Mutation Testing for Artificial Neural Networks: An Empirical Evaluation

Lorenz Klampfl
CD Laboratory for Quality Assurance Methodologies for Cyber-Physical Systems
Institute for Software Technology
Graz University of Technology
Graz, Austria
lklampfl@ist.tugraz.at

Nour Chetouane
CD Laboratory for Quality Assurance Methodologies for Cyber-Physical Systems
Institute for Software Technology
Graz University of Technology
Graz, Austria
nour.chetouane@ist.tugraz.at

Franz Wotawa
CD Laboratory for Quality Assurance Methodologies for Cyber-Physical Systems
Institute for Software Technology
Graz University of Technology
Graz, Austria
wotawa@ist.tugraz.at

***Abstract*—Testing AI-based systems and especially when they rely on machine learning is considered a challenging task. In this paper, we contribute to this challenge considering testing neural networks utilizing mutation testing. A former paper focused on applying mutation testing to the configuration of neural networks leading to the conclusion that mutation testing can be effectively used. In this paper, we discuss a substantially extended empirical evaluation where we considered different test data and the source code of neural network implementations. In particular, we discuss whether a mutated neural network can be distinguished from the original one after learning, only considering a test evaluation. Unfortunately, this is rarely the case leading to a low mutation score. As a consequence, we see that the testing method, which works well at the configuration level of a neural network, is not sufficient to test neural network libraries requiring substantially more testing effort for assuring quality.**

***Index Terms*—deep neural networks, mutation testing**

I. Introduction

In recent years there have been a growing interest in utilizing methods and techniques originating from artificial intelligence for providing extended functionality. Autonomous driving, where a car itself steers passengers from one place in a town to another, is one example heavily relying on artificial intelligence, e.g., for analyzing images and other sensor data in order to detect obstacles. Such image recognition tasks often make use of machine learning techniques like deep neural networks. Hence, there seems to be a tendency to embed artificial intelligence methods and techniques into other systems including safety-critical systems like cars. This immediately raises the question whether ordinary quality assurance methodologies that have been developed and optimized for systems without artificial intelligence can still be used.

In this paper, we partially tackle this question focusing on the applicability of mutation testing in the context of neural networks. Mutation testing [1], [2] was originally introduced for evaluating test suites, which is a very much important task of software and system testing to assure quality. In the context of testing, we ask our self when to stop testing. One way of answering this question is to come up with a test suite that fulfills certain criteria. In mutation testing we change the original program, for example, via modifying constants, changing variable accesses or operators, and check whether the given test suite still is able to detect the introduced fault in the new mutated program. If a test set can detect all considered mutation, we may stop testing.

It is worth noting that we focus on the use and applicability of available mutation testing tools for certain programming languages like Java and do not consider specialized mutation tools for neural networks. There are three reasons behind this decision. First, neural networks and other methods and tools from artificial intelligence are embedded into more traditional systems and software. Hence, when testing such systems and software as a whole, we have to rely on available tools. Second, software engineering tools are more and more closely integrated into the development process, which make them easy to use. Finally, we have been interested in checking whether evaluating a neural network making use of test data is sufficient from the perspective of software engineering.

Therefore, we carried out an experimental evaluation considering different neural network implementations and utilizing an available mutation testing tool for checking whether ordinary mutation testing (i.e., mutation testing not adopted for neural networks) can be used for obtaining valuable information about the degree of testing those neural network implementations. In particular, we make use of an evaluation approach of neural networks where we make use of available test data for computing a mutation score, i.e., the fraction of detected mutations and all mutations introduced in the neural network library implementation. In addition, we investigate on the influence of the size of the test data to the mutation score.

Hence, the contributions of this paper are as follows:

- Introducing a way of making use of mutation testing in the context of neural networks considering the evaluation of neural networks.
- Discussing an experimental evaluation answering the question whether mutation testing of neural network implementations relying on test data used for evaluation is sufficient to assure the correctness of these implementations.

978-1-7281-8913-0/20/$31.00 ©2020 IEEE
DOI 10.1109/QRS51102.2020.00054

- Answering the question, whether there is an impact of the size of the test data when using mutation testing of neural networks.

We organized this paper as follows: First we discuss the methodology behind our experimental evaluation in Section II. Afterwards, in Section III we introduce and discuss the first experimental evaluation considering different neural network implementations and training and test data. In Section IV, we discuss the second evaluation aiming at clarifying the influence of the size of the test data to the mutation score making use of randomly generated data. Finally, we review related research in Section V, and summarize the content of this paper in Section VI.

II. Methodology

Mutation testing [1], [2] in traditional software testing is used to evaluate test suite quality regarding its capacity of revealing possible faults in the program code. Basically, mutation testing consists of inserting various faults by making different types of syntactical changes on the original program. This leads to modified versions of the program, which are called mutants. In case, a test executed on a mutant shows different results than expected, this mutant has been detected and is considered as killed by the test case. Otherwise, if there is no test case that kills the mutation, it is considered to survive.

Eventually, a mutation score is computed that is the ratio of number of killed mutants with respect to the total number of created mutants and indicates how good the underlying test suite is. It is worth noting that there are mutants that can never be killed, i.e., mutants that do not change the behavior of a program. Such mutants are called equivalent mutants, and as a consequence, the mutation score can hardly be 1 meaning that all mutants are killed. In practice, tools try to avoid generating equivalent mutants such that the mutation score reflects the quality of the available test suite for finding faults in the given program.

In a previous study [3], authors already tried to apply mutation testing in the context of deep learning systems using a traditional mutation tool. There, empirical results showed that mutation testing can be applicable for testing different implementations of neural networks by measuring the difference of prediction accuracy of mutated networks against the accuracy of the original network. The authors also reported that certain mutation operators provided by the tool have a significant impact on the network configuration like weights, number of nodes and learning rate. These mutations led to a reduced learning capability of the mutated networks compared to the original one and, therefore, were able to be killed. However, in the previous study the focus was solely on the configuration of neural networks but not on the library implementing the neural network, i.e., the classes required for handling neurons and their interconnections as well as the training of the networks.

Hence, there is a need for further investigating in answering the question whether it is possible to use the same tool to create mutations at a deeper level surpassing the network configurations level and apply mutation testing on classes, which are called during the network training process. In order to do this, we make use of an ordinary mutation tool, i.e., PIT [12], for generating mutations at the class and method level of the considered neural network libraries. Moreover, we assume that we still have training data for allowing a specific neural network learning a certain tasks like classification, and test data for evaluating a network. Note that the purpose of the evaluation (which is also called testing in the context of neural networks) is to show the degree of fulfillment performing a trained task and not to prove whether the library implementation is faulty. What we now need to make use of mutation tools is to find a way of classifying mutated implementations as being killed or not taking care of available information.

Algorithm 1 Test Oracle

Input: OriginalModels
Output: MutationScore μ.

1: Let $MinAcc_{orig}$ be the minimum test accuracy out of the trained original model.
2: Let $MutantTestAcc$ be the test accuracy of one mutant.
3: Let s be the total number of created mutants by PIT.
4: Let r be the number of killed mutants.
5: Let $MinAcc_{orig} = ComputeMinAccuracy(OriginalModel)$
6: Create mutants by running PIT on defined classes .
7: **while** new mutant **do**
8: $TrainMutant()$
9: Let $MutantTestAcc$ be $EvaluateAccuracy()$
10: **if** $MutantTestAcc \geq MinAcc_{orig}$ **then**
11: Mutant survives.
12: **else**
13: Mutant is killed.
14: Let r be $r + 1$
15: **end if**
16: **end while**
17: $\mu = r/s$
18: **return** μ

For classifying mutated networks as being killed or not, we use the idea from [3] and come up with a test oracle. The underlying idea is to compare the outcome of the evaluation of a mutated network with the one of the original neural network using the available test data. In case of substantial deviations, the mutated neural network, i.e., the mutant, is said to be killed. Otherwise, the mutated network is said to survive. Algorithm 1 defines our underlying test oracle in a more formal way. In the test oracle algorithm, we start with the computation of the minimum prediction accuracy in Line 5. We do this by evaluating the original network 20 times, and afterwards defining its minimum prediction accuracy as our threshold for the later created mutated neural networks. It is worth noting that we decided to run the original network 20 times due to the fact that also the original model when trained multiple times can have different prediction accuracies.

Afterwards, in Line 6 we use the mutation tool PIT, which we explain in more detail in the next section, to generate mutants. In every iteration, PIT applies a different mutation operator accordingly to our selected mutation types as given in Table III. In lines 7 to 16 we do the following steps for

each created mutant: (i) We train the mutant on the same training set that we used for training the original network. (ii) We evaluate the trained mutant on the available test set and compare its achieved prediction accuracy to the previously obtained minimum accuracy. A mutant is considered as killed if its accuracy is not equal or above the defined margin. To illustrate this, let us assume that we computed a minimum accuracy of the original neural networks of 97%. A mutant is killed if it has less than 97% prediction accuracy on the test set. Thus, created mutants that are not able to learn due to the applied mutation operators get killed, because either they have no prediction capability at all or a very limited one. Note that when applying this test oracle also to other networks than the presented ones, we may have to adapt the deviation boundary to suit the underlying network.

III. Experimental evaluation

In this section, we give an overview of the used datasets and the evaluated deep learning models. This is followed by a short explanation of the functions used within the neural networks, which were of interest for mutation testing. Afterwards, we will explain our experimental setup in detail. At the end we present and discuss the obtained experimental results of our evaluation. In particular, we answer the research question **RQ1** whether it is sufficient to make use of available test data in the evaluation process of neural networks to assure reaching a high mutation score.

A. Datasets

We performed our empirical evaluation using four different datasets. Two of them, MNIST [4] and CIFAR-10 [5], are publicly available and often used to evaluate convolutional neural networks for image classification. MNIST is a dataset of handwritten digits categorized in 10 classes (i.e. digit 0 to 9). It has a training set containing 60,000 examples, and a test set of 10,000 examples. The CIFAR-10 dataset comprises 32x32 pixel color images that are divided into ten classes (i.e. airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck). CIFAR-10 contains 50,000 training images and 10,000 test images with 6,000 images per class. The two other datasets used are numerical examples representing X and Y coordinates. Each one of them contains 5,000 training examples and 1,000 test examples. The first dataset is used for a two-class "Saturn" classification problem where the first class refers to the planet and the second one represents the ring. In the second numerical dataset the examples are classified into four different classes (i.e. class 1, class 2, class 3, and class 4).

B. Deep learning models

For the experimental part, we have selected three Multi-Layer Perceptron (MLP) networks, namely MLP Classifier Saturn, MLP Classifier X, MLP MNIST, and two convolutional networks that we called Convolutional MNIST and Convolutional CIFAR. The MLP Classifier Saturn, MLP MNIST, convolutional MNIST and Convolutional CIFAR are available in [6] whereas the MLP Classifier X was implemented by our own. In Table I and II we show the structures and configurations of the different networks used in this study.

The first multilayer perceptron network, MLP Classifier Saturn, is composed of two layers both containing 20 nodes. It receives two inputs through a first dense layer that uses a Rectified Linear Unit (ReLU) activation function. The second layer applies a Softmax activation function and leads to two output classes. The MLP Classifier Saturn is trained on the first numerical dataset classifying two classes and achieves 99.16% test accuracy. The second, MLP Classifier X, contains three dense layers composed respectively out of 10, 15, 15 nodes and all are using ReLu activation functions. The output layer is composed of four nodes representing the four different output classes. We train the MLP Classifier X on the second numerical dataset containing four different classes where it accomplishes a test accuracy of 97.35%.

The third multilayer perceptron, MLP MNIST, is composed of two dense layers having respectively 500 and 100 nodes and both applying ReLu activation. The output layer contains 10 nodes corresponding to the 10 handwritten digits. MLP MNIST has a 97.67% test accuracy. Regarding the two convolutional networks, the first one is also trained on MNIST and attains 98.82% of test accuracy. It is composed of four convolutional layers and two subsampling layers applying max pooling with a kernel size of [2,2] followed by two dense layers that uses ReLu activation function and an output layer comprising 10 output nodes. The last model is a big convolution network composed of seven convolutional layers using LEAKY Relu activation, seven Batch Normalization layers and three Subsampling layers with a [2,2] kernel size, two of them use max pooling and the third one applies AVG pooling technique. The output layer holds 10 output nodes. This model has a test accuracy of 70.51%.

In the following, we explain some basic functions and algorithms employed during the training of our models and that have been mutated in the experiments.

1) Loss function and weight initialization: When training a model, the objective is to find the minima of a loss function that is used to estimate the error given a set of parameters (i.e. weights). The main rule of the optimization process is that the model gets better as the loss functions is approaching 0. The loss function used in our neural networks is the negative log likelihood. When the network assigns high probability to the correct class, the negative log applied to this probability value gets low, but when the network assigns low probability to the correct class, the negative log loss gets high. Basically, the goal is to maximize the model performances by minimizing the negative log likelihood.

All networks used in this study apply an Xavier scheme for weight Initialization. This method is commonly used in order to avoid exploding gradients problem, which occurs when the input signal starts to drop to a really low value and stops spreading through the networks layers. Xavier initialization scheme ensures that the variance of the outputs of each layers stays equal to the variance of its inputs. It assigns the weights

using a Gaussian distribution with zero mean and a variance equal to $1/n$, where n represents the number of input neurons (see [7]).

2) Gradient descent optimization algorithms: Gradient descent is an iterative algorithm, which mainly forms the basis of the learning process of neural networks. It randomly picks a starting initial point on a function and keeps iterating in steps, descending its slope, until it reaches the lowest point (i.e. minimum value) of that function [8]. As shown in Table II the Convolutional CIFAR network is trained using Stochastic Gradient Descent (SGD). It is one of the popular optimization algorithm commonly used in different machine learning algorithms as it is known to be computationally friendly especially for models having high number of parameters such as neural networks. It also applies Adadelta [9] updater, which optimizes the learning process by stocking past gradient descent parameters. However, it limits their storing to some fixed size by recursively defining the average of all previous squared gradients instead of inefficiently accumulating all of them.

The MLP Classifier Saturn and Convolutional MNIST network employ a more optimized version of gradient descent called Nesterov Accelerated Gradient (NAG) [10], which speeds up the learning process comparing to plain Stochastic Gradient Descent. It provides faster convergence by taking into accounts previous gradients and by introducing a velocity component in the update rule at each iteration, thus giving an approximation of the next position of the parameters [8]. The MLP MNIST uses another Gradient decent optimization algorithm named Nadam (Nesterov-accelerated adaptive moment estimation). It is a combination of NAG, which performs a more accurate step in the gradient direction and Adaptive Moment Estimation (Adam) [11], which is an adaptive learning rate based method that adjusts the learning rate to the parameters according to their frequency. It makes larger updates for infrequent parameters and smaller updates for frequent ones.

3) Activation functions: In deep systems, activation functions are crucial components for the network learning. They are mathematical equations that are responsible of determining the output of the neural network, its accuracy, and the training computational efficiency. Their main role is to normalize the output of each neuron to a range between 1 and 0 or between -1 and 1. Rectified Linear Unit (ReLU) is a default activation function for many types of neural networks because it is computationally efficient and allows the network to converge very quickly. It outputs the input directly if it is positive, otherwise, it will output zero. Leaky ReLU is a variation of ReLU that helps preventing the gradient of a function from approaching zero as it has a small positive slope in the negative area so it does allow back propagation even for negative input values. The softmax activation function is often employed at the output layer of a neural network. It's commonly used for multi-class learning problems where a set of features can be related to one of k classes. Intuitively, the softmax basically compresses a vector of size k between 0 and 1. We can then take the output of the Softmax as the probabilities that a certain set of features belongs to a certain class.

4) Neural network layers: Deep neural networks are principally composed of an input layer that receives the input signal corresponding to a set of input features, an output layer that makes a decision or prediction about the given input, and in the middle, an arbitrary number of hidden or dense layers that perform different kinds of transformations and form the basic computational engine of the MLP.

In addition to the input, output layers and multiple of regular hidden layers, the convolutional neural network (CNN) architectures contain other specific layers including convolutional, pooling layers and batch normalization layers. The convolutional layer represents the principal component of a CNN architecture. It's composed of a set of learnable filters. The main role of this layer is to slide the filter through input images and computes the scalar product. The output of the scalar product indicates whether the pixel pattern in the input image matches the pixel pattern represented by the filter. Generally, in a CNN architecture, subsampling or pooling layers are periodically inserted between successive convolutional layers. They are used to progressively reduce the spatial dimension of the representation. There are several nonlinear functions to implement pooling such as max pooling and average pooling. The pooling layer partitions the input image into a set of non-overlapping rectangles and, for each such sub-region, outputs the maximum or the average. Batch normalization layers help to accelerate the training process of the network by adjusting and scaling the inputs to activation functions. It normalizes the inputs to a layer at each batch by maintaining the mean activation close to 0 and the activation standard deviation close to 1.

C. Experimental setup

We now introduce the experimental setup of our empirical evaluation. At first, we provide a short overview of the deep learning framework and the mutation tool. This is followed by a description of the different mutation operators used in our experiments. In the last part of this section we present and discuss the obtained results.

1) Framework and tools: Similarly to the previous study [3], we make use of the Deeplearning4j framework [6] for implementing our neural networks and performing mutation testing on them. This framework is an open source library for Java Virtual Machines (JVM), which offers a variety of deep learning model implementations that could be used for different experiments.

As a mutation testing tool, we selected PIT [12]. PIT is an open source mutation tool that is considered to be fast as it runs directly on byte code instead of source code level. During the mutation process, equivalent mutants, which transform the original program to a behavioral equivalent one, may be created, nevertheless PIT offers a quite stable version, as it avoids the creation of equivalent mutants when only the default mutators are activated. Therefore, in order to avoid equivalent mutants, we selected only the default mutators as recommended. Yet, we decided to activate an additional

TABLE I
MODEL STRUCTURES AND CONFIGURATIONS OF MLP CLASSIFIER SATURN, MLP CLASSIFIER X AND MLP MNIST.

Configurations	MLP Classifier Saturn	MLP Classifier X	MLP MNIST
Weight Initialization	XAVIER	XAVIER	XAVIER
Loss Function	NegativeLogLikelihood	NegativeLogLikelihood	NegativeLogLikelihood
Updater	Nesterovs	Nesterovs	Nadam
Layers + Activation functions	Dense (2, 20)+ ReLu	Dense (2, 10)+ ReLu	Dense (784, 500)+ ReLu
	OutputLayer(20, 2)+ Softmax	Dense(10, 15) + ReLu	Dense(500, 100) + ReLu
		Dense(15, 15) + ReLu	OutputLayer(100, 10)+ Softmax
		OutputLayer(15, 4)+ Softmax	
Testing Accuracy	99.16 %	97.35%	97.67 %

TABLE II
MODEL STRUCTURES AND CONFIGURATIONS OF THE CONVOLUTIONAL MNIST AND CONVOLUTIONAL CIFAR NETWORKS.

Configurations	Convolutional MNIST	Convolutional CIFAR
Weight Initialization	XAVIER	XAVIER
Loss Function	NegativeLogLikelihood	NegativeLogLikelihood
Updater	Nesterovs	AdaDelta
Optimizer		Stochastic Gradient descent
Layers + Activation functions	Conv(1, 16) + ReLU	Conv(3, 32) +LEAKYReLU
	Conv(16, 32) + ReLU	BatchNormalization(32, 32)
	Subsampling(MaxPooling(2,2))	Subsampling(MaxPooling(2,2))
	Conv(32, 64) + ReLU	Conv(32, 16)+LEAKYReLU
	Conv(64, 64) + ReLU	BatchNormalization(16, 16)
	Subsampling(MaxPooling(2,2))	Conv(16, 64)+LEAKYReLU
	Dense (1024,100)+ReLU	BatchNormalization(64, 64)
	Dense (100, 200)+ReLU	Subsampling(MaxPooling(2,2))
	OutputLayer(200, 10)+Softmax	Conv(64, 32)+LEAKYReLU
		BatchNormalization(32, 32)
		Conv(32, 128)+LEAKYReLU
		BatchNormalization(128, 128)
		Conv(128, 64)+LEAKYReLU
		BatchNormalization(64, 64)
		Conv(64, 10)+LEAKYReLU
		BatchNormalization(10, 10)
		Subsampling(AVG(2,2))
		OutputLayer(490, 10)+Softmax
Testing Accuracy	98.82 %	70.51 %

mutator, namely the inline constant mutator, since we have noticed from the previous study that this mutation operator has a significant impact on our network implementations as it applies changes on the network configuration parameters such as initial weights, learning rate, schedulers, and also optimization algorithms used during the network learning process.

To ensure the reproducibility of our results we worked with stable versions of both tools particularly version 1.0.0-beta6 for the Deeplearning4j framework and 1.5.0 for the PIT mutation tool. The integration of both tools is usually done via build tools like Maven, which downloads already compiled dependencies of the tools that can be imported in an existing project. However, this was not possible for our evaluation, since the main goal is to mutate specific classes, mentioned in the previous section. Therefore, two options were available for applying mutations on the specified classes. First one is that Deeplearning4j can be built completely from source, but as stated by the developers, this approach should be avoided since it is quite complicated and very prone to errors during the build process. The second option is to make a workaround in order to avoid the complete build from source process but still ensure the possibility to mutate specific class functions. Therefore we used the Deeplearning4j examples, a repository containing several different neural network application types, which can be built with the Maven build tool. Later on, to perform mutation testing on the selected classes, these files were included with their dependencies as source files within our project. With this workaround it was possible to extend the previous approach to a deeper level surpassing only the mutation on neural network configuration level.

2) PIT mutation operators: In the previous paper [3], authors provided a table depicting examples of available mutation operators. In Table III, we present an updated version of the new default mutators introduced by PIT. As mentioned before, we try to limit the creation of equivalent mutants by activating only the default mutation operators and the inline constant mutator. As it is stated in Table III, to avoid equivalent mutants, some additional investigations are made by the PIT tool before creating a mutant. For instance, the primitive returns mutator replaces the original value just in case it is not hard coded already to 0 and therefore no equivalent mutant is

created here.

Besides taking the measure to just activate the default mutators and the inline constant mutator, we defined additional configuration parameters where we restrict the PIT tool of mutating functions like printing, logging or similar. Also some functions that were not covered by the executed tests were excluded as well. However, we were not able to apply these restrictions for all classes due to the fact that in some cases a huge amount of functions are implemented within one class and it would have not been feasible to exclude all these functions manually. To the best of our knowledge, it is still not possible to create mutants only in functions covered by the implemented tests within PIT.

D. Experimental results

In Table IV we present the results for the three multilayer perceptron networks. For each network, we have applied mutations on ten classes. A total number of 1,526 mutants were created for the MLP Classifier Saturn from which only 202 were able to get killed by the test oracle. For the MLP classifier X, the tool created 1,526 mutants in total and killed only 208. The MLP MNIST example resulted in 1,536 created mutations and a total of 219 killed ones. We have noticed that despite the larger size of MLP Classifier X and MLP MNIST networks in terms of number of nodes, there was not a significant increase in the number of created mutants compared to the MLP Classifier Saturn. The reason behind this could be that the three networks use basically the same layers and functions during learning. This also explains the same mutation score obtained for most of the mutated classes. We only remark a small difference for the updater class and the loss function class where a higher mutation score is achieved at the MLP MNIST network.

Table V shows the results for the more complex convolutional neural networks. Here 13 classes were selected for mutation on both types of networks. For the convolutional network evaluated on the MNIST dataset a total of 1,604 mutants were created and 262 of them were killed. On the other hand, 1,600 mutants were created for the convolutional CIFAR network where 305 were able to get killed. As for the MLP networks it can be seen that although the convolutional CIFAR network is much bigger in terms of used layers and activation functions a similar amount of mutants was created during the training process. The number of killed mutants is slightly higher for the convolutional CIFAR model. This may be the case because a significant higher score was achieved within the LEAKY ReLu activation function and AdaDelta updater class. Also within the batch normalization class, which is not used by the convolutional MNIST, 38% of the mutants got killed. Nevertheless with respect to the overall score no significant difference between the two convolutional networks was recognized.

E. Discussion

In comparison to the results of the previous paper [3], we see a significant decrease of the achieved mutation score when applying mutation testing on a deeper class level than only the configuration level of neural network implementations. This can be accredited to mainly two reasons:

The first explanation for this is the importance of certain mutation operators to training results. From the previous study we obtain that the inline constant mutator has a huge impact on the network parameters that resulted in a significant increase on the general mutation score. However, the same effect is not the case when mutations were applied on other specific library classes, since there are only a few number of constants defined within these more specific mutated classes. Often, these functions receive the constant network parameters as input directly from the configuration level. Thus, within the deeper class functions most of the defined parameters on configuration level are handled as variables where no mutation operator is capable of producing an influential mutation.

Second, we noticed that the PIT tool creates many mutants inside several functions that are not called by the the mutated network or which are within branches that are never reached during training because of for instance an if-else-statement where the precondition is not satisfied. This can lead to a high number of created mutants within big classes that contain numerous functions and that are not covered by our tests and therefore are considered as survived mutants. As an example we noticed that for instance within one class around 1,000 mutations were created but since this class contains a lot of different functions and helper functions that may only be called for other network types than investigated in this paper we see a very low mutation score there.

These observations lead to the conclusion that in order to perform a complete and detailed evaluation of all different learning functions used by the network, it is necessary to create more specific test methods that assure covering the implemented source code of neural network libraries. The evaluation of neural networks using the test data alone seems to be not sufficient considering the low mutation score obtained. Hence, research question **RQ1** can be answered negatively, i.e., it is not sufficient to make use of available test data to assure reaching a high mutation score. However, what is still open is answering the question, whether a larger test dataset would lead to a higher mutation score. We are going to answer this question in the next section.

IV. Random test set generation and evaluation

In this section, we use random test generation for investigating whether we can improve the mutation score when increasing the size of the test set. Hence, we want to answer the second research question **RQ2**, i.e., whether the size of a test set has an impact on the mutation score. To answer this question, we randomly generated new test sets while extending the size of the test set at each iteration. In the next subsections we give an overview of the experimental setup and present the obtained results followed by a short discussion on the obtained results.

TABLE III
SOME EXAMPLES OF THE AVAILABLE MUTATION OPERATORS PROVIDED BY PIT.

Mutator type	Description
Conditionals Boundary Mutator	relational operators are changed to their boundary counterpart (e.g. $<$ to $\leq$, $>$ to $\geq$).
Empty Returns Mutator	return values are replaced by an empty value of the same type (e.g. $java.lang.String$ to "").
False Returns Mutator	primitive and boxed boolean return values are replaced with false.
Increments Mutator	replaces increments with decrements of local variables and vice versa.
Inline Constant Mutator	changes inline constants (e.g. $true$ to $false$, 1 to 0, -1 to 1, increment by 1).
Invert Negatives Mutator	inverts negation of integer and floating point numbers.
Math Mutator	replaces binary arithmetic operations for either integers or floats with another operation (e.g. $+$ to $-$, $*$ to $/$, & to |).
Negate Conditionals Mutator	mutates all found conditionals (e.g. $==$ to $!=$, $\leq$ to $>$, $\geq$ to $<$, $<$ to $\geq$, $>$ to $\leq$).
Null Returns Mutator	return values are replace with $null$. If the method can be mutated by the empty returns mutator it is not mutated by this mutation operator.
Primitive Returns Mutator	$int, short, long, char, float$ and $double$ return values are replaced with 0.
True Returns Mutator	primitive and boxed boolean return values are replaced with true.
Void Method Call Mutator	method calls to void methods are removed.

TABLE IV
MUTATION TESTING RESULTS OF THE MLP CLASSIFIER SATURN , MLP CLASSIFIER X AND THE MLP MNIST USING PIT.

Mutated Classes	Mutation Score		
	MLP Classifier Saturn	MLP Classifier X	MLP MNIST
Dense Layer	38%	38%	25%
Output Layer	50%	50%	50%
Weights(Init Xavier)	25%	75%	50%
Activation function: ReLu	100%	100%	20%
Activation function: Softmax	100%	100%	100%
Updater	Nestrovs: 22%	Nestrovs: 28%	Nadam: 47%
Loss function (Negative Log Likelihood)	24%	23%	24%
Total number of classes	10	10	10
Total number of created mutations	1,526	1,526	1,536
Total number of killed mutants	202	208	219

A. Experimental setup

For the random test set generation experiments, we have decided to use the MLP Classifier Saturn network and the MLP Classifier X. The reason behind this selection is that it is rather easy to generate random numerical test sets for MLPs compared to convolutional MNIST and convolutional CIFAR, which use images as an inputs. For these convolutional networks it would have take a huge effort to collect a big amount of new labeled test sets, which we may not be able to generated automatically.

For each of the two MLP classifiers, we created in total five additional random test sets to extend the mutation testing evaluation done in the section above. To see if it is possible to increase the mutation score by bigger test sets we decided to increment the size of the test set at each evaluation step. The five randomly generated test sets include respectively 2,000, 4,000, 6,000, 8,000, and 10,000 test examples.

After creating the test sets, we applied the same steps as before given in Algorithm 1. We trained and evaluated the original network 20 times without applying mutations for identifying the threshold that defines if a mutant is killed or not. It is worth noting that, for setting our comparison baseline, we trained and tested the original networks on the same training set and test set from the first part of our experimental evaluation, which comprise respectively 5,000 training examples and 1,000 test examples.

Afterwards, for each network, we performed the same procedure of mutation testing five times. At each iteration, we train the mutated network on the same training set as the original model, then, we evaluate its test accuracy using the respective random generated test sets, with increasing their size from 2,000 to 10,000 test examples at each different run.

B. Experimental results

We now examine the outcome of the random test set generation experiment. In Fig. 1 and Fig. 2, we see two graphs: one for the MLP Classifier Saturn and the other for the MLP Classifier X. The graphs illustrate the changes occurring on the mutated networks from test accuracy perspective when evaluated on the five different randomly generated test sets. The graphs also show our evaluation baseline (i.e. highlighted in blue) that represents the mean accuracy of the original networks.

TABLE V
MUTATION TESTING RESULTS OF THE CONVOLUTIONAL MNIST AND THE CONVOLUTIONAL CIFAR NETWORKS USING PIT.

Mutated Classes	**Mutation Score**	
	Convolutional MNIST	Convolutional CIFAR
Dense Layer	25%	-
BatchNormalization	-	38%
Output Layer	50%	50%
Convolutional Layer	54%	50%
Subsampling Layer	45%	50%
Weights(Init Xavier)	100%	50%
Optimizer (Stochastic Gradient Descent)	-	30%
Activation function	ReLu: 20%	LEAKYReLu: 50%
Activation function: Softmax	100%	100%
Updater	Nestrovs: 33%	AdaDelta: 56%
Loss function ((Negative Log Likelihood)	23%	23%
Learning Schedule	65%	-
Total number of classes	13	13
Total number of created mutations	1,604	1,600
Total number of killed mutants	262	305

For both networks we observed a significant drop of test accuracy for some mutants (i.e. around 50% for the MLP Classifier Saturn and around 28% for the MLP Classifier X) and on the other hand a quite stable accuracy level nearly the same as the accuracy of the original model (i.e. around 99% for MLP Classifier Saturn and 97% for MLP Classifier X) as some mutations do not significantly effect the learning ability of the network. We noticed these two observations similarly for all the five random generated test sets, for both, the MLP Classifier Saturn and MLP Classifier X. Note that, not all accuracy values of all created mutants are outlined in the graphs since some of them are not even computed due to the fact that the mutated network was not even able to learn at all because for instance of a removed method call such as the training method or the evaluation method.

Regarding the mutation score, we do not see any change for the five random test sets. We obtained a total mutation score of 13% for MLP Classifier Saturn and 14% for MLP Classifier X that stayed the same over all five random test set evaluations. A reason for this may be that for applied mutations on the network, the accuracy either drops significantly beneath the defined threshold or is still capable of training and therefor achieves similar accuracies than the original model. We noticed no accuracy that is just slightly beneath the threshold and therefore got killed. An explanation for this may also be the behavior of the PIT tool and how it mutates specific functions or constants. For instance a constant that is initially set to 0.01 will be mutated to 1, which would be a big difference for example for a learning rate constant used by a neural network and will in most of the cases lead to a significant accuracy drop.

C. Discussion

From the obtained results we conclude that there is no impact of the underlying test dataset size on the mutation score for neural network libraries and there in particular MLPs.

Because of similarities between the underlying algorithms of different neural network implementation, we believe that this results is also valid in general but there is still an extended experimental evaluation required. Considering MLPs we can answer **RQ2** also in a negative way. There seems to be no influence on the test dataset size on the mutation score. From our point of view, and based on our observations of the results obtained after performing the above experiments, this can be explained by the type of mutations created by the tool. Some mutations are more effective than others and lead for instance to a stop of the network learning process, for example changing the return value of the activation function to null. In this case, the size of the test dataset does not matter since the network will not be capable of training and the accuracy on either a small or a large test set will be in the same accuracy range and therefore no change on mutation score will be observed. Similarly, other types of mutations have no significant influence as they are created for instance in functions or branches that are not executed during the training process of the neural network. For instance, several Nestrovs updaters are defined within the updater class that are not even called during the learning phase of the network. We noticed that these types of mutations do not cause a drop in the network accuracy independently from the test dataset size, therefore, increasing the size of the test set or decreasing it will not influence the obtained mutation score. This is caused by the fact that the tool does not allow eliminating mutations created on specific uncalled functions within classes. It is worth mentioning that these interpretations are based on the results achieved when training the two models with these specific numerical datasets and that there might be additional explanations related to other types of data such as in case of image classification. Therefore, as mentioned before, further investigations should be conducted using other network implementations and different datasets. This result is a further indicator for stating that a pure neural network evaluation is not sufficient for testing a neural network implementation considering ordinary software testing measures like the mutation score.

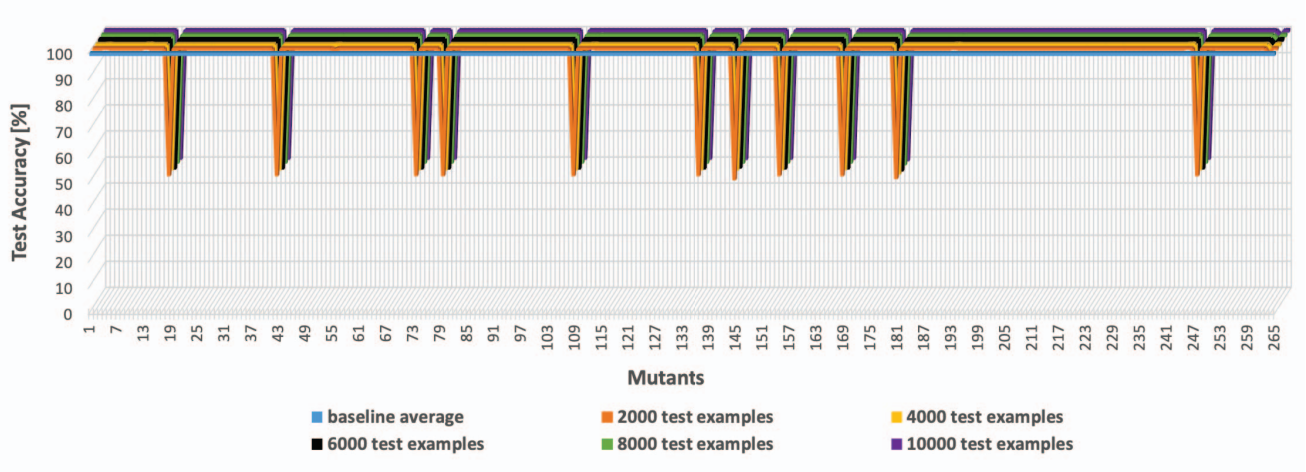

Fig. 1. Test accuracies of the mutated MLP Classifier Saturn.

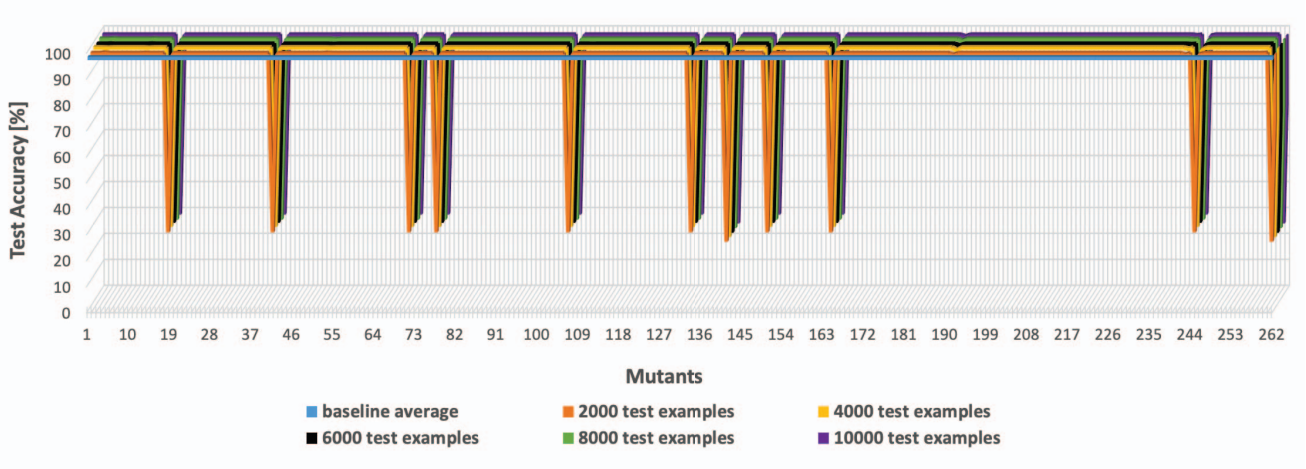

Fig. 2. Test accuracies of the mutated MLP Classifier X.

V. Related work

Testing deep learning systems has been recently of interest for many researchers. Several authors have initially started by focusing on testing neural networks while being inspired by different traditional test coverage criteria like Modified Condition/Decision Coverage (MC/DC) coverage, and tried to personalize them to the distinct features of deep neural networks systems such as in [13]. Others have tried to create more specific coverage metrics such as neural coverage in [14]. In [15], they have also adapted a set of combinatorial testing criteria based on the neuron input interaction.

Initial attempts of applying mutation testing for deep neural networks, were first made by [16] where they study two main research questions. The first one is about the mutation effect on neural networks and the second one deals with the impact of the network depth on mutation. The authors do not focus on the syntax rules but more on the characteristics of the neural network. They propose a mutation analysis method called MuNN, which basically contains five types of mutations operators that delete neurons in input and hidden layers, change bias, weight and activation functions. The experimental results show that the network depth is a sensitive factor in mutation analysis as the mutation effects are gradually weakened with the deepening of neurons. The authors also, noticed that mutation analysis of neural networks can have strong domain characteristics, which can support researchers in better understanding the works of neural network.

Another mutation tool for artificial neural networks named DeepMutation was proposed in [17]. Here, they focus on both source level and model level. The former category consists in two types, either applying changes on the original training data such as duplicating data, erroneous labels, deleting part of data and adding noise perturbation or modifying the model structure before training, like deleting or adding layers

and removing activation function. The model level mutation operators change the weights, biases or structure of an already trained model.

Recently, in [18], the authors have performed a thorough empirical study to investigate all different mutation operators provided in the existing deep learning mutation tools mentioned previously. They performed a very detailed analysis in order to identify the most effective mutation operators together with the associated configurations that eliminate equivalent and trivial mutants. Besides, they proposed a new mutation killing definition as alternative for the accuracy threshold-based definition used in previous studies since the latter does not take into consideration the stochastic nature of the network training process and could be inconsistent through different runs.

VI. Conclusion

In this paper, we investigated on the applicability of mutation testing for quality assurance of systems comprising neural networks. In particular, we were interested in answering the question whether the testing phase that occurs after training a neural network mainly for evaluation purposes is sufficient for testing the neural network implementation itself. For answering this question, we assumed available test datasets to be sufficient in case they lead to a high mutation score considering the whole neural network library used. We carried out experiments considering MLPs and convolutional neural networks showing that the mutation score is not enough for the purpose of finding a sufficient number of mutants. In addition, we extended the experiments also covering the case of different sizes of test datasets, which also indicates that test data is not sufficient of testing neural networks.

As a result we see that there is a need for coming up with specialized test suites for neural network libraries aiming on increasing the mutation score substantially. It might be of interest to investigate on test case generation that takes particular neural network models into account for generating a small test suite that reaches a certain mutation score. In addition, future research includes extending the evaluation making use of more network implementations and test datasets.

Acknowledgment

The financial support by the Austrian Federal Ministry for Digital and Economic Affairs, the National Foundation for Research, Technology and Development and the Christian Doppler Research Association is gratefully acknowledged.

References

[1] T. Budd, R. DeMillo, R. Lipton, and F. Sayward, "Theoretical and empirical studies on using program mutation to test the functional correctness of programs," in *Proc. Seventh ACM Symp. on Princ. of Prog. Lang. (POPL)*. ACM, Jan. 1980.

[2] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, 2011.

[3] N. Chetouane, L. Klampfl, and F. Wotawa, "Investigating the effectiveness of mutation testing tools in the context of deep neural networks," in *International Work-Conference on Artificial Neural Networks*. Springer, 2019, pp. 766–777.

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

[5] A. Krizhevsky, V. Nair, and G. Hinton, "The cifar-10 dataset," *online: http://www. cs. toronto. edu/kriz/cifar. html*, vol. 55, 2014.

[6] A. Gibson, C. Nicholson, J. Patterson, M. Warrick, A. Black, V. Kokorin, S. Audet, and S. Eraly, "Deeplearning4j: Distributed, open-source deep learning for java and scala on hadoop and spark," January 2016, DOI: 10.6084/M9.FIGSHARE.3362644.V2.

[7] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, 2010, pp. 249–256.

[8] S. Ruder, "An overview of gradient descent optimization algorithms," *arXiv preprint arXiv:1609.04747*, 2016.

[9] M. D. Zeiler, "Adadelta: an adaptive learning rate method," *arXiv preprint arXiv:1212.5701*, 2012.

[10] Y. Nesterov, "A method for unconstrained convex minimization problem with the rate of convergence o (1/kˆ 2)," in *Doklady an ussr*, vol. 269, 1983, pp. 543–547.

[11] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[12] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "Pit: a practical mutation testing tool for java," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 2016, pp. 449–452.

[13] Y. Sun, X. Huang, D. Kroening, J. Sharp, M. Hill, and R. Ashmore, "Testing deep neural networks," *arXiv preprint arXiv:1803.04792*, 2018.

[14] K. Pei, Y. Cao, J. Yang, and S. Jana, "Deepxplore: Automated whitebox testing of deep learning systems," in *proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 1–18.

[15] L. Ma, F. Zhang, M. Xue, B. Li, Y. Liu, J. Zhao, and Y. Wang, "Combinatorial testing for deep learning systems," *arXiv preprint arXiv:1806.07723*, 2018.

[16] 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)*. IEEE, 2018, pp. 108–115.

[17] L. Ma, F. Zhang, J. Sun, M. Xue, B. Li, F. Juefei-Xu, C. Xie, L. Li, Y. Liu, J. Zhao *et al.*, "Deepmutation: Mutation testing of deep learning systems," in *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2018, pp. 100–111.

[18] G. Jahangirova and P. Tonella, "An empirical evaluation of mutation operators for deep learning systems," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2020, pp. 74–84.