutation operators and for each of them we determined the hyper-parameters that make them non equivalent and non trivial.*

4.4.2 RQ2 (Definition of Mutation Killing)

We measured the agreement rate for all mutation operators under all configurations that we have evaluated. Due to space constraints, in Table 7 we report only the agreement rate measured for the configuration with the lowest *percentage* parameter that makes the mutation killable. For the layer/activation related mutation operators, we report only the highest values of agreement rate.

As Table 7 shows, for the source level mutation operators the threshold based approach is usually not able to recognise the killed mutation, as the drop in accuracy is less than the threshold. For model level mutation operators, the agreement rate is mostly higher than 0, however it is rarely equal to 1, which shows that the threshold-based approach is often inconsistent across multiple runs.

For Udacity the mutation is considered killed if there is at least one image the for which the steering



MNIST-M1: "Data Missing: Label" CIFAR-M1: "Label Error: Random"

Figure 1: Agreement rate at increasing values of hyper-parameter *percentage*, for statistical and threshold based approaches, with thresholds equal to 1%, 5% and 10%

angle prediction is different from the correct one by a value more than a given threshold (we used 0.1, 0.25 and 0.5). This criterion is very weak, as across all the mutations there is always such an image. As a result, the mutations are always killed and the agreement rate is always equal to one.

To provide a more complete picture on the behaviour of the agreement rate across the parameter values of the mutation operators, we analyse two case studies in more detail. Figure 1 (left) plots the changes in agreement rate for the "Data Missing: Label" operator applied to MNIST-M1. Until the value of *percentage* is less than 75%, both approaches agree that the mutation is not killed. Once the value is 75% our approach considers the mutation as *killed*. However, none of the 3 thresholds reach this outcome until the hyper-parameter *percentage* is 99%, at which the 1% threshold's agreement rate is 0.84 (21 runs out of 25 report the mutation as *killed*). Only when the value of percentage is 100%, i.e. the maximum possible value, the 1% and 5% thresholds reach an agreement rate of 1, while the 10% threshold still does not.

Figure 1 (right) provides a similar analysis for the mutation operator "Label Error: Random" applied to CIFAR-M1. For the hyper-parameter *percentage* equal to 5% and 10%, the agreement rate for the 1% threshold is greater than zero, i.e. there are some runs for which the mutation is considered killed. For our approach the mutation is killed only once *percentage* is equal to 25%. When *percentage* is equal to 85% or higher, our definition and the 1% threshold both regard the mutant as *killed*.

Summary: *The threshold-based definition of mutation killing is problematic because it often produces inconsistent results across runs and it requires careful selection of the thresholds, which depend on the mutation operator's hyper-parameters. On the contrary, the statistical notion of mutation killing accounts for all available runs and does not require to set any accuracy drop threshold.*

Table 8: Syntactic Mutations

Subject	LOC	#Mutations	Excluded	Failures	Analysed	Killed
MNIST-M1	85	68	11	34	23	5
MNIST-M2	102	119	11	85	23	7
CIFAR-M1	62	92	25	35	32	14
CIFAR-M2	176	40	1	8	31	11
Udacity	52	70	20	14	36	13
		389	68	176	145	50

4.4.3 RQ3 (Syntactic Mutations)

As shown in Table 8, among 389 syntactic mutations generated for 5 different models, 176 lead to failures. Moreover, among 145 analysed mutants, only 50 were killed by the largest available test set (i.e., the mutation caused a statistically significant accuracy drop). This indicates that syntactic mutations mostly lead to the injection of simple faults that either result in a failure or produce syntactic changes not related with the behaviour of the model being trained.

We performed a qualitative analysis of the killed syntactic mutations, to check whether they are similar in nature to the DL-specific ones. Out of 50 killed mutations, 10 change the way the training or testing data is normalised, 9 remove a layer from the model, 3 change the shape of the layer and the other 3 change the dropout rate of the layer. There are also single cases that alter hyper-parameters such as the batch size, learning rate, number of epochs or the kernel and pool size of the layer.

Summary: *Apart from deleting a layer from the model, syntactic mutation operators are not similar to the DL-specific ones analysed in this paper. This justifies the definition of DL specific mutation operators and the construction of a DL specific mutation tool.*

4.4.4 RQ4 (Adequacy Criteria)

Table 9 lists the coverage values obtained for each adequacy criterion. The results show neuron coverage is unable to distinguish between big/small or easy/difficult test suites, as its value is almost a constant across all the test suites of the same subject. Similarly, DSA provides very similar values for the test suites of MNIST-M1 and MNIST-M2. In case of CIFAR-M1 and CIFAR-M2, the values for TS50 are a bit higher than for TS5. In contrast, LSA can in most cases distinguish the stronger from the weaker test suites for all subjects except CIFAR-M1. quite surprisingly, for MNIST-M1 LSA decreases between TS5 and TS50, despite the latter test suite is larger and has higher mutation killing capability. We think this might be a side effect of approximate estimation of probability densities when the number of samples available for such estimate is low (i.e., just 5% of the whole test set). When it comes to mutation score, across all the subjects it has a higher value for TS50 than for TS5. It is also always successful in distinguishing between easy and difficult test suites, built based on the DNN's confidence level.

Summary: *The mutation score is a very discriminative and powerful adequacy criterion that can provide a reliable assessment of the quality of the test data. Among the other proposed adequacy criteria, LSA is the one that better matches the mutation score.*

Table 9: Adequacy Criteria Comparison

Adequacy Criterion	MNIST-M1				MNIST-M2				CIFAR-M1				CIFAR-M2			
	TS5	TS50	TSEasy	TSDiff	TS5	TS50	TSEasy	TSDiff	TS5	TS50	TSEasy	TSDiff	TS5	TS50	TSEasy	TSDiff
Neuron Coverage	51.00	53.00	53.00	53.00	42.00	46.00	47.00	42.00	35.00	35.00	35.00	34.00	41.00	42.00	43.00	43.00
LSA	27.10	11.95	8.36	24.85	0.665	1.09	0.6	1.04	0.125	0.125	0.125	0.125	3.750	5.19	0.2	0.21
DSA	0.10	0.11	0.10	0.10	0.10	0.11	0.10	0.10	0.20	0.23	0.10	0.10	0.20	0.28	0.1	0.1
Mutation Score	54.00	76.00	33.00	70.00	60.00	80.00	50.00	60.00	33.00	50.00	40.00	45.00	65.00	57.00	25.00	48.00

5 Discussion

Cost of Mutation Testing. In traditional software systems mutation testing is considered expensive, as it requires to run the whole test suite on each mutation. In case of DL-specific mutation operators, some of them require the DL system to be retrained, i.e., even generating a mutant is a costly process. On top of that, our definition of mutation killing requires each mutation operator to be applied n times, thus making the process n times more resource-demanding. This raises a valid question on whether such an approach is applicable in practice and whether the benefits are worth the cost.

Deep Learning is becoming a part of safety-critical systems the proper testing of which is extremely

important. Our results show that the mutation score is a promising test adequacy criterion for DL systems. Moreover, it is the only adequacy criterion that simulates the occurrence of faults. Therefore, we believe that while the machine cost of such an analysis is high, the potential benefits are worth investing on.

Metamorphic Relationships. Metamorphic testing has been advocated as a viable approach for DL systems [18, 23] and the work by Xie et al. [25] proposes a set of metamorphic relationships for machine learning systems. This set has an intersection with the mutation operators analysed in this paper. For example, while in DeepMutation shuffling training data is considered a mutation operator that should change the model's behaviour, this work considers it a metamorphic relationship which actually should not produce any changes. We find this relationship between mutation operators and metamorphic relationships interesting. Indeed, all the mutation operators that we filtered out as equivalent can be used as metamorphic relationships for our subject systems.

Test Quality Improvement. The aim of mutation testing is to assess the quality of a test suite. In traditional software systems, when an unkilled mutant is encountered, it provides guidance into the test quality improvement, as we can see what is the fault injected into the program's logic and based on this we can create a test case that targets the mutant. This does not always hold for DL-specific mutation operators. For instance, a killable change of a model's internal structure (e.g., layers or activation function) might require additional test data to observe a statistically significant accuracy drop. However, the mutation itself is not useful to understand how the available data should be augmented to kill the mutant.

Fault injection. The proposed DL specific mutation operators are useful for the purpose of fault injection. However, we believe that more research should be conducted on the relation between real DL faults and DL mutation operators.

6 Threats to Validity

Construct. One threat to construct validity might be the way we have implemented our definition of mutation killing. We used Cohen's d to measure effect size and GLM to analyse whether the difference between the original and the mutated models is statistically significant. Our list of killed mutants might have been affected by these choices, i.e. it is possible that using some other statistical test would have produced slightly different results.

Internal. The mutation operators that we evaluated in this study are parametric. While for some of them we were able to perform an exhaustive analysis in terms of parameter values, for others this was practically infeasible. Hence, we had to use a list of fixed values. To address this threat to validity, we ensured the list to be large in size and sufficiently fine-grained.

External. All the subject DL systems that we used in our evaluation were trained using Keras models. This may limit the general validity of our results beyond such models. However, we have selected Keras as it is one the most popular DL frameworks [1]. Moreover, we have analysed different types of models, such as convolutional, residual and models with transfer learning.

7 Related Work

The research area of DL testing has grown at an impressive rate in recent years. It is impossible to account for all existing works and contributions in this section, so we provide only a selection of approaches, highlighting the main trends.

The generation of critical inputs for DL systems has been addressed by resorting to a variety of algorithms and techniques, including generative adversarial networks [26] and image transformation for differential testing [20]; search based algorithms [8] and gradient descent [11]. Adversarial

input generation [24] and fuzzing [9] are other interesting directions that eventually produce input data as their final result.

Other researchers investigated adequacy criteria that determine if the DL system under test has been exercised adequately or not by the test data. Neuron coverage [20, 16] is one of the earliest attempt to define adequacy criteria for DL systems. A different view on adequacy is based on the diversity of the considered input data, or their degree of "surprise" [13].

Mutation testing for DL system is a very active area of research and we have already presented the two main existing tools, DeepMutation [17] and muNN [21], in Section 2. The contribution of our work to this area consists of a novel definition of mutation killing and its empirical validation. We are the first to investigate the configuration space of the hyper-parameters associated with DL mutation operators. Moreover, we are also the first who provide empirical evidence on the need for DL specific mutation operators and on the relation between existing DL test adequacy criteria and mutation score.

8 Conclusion

We have conducted a thorough empirical investigation of the DL mutation operators in the existing literature. We have identified the "interesting" mutation operators and the configurations that make them non equivalent and non trivial. We have proposed a new definition of *mutation killing* that takes into account the stochastic nature of DL systems. We have compared this definition to the threshold-based drop in accuracy and demonstrated that the latter is inconsistent across different runs. In our future work, we will investigate the use of DL mutation for fault injection, considering the relation between real faults and mutants.

References

[1] Framework data, 2019.

[2] Keras cifar10 cnn model, 2019.

[3] Keras cifar10 resnet model, 2019.

[4] Keras mnist cnn model, 2019.

[5] Keras mnist transfer learning model, 2019.

[6] Mutpy, 2019.

[7] Jacob Cohen. A power primer. *Psychological bulletin*, 112(1):155, 1992.

[8] Alessio Gambi, Marc Müller, and Gordon Fraser. 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*, pages 318–328, 2019.

[9] Jianmin Guo, Yu Jiang, Yue Zhao, Quan Chen, and Jiaguang Sun. 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*, pages 739–743, 2018.

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

[11] Andrew Ilyas, Logan Engstrom, Anish Athalye, and Jessy Lin. Black-box adversarial attacks with limited queries and information. In *Proceedings of the 35th International Conference on Machine Learning, ICML*, pages 2142–2151, 2018.

[12] Ken Kelley and Kristopher J Preacher. On effect size. *Psychological methods*, 17(2):137, 2012.

[13] Jinhan Kim, Robert Feldt, and Shin Yoo. Guiding deep learning system testing using surprise adequacy. In *Proceedings of the 41st International Conference on Software Engineering*, pages 1039–1049. IEEE Press, 2019.

[14] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. The cifar-10 dataset. *online: http://www. cs. toronto. edu/kriz/cifar. html*, 55, 2014.

[15] Yann LeCun. The mnist database of handwritten digits. *http://yann. lecun. com/exdb/mnist/*, 1998.

[16] 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. Deepgauge: multi-granularity testing criteria for deep learning systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE*, pages 120–131, 2018.

[17] Lei Ma, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Felix Juefei-Xu, Chao Xie, Li Li, Yang Liu, Jianjun Zhao, and Yadong Wang. 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*, pages 100–111, 2018.

[18] Christian Murphy, Kuang Shen, and Gail E. Kaiser. Automatic system testing of programs without test oracles. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA*, pages 189–200, 2009.

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

[20] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. Deepxplore: Automated whitebox testing of deep learning systems. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 1–18. ACM, 2017.

[21] W. Shen, J. Wan, and Z. Chen. Munn: Mutation analysis of neural networks. In *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 108–115, July 2018.

[22] Andrea Stocco, Michael Weiss, Marco Calzana, and Paolo Tonella. Misbehaviour prediction for autonomous driving systems, 2019.

[23] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. Deeptest: automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th International Conference on Software Engineering, ICSE*, pages 303–314, 2018.

[24] Matthew Wicker, Xiaowei Huang, and Marta Kwiatkowska. Feature-guided black-box safety testing of deep neural networks. In *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS*, pages 408–426, 2018.

[25] Xiaoyuan Xie, Joshua W. K. Ho, Christian Murphy, Gail E. Kaiser, Baowen Xu, and Tsong Yueh Chen. Testing and validating machine learning classifiers by metamorphic testing. *Journal of Systems and Software*, 84(4):544–558, 2011.

[26] Mengshi Zhang, Yuqun Zhang, Lingming Zhang, Cong Liu, and Sarfraz Khurshid. 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*, pages 132–142, 2018.