Bachelor Thesis

Lennart Mühlhahn

Mutation-Based Accuracy Improvements in Neural Networks using Spectrum-Based Fault Localization

March 6, 2024

supervised by:
Prof. Dr. Sibylle Schupp
Daniel Rashedi

Hamburg University of Technology (TUHH)
Technische Universität Hamburg
Institute for Software Systems
21073 Hamburg

Declaration

I, Lennart Mühlhahn, hereby affirm that the Bachelor's thesis titled "Mutation-Based Accuracy Improvements in Neural Networks using Spectrum-Based Fault Localization" is entirely my own work, completed independently and without unauthorized external help.

I have duly acknowledged and cited all instances of direct quotations and paraphrased content from other authors, ensuring that all such references are properly sourced.

Hamburg, March 6, 2024

Lennart Mühlhahn

Abstract

In this thesis, we introduce an approach to enhance neural networks by mutating neurons identified through spectrum-based fault localization. This method, initially demonstrated in DeepFault, is advanced by mutating weights and biases based on suspiciousness values. The thesis introduces an automated mutation algorithm, and evaluates the approach using the Fashion-MNIST dataset across various model architectures and training dataset sizes. Results show notable accuracy improvements and provide insights into the impact of suspiciousness measures, dataset sizes, and initial training extents. This research contributes to the field by offering a refined technique for neural network accuracy enhancement and opens avenues for future exploration in neural network fault localization and repair strategies.

Contents

List of Figures

Listings

1. Introduction

Deep Neural Networks (DNNs) [27] are increasingly used in today's world and more capable than ever, and are often used in highly sensitive applications domains like medical diagnosis [30] or autonomous vehicles [1]. DNNs are highly advanced in areas like image [26, 7], video [23], or speech recognition [19]. There, the range of applications can range from simple recognition towards end to end learning [1].

Especially because of the use of DNNs in safety-critical domains, it is from the uttermost urgency to ensure the safety of every user and every other person who is encountering these systems. But how can we ensure the proper operation of the neural network, and find bad actors in neural networks and repair these neurons?

The Method we explored in this thesis is the mutation of neurons, which we first identified by using spectrum-based fault localization (SBFL). While SBFL was already first successfully used in Eniser et al.'s DeepFault [11] in conclusion with an input synthesizes guided by the suspiciousness values, elicited by the SBFL, we want to go on to a more in-depth level and mutate the weights and biases based on these suspiciousness values.

For our approach, we first adapted the SBFL provided by DeepFault to today's versions of the TensorFlow framework, and the other used frameworks, afterwards we added the functions for the mutation of the weights and biases based on the ranked neuron locations provided by the SBFL. The mutations either assign a value randomly, a predefined value, scales by a predefined or random value to the bias or weights of a neuron, we can also decapitate a neuron. Then we added an algorithm to perform the mutation automatically by just providing the model, the data-set and the wished parameters. We first perform the SBFL and rank the neuron in our network based on a suspiciousness measure, mutate the node, if wished, we train the model a further epoch. Then we evaluate the model and if our break condition is met we give back the model, else we try to further improve the model by starting the loop with a new neuron again.

To evaluate our Approach, we used the Fashion-MNIST dataset by Xiao et al. [50]. We used 4 model architectures, two Deep Neural Networks (DNN) and two Convolutional Neural Networks (CNN) to run our evaluation. For the training, we haven't only used the full dataset to train the model, but also half and a quarter of the dataset. Another thing that differentiated the models, we also trained them for either one or six epochs, before using the algorithm. We evaluated our approach based on the change of the performance (loss and accuracy) of the network against the baseline network. If we further trained the model, we compared against the further trained reference model, else we compared the change against the initial performance of the model.

There by we elicited an accuracy improvement in some cases up to in 25% for the first three epochs of the algorithm for the not further trained models. For the further trained models, we see an accuracy improvement of up to 7.5%. We see similar improvements in the loss values for the models.

Furthermore, we investigated the impact of the different suspiciousness measures and compared the usefulness against random choosing of the neurons. Another thing we tried to elicit is the effect of the different dataset sizes and the extent of the initial training. Of course, we also tried to find the best configuration of the parameters for our Algorithm. Exemplary of this is if the usage of an offset for the accuracy and loss, which leads to better results. For that, we also saw the need to evaluate the different break conditions for the algorithm. And at last, we evaluated our different mutation functions and the assigned values.

To continue this Thesis, we will first give you some insights in to the techniques that are at the heart of our approach, neural networks, its testing and spectrum-based fault localization. Then we give you a look at some other neural network repair methods and the first spectrum-based fault localization method for neural networks, DeepFault [11]. Afterwards, we will provide a deep dive in to our approach until we present you our findings and an outlook to future research based on our work.

2. Background

In this chapter, we are going to present the background information that is necessary for our approach. What are neural networks? How can we test them, and what is spectrum-based fault localization?

2.1. Neural Networks

Without further ado, we will discuss what neural networks are and how they are composed. We describe what a layer is and what types there are. For what do we need optimizers and activation functions and what types are there. Neural Networks are useful in a multitude of fields like classification, regression, transcription or machine translation [16]. Neuronal networks are composed of layers, which are composed of neurons. The neuron is the computational component in a deep neural network.
A neuron transforms a vector of inputs $\mathbf{x}$ to the output y, $\mathbb{R}^n \rightarrow \mathbb{R}$ by using a vector of weights $\mathbf{w}$, a bias b and an activation function f as seen in equation 2.1.

$$y = f\left(\sum_{i=1}^{n} w_i \cdot x_i + b\right) \tag{2.1}$$

For example the machine learning software library TensorFlow, which is based on Keras, uses by default Glorot uniform initializer [43, 14], for the weights and initializes the bias with zeros.
Another important component, are the activation functions, one of the most common ones is the sigmoid function, which is today mostly used in output layers. The rectified linear unit, short ReLU [12, 15], as seen in Function 2.2, enables a better training for a neural network compared to the formerly common sigmoid function.

$$f(x) = \begin{cases} x & x > 0 \\ 0 & x <= 0 \end{cases} \tag{2.2}$$

Another, even more promising activation function is the Gaussian Error Linear Unit, short GELU, seen in function 2.2, which was proposed by Hendrycks and Gimpel in 2016 [18] is used by natural language processing models like BERT [9]. It does not just set all negative values to zero like ReLU but weights them. The differences can be seen in the figure. 2.1

$$f(x) = 0.5x\left(1 + \tanh\left[\sqrt{\frac{2}{\pi}}\left(x + 0.044715x^3\right)\right]\right) \tag{2.3}$$

The neurons are then combined into layers, in a Deep Neural Network (DNN) we have three types of Layers: Input layers, output layers and the hidden layers. There is one input and one output layer, but there is a plurality of hidden layers, as you can see in the figure 2.2. The input layer, processes the input for the hidden layers, for example, it

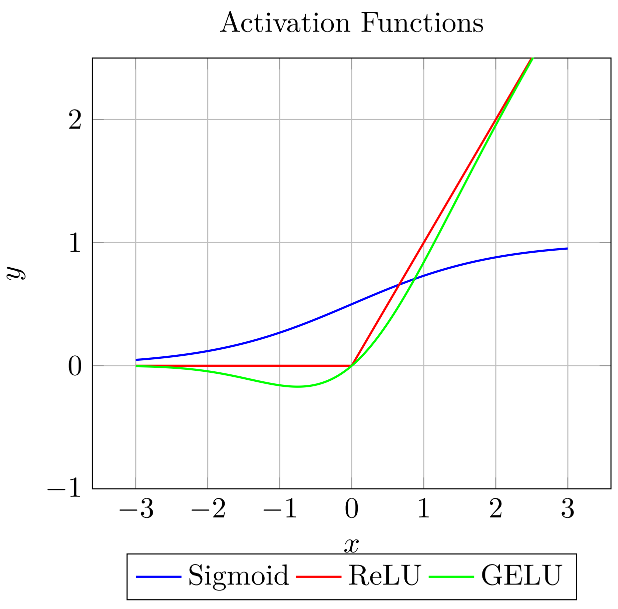


Figure 2.1.: Activation Functions

converts a 2-dimensional picture to a 1-dimensional array, which can then be processed by the subsequent hidden layers. The hidden layer performs most of the computational work of a DNN, it works based on the neural equation, that can be seen in the function 2.1. The output layer takes the output of the last hidden layer and transforms it to produce a meaningful output. In the case of regression or binary classification, there is usually only one neuron, but for multi-class classification, the amount of neurons corresponds to the amount classes to be classified. The output layer follows the same principle as the formula above, but it does not use a “normal” activation function. Either a linear activation function or no activation function is used for regression, or the softmax function, which can be seen in the function 2.4, is used for classification models.

$$f(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}} \tag{2.4}$$

But, how are deep neural networks trained? [27] The first step is forward propagation, the training data is fed in to the network and flows through the layers, where each neuron performs it operation, according to the function 2.1. Afterward the backpropagation or backpropagation of error is performed. [39] Then the output is compared to the desired outcome of the training data, hence, if the predicted outcome of the network matches the actual outcome. If it doesn't match, it calculates how far off the prediction is. The process of backpropagation is initiated, which uses an algorithm like Stochastic Gradient Descent (SGD) or one of its modern successors like Adam. SGD is an iterative method used to optimize an objective function, making it particularly advantageous for high-dimensional optimization problems. With each iteration SGD updates the model parameters by using the gradient of the loss function concerning the parameter. This

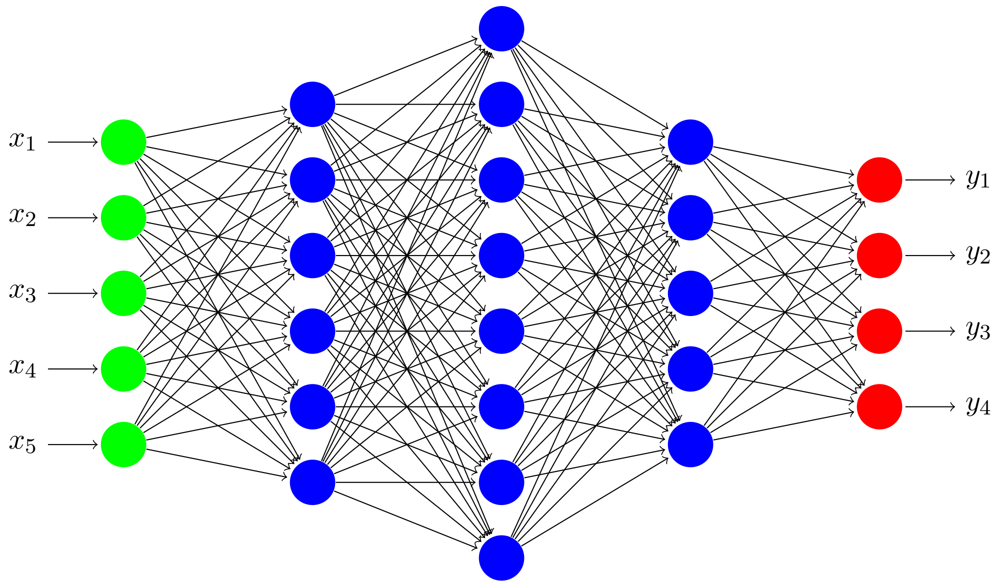


Figure 2.2.: Neural Network
🟩 Input Layer 🟦 Hidden Layer 🟥 Output Layer

isn't done for the whole dataset but just some data points. The learning rate describes how far a parameter is changed in the direction of a derivation for one iteration, in SGD this Learning Rate is constant and manually set.

Adaptive Moment Estimation (Adam) [25] is one of the successors of SGD. Adam, uses a variable learning rate, which is adapted for each parameter, which uses the estimation of the first moment, the mean, and the second moment, the variance to change the learning rate. Then this process is repeated until we are gone through the entire training set, which is called an epoch.

Core Layers in Neural Networks

In the following, we will discuss the core layers in neural networks, which are the dense layer, the convolutional layer and the pooling layer. We use them in our evaluation, hence, we will discuss them in detail.

Dense Layers

It is characterized by its fully connected architecture, like seen in the figure 2.2, meaning every node of the layer is connected to every node of the predecessor layer and every node of the successor layer. Dense layers are pivotal in learning intricate patterns from data, their versatility allows them to be stacked where each layer captures different levels of data abstraction. The theoretical function of a dense layer can be seen in the function 2.7. Where $x = (x_1, x_2, \dots, x_n)^T$ is the input vector, $W = \begin{pmatrix} w_{11} & w_{12} & \dots & w_{1j} \\ w_{21} & w_{22} & \dots & w_{2j} \\ \vdots & \vdots & \ddots & \vdots \\ w_{i1} & w_{i2} & \dots & w_{ij} \end{pmatrix}$ is the

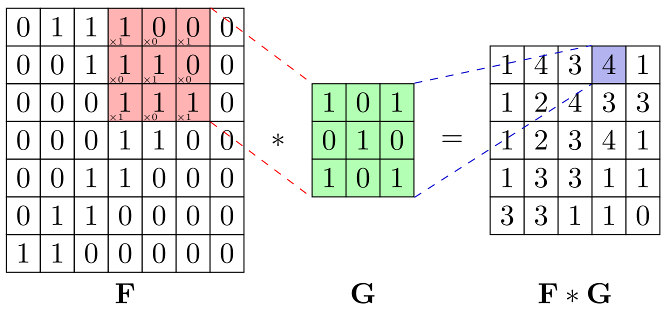


Figure 2.3.: Example Convolution
Figure based on a post by Riebesell. [38]

weight vector and $b = (b_1, b_2, \ldots, b_n)^T$ is the bias and f is the activation function and the output function $y = (y_1, y_2, \ldots, y_n)^T$.

$$y_j = f\left(\sum_{i=1}^{n} w_{ij} \cdot x_i + b_j\right) \tag{2.5}$$

$$y = \left[f\left(\sum_{i=1}^{n} w_{i1} \cdot x_i + b_1\right), f\left(\sum_{i=1}^{n} w_{i2} \cdot x_i + b_2\right), \ldots, f\left(\sum_{i=1}^{n} w_{ij} \cdot x_i + b_j\right)\right]^T \tag{2.6}$$

$$y = f(W^T x + b) \tag{2.7}$$

Convolutional Layer

Convolutional layers, are used in image processing [28, 42, 26], unlike dense layers that try to analyse an image as a whole, but uses the concept of convolution to get a set of smaller pictures which depict isolated features of a picture. A convolution describes how a function f is modified by a function g, for example in the discrete case, can be seen in the equation 2.8.

$$(f * g)[n] = \sum_{m=-\infty}^{\infty} f[m] \cdot g[n-m]] \tag{2.8}$$

But how does this correspond to neural networks and image processing, foremost, we have not 1Dimensional data, but 2Dimensional pictures, as you can see in the figure 2.3 and equation 2.9.

$$(f * g)[x, y] = \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} f[m, n] \cdot g[x-m, y-n] \tag{2.9}$$

In convolutional neural networks, matrices of weights called kernels or filters replace singular weights in dense layers. The output of the convolutional layer consists of feature maps, each representing the result of applying one filter to the input data. These maps capture different aspects such as edges or textures. A convolutional layer has two other variables: the number and size of filters, and the activation function. The terms stride and padding are used to describe the movement of the filter and the preservation of information on the edge of an input, respectively. In the figure 2.3, you can see a filter of size 3x3, which is applied to a 7x7 matrix, with a stride of 1 and no padding, which results in a 5x5 matrix. The stride determines the amount steps the filter takes between convolutions, with a larger stride resulting in a smaller filter map and less detail. Padding is an optional parameter that helps to maintain information on the edge of an input and preserve the scale of a picture.

Pooling Layer

Pooling is an often used concept in Convolutional Neural Networks to reduce the size of the individual feature map produced by the convolutional layer, while conserving important details of the feature map. Especially to produce an output, which has a similar size as the input, a pooling layer can be avoided [22]. There are multiple types of pooling operations, but the most used ones are Max-Pooling as seen in figure 2.4 and Average-Pooling seen in figure 2.5. Which operation is best depends on the type of data to be analysed and its pooling cardinality, which describes the amount of extracted features. There can be said, for smaller cardinalities a Max-Pooling should be used, for larger ones Average-Pooling [2]. In the complex layer terminology [16] there are, the convolution stage, the detector stage (Which is just the usage of a nonlinear activation function) and the pool stage, these stages are sublayers of the large convolutional layer. In the simple layer terminology, those three are all individual layers.

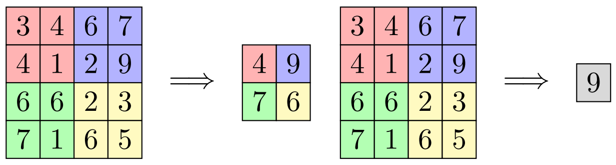


Figure 2.4.: Max Pooling
Left: Max Pooling, Filter: (2,2) Stride: 2 Right: Max Global Pooling

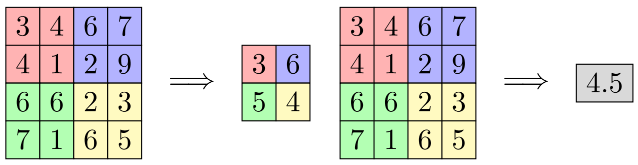


Figure 2.5.: Average Pooling
Left: Average Pooling, Filter: (2,2) Stride: 2 Right: Average Global Pooling

Now, with the usage of the convolutional layer and the pooling layer, we can preprocess data to get better results with the dense layers, which we still need to classify our data.

2.2. Neural Network Testing

The testing methods can be categorized into two distinct categories, which are akin to testing in conventional software. They are coverage criteria and test case generation. In the subsequent section, we shall examine various methodologies employed for testing singular deep neural networks [21]. But first, how do we define an erroneous behaviour of a neural network, as defined in the equation 2.10? Where $f : \mathbb{R}^{s_1} \rightarrow \mathbb{R}^{s_K}$ is a trained neural network, $\mathcal{H} : \mathbb{R}^{s_1} \rightarrow \mathbb{R}^{s_K}$ and a legitimate input $x \in \mathbb{R}^{s_1}$, then we define the erroneous behaviour as:

$$\arg\max_j f_j(x) \neq \arg\max_j \mathcal{H}_j(x) \tag{2.10}$$

Coverage Criteria

We need the coverage criteria to get a quantitative basis for deciding how thoroughly our neural network is tested and to ensure that in our network key aspects aren't overlooked. The criteria used in neural network testing differ from those in software testing.

Neuron Coverage

Neuron coverage is the most basic coverage criteria, it is the ratio of the amount of neurons activated by the test cases to the total amount of neurons in the network [36]. A neuron is activated if the output of the neuron is greater than zero. The neuron coverage is equivalent to the statement coverage in software testing, it helps to measure the thoroughness of the test cases.

Safety Coverage

The Safety Coverage is derived, by discretizing the input space into a set of hyper-rectangles [46], each hyper-rectangle exhibit the same pattern of activations of neurons, because it has a similar feature set. A hyper-rectangle is considered safely covered if a test case is classified correctly for all points in the hyper-rectangle.

Modified Condition/Decision Coverage

Modified Condition/Decision Coverage (MC/DC) is a method in software testing [17], which requires that each condition in a decision is tested independently and that each condition is shown to affect the decision outcome independently. This is done by varying the input of the condition while holding the other conditions constant [41]. The method is particularly useful in identifying cases where multiple conditions contribute to a decision's outcome. By using a sign change, and a value change to change, it is tried to exploit the relationship between those and use these as a coverage criterion.

Quantitative Projection

Quantitative Projection Coverage, is based on the assumption that the input differs on some kind of operation condition [5]. For example, for self-driving cars: weather, landscape or impeding items.

Test Case Generation

But how do we now generate our test case, to get some useful information? We have, of course, multiple methods to generate our test cases:

Input Mutation

Input Mutation, is a method where we take one of the inputs and transform it, by some predefined rules, to generate our test cases . For example, in combination with safety coverage, we try to mutate the input in that way so that we cover every hyper-rectangle.

Fuzzing

Fuzzing, is a method where we generate random inputs, which are modified from the input set, those are then fed into the network [46]. There by, we try to cover as much of the input space as possible. This can help us because a neural network, is designed for high-dimensional data, aim is to find cases where these small changes to the input cause the DNN to fail or behave unexpectedly.

Symbolic Execution

Despite the efficacy of input mutation and fuzzing in generating a substantial quantity of random data, it is uncertain whether certain test objectives will be fulfilled. With Symbolic Execution, we try to find out which input causes a part of a program to execute. One approach in Symbolic Execution is concolic testing, which is a hybrid testing technique, which is adapted for neural networks with DeepConcolic [41]. Concolic testing combines the concrete execution of a program, with random input generation, hence symbolic data. There by, the findings of the first step are used to generate the second set of inputs.

2.3. Spectrum-Based Fault Localization

Another important detail that requires introduction, Spectrum-Based Fault Localization (SBFL) [47], it is a technique to identify the source of faults in software. The term spectrum, describes in this context the execution profile of the program, which is collected during the execution of a test suite. During the execution, the coverage of each statement by a test case is recorded and if this test case is a success or a failure.

SBFL records not only failed test cases but also successful ones. This provides information on the contrast between failure and success, aiding in the identification of truly

		Is the statement covered?	
		Yes (1)	**No (0)**
Execution result	**Failed (0)**	a_{ef}	a_{nf}
	Successful (1)	a_{es}	a_{ns}

Table 2.1.: Symbols used in coefficients

faulty statements. For example, in the first case, code that is often executed or every time, would be marked faulty simply for being in a programme. In the second case, the code wouldn't be marked faulty because it is also executed frequently in successful test cases. The term "code coverage" is frequently called "executable statement hit spectrum" (ESHS), which denotes the extent to which certain components of the program under testing have been covered during execution. Some popular ESHS-based are based on a similarity coefficient, for example Tarantula [24] as seen in equation 2.11, Ochiai [33] seen in equation 2.12 or Dstar [49] in equation 2.13 (Often also called D^*, where $*$ denotes the used exponent).

$$\frac{\frac{a_{ef}}{a_{ef}+a_{nf}}}{\frac{a_{ef}}{a_{ef}+a_{nf}}+\frac{a_{es}}{a_{es}+a_{ns}}} \tag{2.11}$$

$$\frac{a_{ef}}{\sqrt{(a_{ef}+a_{nf})\cdot(a_{ef}+a_{es})}} \tag{2.12}$$

$$\frac{{a_{ef}}^*}{a_{es}+a_{nf}} \tag{2.13}$$

By the work of Lee et al. [20] the formula for Tarantula could even be more simplified like seen in equation 2.10. Yoo et al. [51] showed that using genetic programming to create ranking metrics can consistently exceed many human-designed ranking metrics, like the measures discussed before.

$$\frac{a_{ef}}{a_{ef}+a_{es}} \tag{2.14}$$

In the following, you can see an Example from Parsa et al. [34] to see what spectrum analysis is, and what a pitfall of this method is. The Listing 2.1 depicts some simple calculation program, which takes the two integers a and b as input and an integer c for the branching of the calculation. At line 11, the Max value is wrongly set, this is the fault that needs to be found. In the Table 2.2, you can see the suspiciousness values calculated with ochiai and tarantula.

But because line 8, is always executed with line 11 and is overall less executed than line 11, it is the more suspicious statement. Hence, we can see that spectrum analysis isn't the silver bullet, for eradicating faults in programs, but it can still be a useful tool to reduce the search space for the fault.

```
1 int getImpact()
2 {
3     scanf("%d %d %d", &a, &b, &c);
4      Impact = 0;
5      Division = 1;
6      Sum = a + b;
7      if ((a > 0) && (b > 0))
8          Division = a / b;
9      Max = b;
10     if (a > b)
11         Max = b; // Correct: Max = a;
12     if (c == 1)
13         Impact = Sum;
14     if (c == 2)
15         Impact = Division;
16     if (c == 3)
17         Impact = Max;
18     return Impact;
19 }
```

Listing 2.1: Example code snippet

Table 2.2.: Fault values, for code snippet

Line	a_{es}	a_{ef}	Tarantula	Ochiai
6	6	6	0.5	0.71
7	6	6	0.5	0.71
8	2	6	0.75	0.87
9	6	6	0.5	0.71
10	6	6	0.5	0.71
11	3	6	0.67	0.81
12	6	6	0.5	0.71
13	1	0	0.0	0.0
14	6	6	0.5	0.71
15	2	0	0.0	0.0
16	6	6	0.5	0.71
17	3	6	0.67	0.81
18	6	6	0.5	0.71

There are even more similarity coefficients, to an extent, which wouldn't be feasible to be covered in this work, for an extensive survey one can refer to the one by Wong et al. [48].

There are some other examples of SBFL, which don't use ESHS, for example Program Invariants Hit Spectrum, which the coverage of program invariants. Those individuals attempt to identify violations of program properties in failed program executions to locate bugs. Another one would be the Method Calls Sequence Hit Spectrum, which collects information about the sequence of method calls, during a test-case.

3. Related Work

In this chapter, we will discuss proposed approaches for repairing neural networks. These approaches [32] can be classified into three categories: training-centric, data-centric, and model-centric.

3.1. Training-Centric Approaches

The first training centric solution we want to present is AutoTrainer [54] an approach to identify varying issues in training vanishing and exploding gradient, dying ReLU oscillating loss and slow convergence. AutoTrainer first starts training the model and records data on the training like the loss, it then conducts regular analysis to identify possible issues encountered during training. Upon detecting an issue, the solution scheduler selects an appropriate repair method. If a problem persists even after trying a solution, the scheduler moves on to the next solution. This process continues until the issue is resolved, or all solutions are exhausted unsuccessfully. The solutions that are used are:

- Adding Batch Normalization Layers
- Substituting Activation Functions
- Adding Gradient Clipping
- Substituting Initializers
- Adjusting Batch Sizes
- Adjusting Learning Rates
- Substituting Optimizers

Another training centric solution is DeepDiagnosis [45], which isn't like AutoTrainer an automatic approach, but more functioning like a debugger. Hence, it tries to enable the developer to make sound decisions to enhance the training of a DNN. It monitors the training of the Network and checks for eight error conditions:

- Dead Nodes
- Saturated Activation Functions
- Exploding Tensors
- Not increasing Accuracy
- Not decreasing Loss
- Unchanged Weights

- Exploding Gradients
- Fading Gradients

When these conditions are met, DeepDiagnosis makes these findings available to the developer and provides a recommendation for actionable fixes to the developer.

3.2. Data-Centric Approaches

One data centric approach, we want to present for the repair of deep neural networks is DeepRepair [52] which uses a style guided approach to enhance the training data of an DNN. The Idea for this approach is it to mitigate the gap between the training data, and the real-world data provided later when the network is in production, which often contains noise patterns. This is done by the style of guided data augmentation by introducing the noise patterns, which are frequently leading to failure.

To make the augmentation more effective, DeepRepair uses a clustering-based method for generating corrupted data, by identifying and grouping similar failure patterns. There by, it is ensured that the data covers a broader spectrum of potential real-world failure scenarios, enhancing the robustness of the model.

Another data-centric approach is SENSEI [13], which uses a fuzz testing, which can be seen in the section 2.2, derived data augmentation approach to bridge the earlier described gap by exposing and mitigating vulnerabilities in DNNs.

3.3. Model-Centric Approaches

There are numerous model-centric approaches developed over the last few years, so we want to present some of them here at this point. For example, Apricot [53], a weight-adaption approach to fix deep learning models (DLM) iteratively. This is accomplished by leveraging insights from reduced deep learning models (rDLMs) trained on subsets of the original dataset. The approach is based on two main principles: firstly, smaller data sets help to retain the features that are essential for correct classification, and secondly, a set of rDLMs can, on average, classify test cases more accurately than a single model. Apricot generates the rDLMs and categorizes them based on their performance of specific test cases. Based on the correctly working rDLMs the average weights are used to correct the weights of the complete DLM toward the weights of the correctly classifying rDLMs or away from the misclassifying rDLMs.

Another approach is NeuRecover [44], which tries to resolve an issue, which often happens with retraining DNNs. Regression, which means, that if we try to address specific issues to enhance the performance of the models or its areas, it causes a performance decrease in other areas of the model. NeuRecover is attempting to leverage the training history to identify, which parameters should be adjusted, there by aiming to reduce regression. It firstly aggregates the locations of the faults in the network and by using the training history, attempts to make the smallest necessary adjustment, there by considering the dynamic nature of the network.

Finally, we introduce Arachne [40], a search-based approach for repairing DNNs by identifying and adjusting the weights that are most likely to cause the targeted misbehavior. Arachne uses differential evolution to generate patches, which are sets of adjusted neural weights that aim to correct the misclassifications. The algorithm begins with a group of potential solutions and evolves them over several generations towards greater fitness, as determined by a defined fitness function. In Arachne, the fitness function plays a critical role in guiding the search process. It assesses candidate patches based on their ability to correct misclassifications while maintaining correct classifications. This function strikes a balance between the need to correct errors, and the need to maintain the overall accuracy of the model. Some evolutions of the Search-based approach are DistrRep [4], which tries to handle multiple misclassifications at the same time, in contrast to the single misclassification handled by Arachne. Another one is AdRep [29], this approach tries to overcome the static view of the DNN, which is seen in Arachne.

3.4. DeepFault

DeepFault [11] is a white-box testing approach for neural networks, developed by Eniser et al. which is according to the Authors.

> ... the first fault localization-based white-box testing approach for DNNs.

There are two objectives to the approach, the identification of suspicious neurons, which have undesirable behaviour, where the respective neuron is suspected to lead to an undesirable outcome in the network. And the synthesis of new inputs for the neural network, to specially retrain the (most) suspicious values.

In this First Part, DeepFault is establishing a Hit Spectrum(HS) as seen in 3.1 for all neurons, which take the form of a tuple. The input and output layers are left out because they are considered inherently correct.

$$HS_n = (attr_n^{as}, attr_n^{af}, attr_n^{ns}, attr_n^{nf}) \tag{3.1}$$

- $attr_n^{as}$ is the number of times the neuron n is activated in successful test cases.
- $attr_n^{af}$ is the number of times the neuron n is activated in failed test cases.
- $attr_n^{ns}$ is the number of times the neuron n is not activated in successful test cases.
- $attr_n^{nf}$ is the number of times the neuron n is not activated in failed test cases.

Then the hit spectra are in a suspiciousness measures, in DeepFault Tarantula, Ochiai, D^3 are used, which can be seen in the equations 2.11, 2.12 and 2.13. These are not only used to identify one wrong neuron, but rather a set of wrong neurons. For that, the suspiciousness values of the neurons are used to sort the neurons in decreasing order of suspiciousness, if multiple neurons happen to get the same value, the neuron in a deeper layer is used.

Guided by the suspiciousness measures, DeepFault modifies the input, for which the neural network has made the correct decision, in a targeted way. The synthesis task is supported by a gradient ascent algorithm that aims to determine the degree to which an appropriately classified input should and could be modified to enhance the activation values of suspicious neurons.

4. Mutation-Based Accuracy Improvements

In the following chapter, we describe our approach for our “mutation-based accuracy improvements”, starting with the adaptation of DeepFault’s [11] spectrum fault localization. Afterwards, we go through our work and choices for the mutations of a neural network.

4.1. Spectrum-based fault localisation

For the Spectrum-based fault localization, we adapted the DeepFault [10] code to our needs. Especially, we adapted the functions to the recent versions of the used libraries and added some additional functions to save and load the data.

The functions adapted from the DeepFault paper are now invoked in the `run_analysis` function seen in listing 4.1, which is the main function for the analysis. We have implemented it that way, to ease the use of the analysis functions, in comparison to the original DeepFault code, which was just one large python script. The function manages the loading of the model and the experiment or working paths. It checks if the necessary steps of the analysis have already been executed, hence the data is already available out of a previous run and uses them instead, because we assume that the model has not changed, and the data is still valid.

The data we need are the classifications, the layer outputs and the spectrum matrices. The classifications are the indexes of the data of the test dataset sorted in to the categories of correctly predicted and not correctly predicted. The layer outputs are the information about the activation for each neuron, of each layer, for each member of the test dataset. Now with this information we can construct the spectrum matrix, which uses the layer outputs and classifications to determine sums of the symbols denoted in the table 2.1, for each neuron, if it was activated or not, and if the prediction of the network was right or wrong.

When all this data is aggregated, we can determine the suspiciousness of each neuron, which is the main goal of the analysis. All this data is then saved to the working path, for future runs, and the suspicious neurons are returned. The network’s neuron count is aggregated, and if the amount of selected neurons `susp_num` is less than the amount of modifiable neurons, the `susp_num` most suspicious neurons are returned.

Otherwise, all modifiable neurons are returned, ranked by their level of suspicion. To determine the suspiciousness, we use the D*, Tarantula, and Ochiai methods, which you can see in the background chapter 4.1. We also added the option to return a specified amount of random neurons, which can be useful for testing purposes, you can see the function in listing 4.2. The function returns a list of tuples in the format of $(layer, layer\_index)$, ranked from the most suspicious neuron to the least suspicious.

```python
def run_analysis(model_name, approach, test_images, test_labels ,susp_num
    =-1, star=3, group_index=1):
    model = dm.get_model(model_name)
    experiment_path = ut.create_experiment_dir(model_name)

    file_path_classification = experiment_path + '_classifications.h5'
    file_path_layer_outs = experiment_path + '_layer_outs.h5'

    if os.path.isfile(file_path_classification) and os.path.isfile(file_path_layer_outs):
        correct_classifications, misclassifications = ut.load_classifications(experiment_path, group_index)
        layer_outs = ut.load_layer_outs(experiment_path, group_index)
    else:
        correct_classifications, misclassifications, layer_outs, predictions = tn.test_model(model, test_images,
                                                                                               test_labels)
        ut.save_classifications(correct_classifications, misclassifications, experiment_path, group_index)
        ut.save_layer_outs(layer_outs, experiment_path, group_index)

    trainable_layers = tn.get_trainable_layers(model)

    file_path_spectrum_matrices = experiment_path + '_spectrum_matrices.h5'
    if os.path.isfile(file_path_spectrum_matrices):
        scores, num_cf, num_uf, num_cs, num_us = ut.load_spectrum_matrices(experiment_path, group_index)
    else:
        scores, num_cf, num_uf, num_cs, num_us = tn.contruct_spectrum_matrices(model, trainable_layers,
                                                                                 correct_classifications,
                                                                                 misclassifications, layer_outs)
        ut.save_spectrum_matrices(scores, num_cf, num_uf, num_cs, num_us, experiment_path, group_index)

    num_neurons = 0
    for score in scores:
        num_neurons += len(score)
    if susp_num == -1 or susp_num > num_neurons:
        susp_num = num_neurons

    approach_name = approach
    if approach == "dstar":
        approach_name = approach + str(star)
    file_path_suspicious_neurons = experiment_path + '_' + approach_name + '_SN' + str(susp_num) + '_suspicious_neurons.h5'
    if os.path.isfile(file_path_suspicious_neurons) and approach != "random":
        suspicious_neuron_idx = ut.load_suspicious_neurons(experiment_path, approach_name, susp_num, group_index)
    else:
        match approach:
            case "tarantula":
                suspicious_neuron_idx = an.tarantula_analysis(
  trainable_layers, scores, num_cf, num_uf, num_cs, num_us,
                                                               susp_num)
                ut.save_suspicious_neurons(suspicious_neuron_idx,
  experiment_path, approach, susp_num, group_index)
            ...  # other cases
    return suspicious_neuron_idx
```

Listing 4.1: Main analysis function

```
def random_neurons(trainable_layers, scores, suspicious_num):
    layer_length = []
    for score in scores:
        layer_length.append(len(score))
    neurons = []
    index = 0
    for i in trainable_layers:
        ran = range(layer_length[index])
        for j in ran:
            neurons.append((i, j))
        index += 1
    samples = random.sample(neurons, suspicious_num)
    return samples
```

Listing 4.2: Random Choosing, of neurons

The saving and loading functions in listings 4.3 required adaptation due to the depreciation of the `.value` property in h5py [8]. Therefore, every loading function needs to use indexing instead of the property used by the old DeepFault code. Save and load functions were added for the spectrum matrices and ranked neurons, for saving time by not having to re-run the analysis every time the program is executed. Because of the similarity of the functions, we only show the saving and load function for the classifications in listing 4.3. All functions can be found in the appendix A.4.

```
def load_classifications(filename, group_index=1):
    filename = filename + '_classifications.h5'
    try:
        with h5py.File(filename, 'r') as hf:
            group = hf.get('group' + str(group_index))
            correct_classifications = group['correct_classifications'][:]
            misclassifications = group['misclassifications'][:]

            print("Classifications loaded from ", filename)
            return correct_classifications, misclassifications
    except (IOError) as error:
        print("Could not open file: ", filename)
        sys.exit(-1)


def save_classifications(correct_classifications, misclassifications,
    filename, group_index=1):
    filename = filename + '_classifications.h5'
    with h5py.File(filename, 'a') as hf:
        group = hf.create_group('group' + str(group_index))
```

```python
        group.create_dataset("correct_classifications", data=
    correct_classifications)
        group.create_dataset("misclassifications", data=
    misclassifications)

    print("Classifications saved in ", filename)
    return
```

Listing 4.3: Saving and loading the classifications

Additionally, we modified the `get_layer_outs` function as shown in listing 4.4 from native Keras [6] calls to TensorFlow [31] via `tf.keras`, which uses Keras as its backend, as recommended from Keras versions 2.3.0. [37] onward. Because of its better integration with the rest of the TensorFlow library.

```python
def get_layer_outs(model, test_input):
    inp = model.input
    outputs = [layer.output for layer in model.layers]  # all layer
    outputs
    functors = [tf.keras.backend.function([inp], [out]) for out in
    outputs]  # evaluation functions
    layer_outs = [func([test_input]) for func in functors]
    return layer_outs
```

Listing 4.4: Layer outs

4.2. Mutation of the neural network

Now we move on to the second step in our approach: modifying our network. We designed the function `run_modification_algorithm` in listing 4.5 and schematically shown it in 4.1 for this purpose.

We hand over multiple parameters to the function. The first parameter is `run`, which is used to be able to clearly identify the different runs of the experiment during the experimentation, it can be omitted in a production environment. The next parameter is `modelname`, which is the name of the model that we want to modify and is used to identify the model and being able to load the model, and it associated data used in the analysis, as seen in the previous chapter 4.1. The next parameters are `train_images, train_labels, test_images, test_labels` the test data is needed to evaluate the model and perform the spectrum-based fault localisation, and the training data is needed to retrain the model, if this options is chosen.

Next we can choose `analysis_approach`, which is the similarity coefficient we want to use to rank our neurons. The next parameter is `mutation_function`, which is the mutation function we want to use to modify the weights and biases of the neurons, the mutation functions are discussed in the next subsection 4.2.1. The `value` is used to specify the value parameter of the mutation function, which is used to modify the weights and biases of the neurons.

The `old_loss` and `old_accuracy` are then used to save time by offloading the initial model evaluation required at the start of the function, which now only needs to be done once, rather than each time the function is used. In a production environment, these parameters can be omitted, and the evaluation would be performed in the function.

The parameter `train_between_iterations` is used to indicate whether we want to retrain the model or not, we can also specify `loss_offset` and `loss_accuracy` which is used to prevent the model from being terminated prematurely if the loss and accuracy are only slightly worse than the previous iteration, but may improve in the next iteration. We also have the option to specify `regress_loss` and `regress_accuracy` which is used to indicate whether we want to regress the loss and accuracy, which is used in the break conditions we will describe in the next paragraph.

The break conditions are used to determine when the model should be terminated. We have four different break conditions, which are used to compare the loss and accuracy and determine whether the model should be terminated, these can be seen in the listing 4.5 in lines 19 to 23. We have chosen them to see, which condition has the greatest influence on the improvement of the model. For the readability of the model we have omitted the data logging functions of the function in listing 4.5, the full function can be found in the appendix A.5.

At line 7, `tf.random.set_seed(42)` is included for evaluation purposes, to enable comparison of results from different runs, it would be omitted in a production environment.

```
1  def run_modification_algorithm(run, modelname, train_images, train_labels
       , test_images, test_labels, analysis_approach,
2                                  mutation_function, old_loss, old_accuracy,
        train_between_iterations=False, value=0,
3                                  max_num_iterations=-1,
4                                  loss_offset=0, accuracy_offset=0,
       regression_loss_offset=False,
5                                  regression_accuracy_offset=False,
       compare_loss=False, compare_accuracy=True,
6                                  compare_and_both=False):
7      tf.random.set_seed(42)
8      model = data_managment.get_model(modelname)
9      suspicous_neurons = nn_analysis.run_analysis(modelname,
       analysis_approach,test_images,test_labels,max_num_iterations)
10     epoch = 0
11     data_to_append = []
12
13     while len(suspicous_neurons) > 0:
14         mutation_function(model, suspicous_neurons.pop(), value)
15         model.compile(optimizer='adam', loss='
       sparse_categorical_crossentropy', metrics=['accuracy'])
16         if train_between_iterations:
17             model.fit(train_images, train_labels, epochs=1)
18         new_loss, new_accuracy = model.evaluate(test_images, test_labels)
19         if compare_and_both:
20             if (new_loss + loss_offset) > old_loss and (new_accuracy +
       accuracy_offset) < old_accuracy: break
21         else:
22             if compare_loss and (new_loss + loss_offset) > old_loss:
       break
```

```
            if compare_accuracy and (new_accuracy + accuracy_offset) <
    old_accuracy: break
        if regression_loss_offset: loss_offset /= 2
        if regression_accuracy_offset: accuracy_offset /= 2
        epoch += 1
        old_accuracy = new_accuracy
        old_loss = new_loss
    return model
```

Listing 4.5: Modification Algorithm

4.2.1. Mutation Functions

Now, let's discuss the mutation functions that we pass to our main function. We have implemented multiple mutation functions, which are used to modify the weights and biases 2.7 of the neurons in the model. We not only mutate the dense layers, but also the bias and filter of the 2D Convolution Layers, this can later be extended to the convolutional layers in the first dimension and third dimension. Two different mutation function approaches, were implemented, we either assign a value, or we scale the existing values by a factor. Each of these approaches, which are used to modify the weights and biases of the neurons, can be seen in the listing 4.6 and 4.8. The scalar mutation functions can be seen in the listing 4.7 and 4.9. We have chosen these because they are the most simplistic approaches we could think of, and we wanted to see if they would be enough to improve the model and could be a good starting point for future work.

Because of the different types of layers, they need to be addressed differently. The function `modify_weight_all_random_gauss` in listing 4.6 is used to assign a random vector of weights, which is generated by a normal distribution, to the weights of the neurons. While the function `modify_weight_one_random_gauss` in listing 4.6 is used to assign a random value to the weights of the neurons, which is the same across all values.

```
def modify_all_weights(model, coordinate, new_value):
    index_layer, index_neuron = coordinate
    layer = model.get_layer(index=index_layer)
    layer_weights = layer.get_weights()
    if layer.__class__.__name__ == 'Conv2D':
        layer_weights[0][:,:,:,index_neuron] = new_value
    else:
        layer_weights[0][index_neuron, :] = new_value  # Set all weights
    in the specified neuron to the new value
    layer.set_weights(layer_weights)
    return model

def delete_neuron(model, coordinate):
    return modify_all_weights(model, coordinate, 0)

def modify_weight_one_random_gauss(model, coordinate, sigma=0.5):
    random_value = random.gauss(0, sigma)
    return modify_all_weights(model, coordinate, random_value)

```

```
def modify_weight_all_random_gauss(model, coordinate, sigma=0.5):
    index_layer, index_neuron = coordinate
    layer = model.get_layer(index=index_layer)
    layer_weights = layer.get_weights()
    if layer.__class__.__name__ == 'Conv2D':
        random_values = np.random.normal(0, sigma, layer_weights[0][:, :,
 :, index_neuron].shape)
        layer_weights[0][:, :, :, index_neuron] = random_values
    else:
        random_values = np.random.normal(0, sigma, layer_weights[0][
index_neuron, :].shape)
        layer_weights[0][index_neuron, :] = random_values
    layer.set_weights(layer_weights)
    return model
```

Listing 4.6: Weight Modification Functions

The scalar mutation functions are used to scale the existing weights and biases of the neurons. They work nearly the same as the previous functions, but instead of assigning a new value to the weights and biases, they scale the existing values by a factor. They can be seen in the listing 4.7 and 4.9. The two functions `modify_all_weights_by_scalar_random_gauss` and `modify_weight_all_by_scalar_random_gauss` are used to scale the weights by a random value, while the first function scales all weights by the same value, the second function scales each weight by a different random value.

```
def modify_all_weights_by_scalar(model, coordinate, scalar):
    index_layer, index_neuron = coordinate
    layer = model.get_layer(index=index_layer)
    layer_weights = layer.get_weights()
    if layer.__class__.__name__ == 'Conv2D':
        layer_weights[0][:,:,:,index_neuron] *= scalar
    else:
        layer_weights[0][index_neuron, :] *= scalar  # Set all weights in
 the specified neuron to the new value
    layer.set_weights(layer_weights)
    return model

def modify_all_weights_by_scalar_random_gauss(model, coordinate, sigma
=0.5):
    random_value = random.gauss(0, sigma)
    return modify_all_weights_by_scalar(model, coordinate, random_value)

def modify_weight_all_random_by_scalar_gauss(model, coordinate, sigma=1):
    index_layer, index_neuron = coordinate
    layer = model.get_layer(index=index_layer)
    layer_weights = layer.get_weights()
    if layer.__class__.__name__ == 'Conv2D':
        random_values = np.random.normal(0, sigma, layer_weights[0][:, :,
 :, index_neuron].shape)
        layer_weights[0][:, :, :, index_neuron] *= random_values
    else:
        random_values = np.random.normal(0, sigma, layer_weights[0][
index_neuron, :].shape)
        layer_weights[0][index_neuron, :] *= random_values
    layer.set_weights(layer_weights)
    return model
```
Listing 4.7: Scalar Weight Modification Functions

The bias modification functions are used to modify the biases of the neurons. They are working in a similar way as the weight modification functions, but instead of modifying the weights, they modify the biases of the neurons. Hence, they aren't as complex as the weight modification functions, because they are only affecting a single value, rather than a matrix of values. We either assign a new value to the biases or scale the existing values by a factor, which can be seen in the listing 4.8 and 4.9.

```
def modify_bias(model, coordinate, new_value):
    index_layer, index_neuron = coordinate
    layer = model.get_layer(index=index_layer)
    layer_weights = layer.get_weights()
    layer_weights[1][index_neuron] = new_value
    layer.set_weights(layer_weights)
    return model

def modify_bias_random_gauss(model, coordinate, sigma=0.5):
    random_value = random.gauss(0, sigma)
    return modify_bias(model, coordinate, random_value)
```
Listing 4.8: Bias Modification Functions

```
def modify_bias_by_scalar(model, coordinate, scalar):
    index_layer, index_neuron = coordinate
    layer = model.get_layer(index=index_layer)
    layer_weights = layer.get_weights()
    layer_weights[1][index_neuron] *= scalar
    layer.set_weights(layer_weights)
    return model

def modify_bias_by_scalar_random_gauss(model, coordinate, sigma=0.5):
    random_value = random.gauss(0, sigma)
    return modify_bias_by_scalar(model, coordinate, random_value)
```
Listing 4.9: Scalar Bias Modification Functions

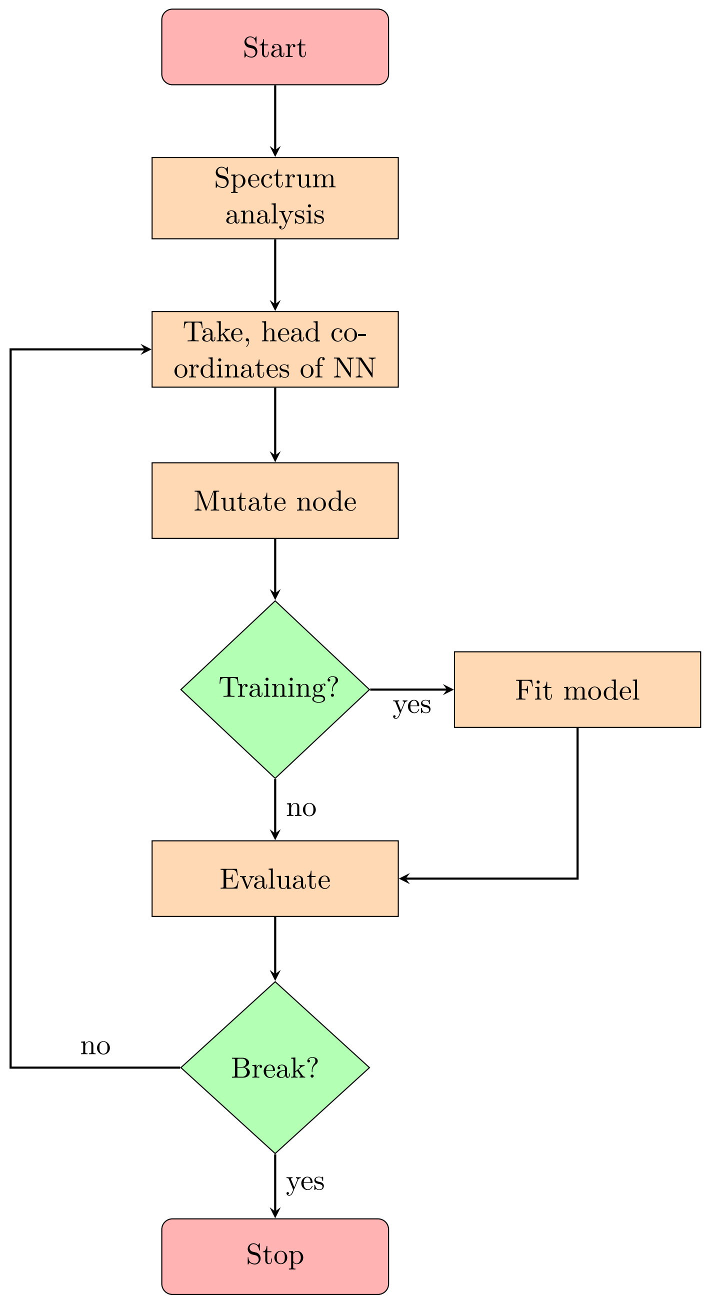


Figure 4.1.: Flowchart of the `run_modification` algorithm.

5. Evaluation and Results

In this chapter, we will discuss the performance of our approach towards neural networks. By performance, we mean the ability of our approach to improve the accuracy of a pre-trained model and how much our approach can reduce the loss of a pre-trained model. We will explore various options available through our approach to the elicit optimal results. To achieve this, the evaluation will be based on the following research questions:

RQ1: Impact of Training on Mutated Models How does training a mutated pre-trained model impact its performance metrics?

RQ2: Effects of Mutation Without Training What is the effect on performance metrics when a pre-trained model is mutated without further training?

RQ3: Effects of the Extent of pre-Training What is the effect on performance metrics, depending on how many epochs the model is pre-trained?

RQ4: Training Dataset Size and Approach Performance What is the impact of the size of the training dataset on the approach's effectiveness?

RQ5: Influence of Suspiciousness Measures How do different suspiciousness measures influence the outcomes of the experiments?

RQ6: CNN vs. DNN Architectural Efficiency Which architecture yields better results for our approach: CNN or DNN?

RQ7: Offset Variations in Loss and Accuracy How do variations in offset for loss and accuracy affect the approach's performance?

RQ8: Break Conditions and Algorithm Performance What is the effect of different break conditions on the efficiency and effectiveness of the algorithm?

RQ9: Contributions of Different Mutation Functions How do different mutation functions contribute to the model's performance with our Algorithm?

5.1. Setup

We evaluated our approach on a Workstation consisting of an AMD Ryzen 9–3900X 12-Core Processor 4,6 GHz, with 32 GB of RAM and an NVIDIA GeForce RTX 4070Ti GPU with 12 GB of VRAM and 7680 CUDA Cores. The setup is running Ubuntu release 22.04, running with Windows Subsystem for Linux, on Microsoft Windows 11 Pro, we use it because since version 2.11[3] TensorFlow doesn't support GPU acceleration natively any more. Regarding the Software, we are using Python version 3.10.12 and using TensorFlow version 2.14.1, with CUDA version 12.3.

Table 5.1.: Overview of Neural Network Architectures used for evaluation

Model Name	Mod. Param.	Architecture
DNN1	16	$< 16 >$
DNN2	64	$< 4x16 >$
CNN1	12	$< CL, 4 >$
CNN2	20	$< CL, CL, 4 >$

Table 5.2.: Results of the initial Training of the different models

Model		Accuracy			Loss		
Name	Init. Epochs	Full	Half	Quarter	Full	Half	Quarter
DNN1	1	82.84%	81.49%	78.65%	49.11%	53.46%	62.00%
	6	84.23%	83.76%	83.57%	44.42%	44.48%	46.94%
DNN2	1	80.46%	79.20%	75.97%	54.05%	57.90%	66.21%
	6	83.52%	83.04%	82.74%	44.23%	46.79%	48.88%
CNN1	1	74.34%	77.31%	42.62%	72.88%	66.83%	147.63%
	6	83.69%	82.14%	61.06%	48.44%	47.35%	95.61%
CNN2	1	70.49%	54.52%	63.20%	80.13%	111.68%	100.31%
	6	78.43%	77.37%	74.56%	60.28%	59.29%	65.74%

5.1.1. Architecture

We are using the Fashion-MNIST dataset[50] for our experiments, which consists of 60,000 training images and 10,000 test images, each of size 28×28 pixels, with 10 classes. For the evaluation we haven't used not only the whole, but also a half and a quarter of the training data, which are derived from the original training data, by using Scikit-learn's[35] `train_test_split` function, with a test size of 0.5 and 0.75 respectively.

We used 2 DNNs and 2 CNNs, in the following table 5.1 a plain number describes the amount of neurons in a dense layer. CL describes a combination of a convolutional layer with a stride of $(1, 1)$ and a kernel size of $(3, 3)$ and a pooling layer with a pool size of $(2, 2)$. We used Adam as our optimizer with a learning rate of 0.001.

The models were selected for their small size and ability to accommodate both deep neural networks and convolutional neural networks. This was done to save time and allow for the evaluation of as many parameters as possible within the time frame of this thesis.

5.1.2. Parameters

Fo our evaluation we used the following parameters:

Similarity coefficient: tarantula, dstar with value 3, ochiai and random

Mutation functions: `modify_weight_one_random_gauss`, `modify_weight_all_random_gauss`, `modify_bias`, `modify_bias_random_gauss`, `modify_all_weights`, `modify_all_weights_by_scalar`, `modify_all_weights_by_scalar_random_gauss`, `modify_weight_all_random_by_scalar_gauss`

Break conditions: loss, accuracy, loss and accuracy, loss or accuracy

Loss offset: 0.005, 0

Accuracy offset: 0.01, 0

Loss and accuracy regression: True for all runs

Values: -1, -0.5, 0, 0.5, 1
0 just for value assignment, basically a deletation of a neuron

Sigma for random: 0.5, 1

We chose these parameters to evaluate the impact of different mutation functions, suspiciousness measures, and break conditions on the performance of our approach. Our main objective was to cover a large area of the possible parameter space to evaluate the performance of our approach, with as little parameters as possible.

5.2. Results

This section will present the findings of our experiments. The two measures we used to compare our methods are the change in loss and the change in accuracy. The change is based on the benchmark, which is the model without any mutations. For the further trained models, we compared the comparison model with the respective epoch of the algorithm. For example, the results of the 1st epoch of our algorithm for a model pre-trained for 1 epoch is compared the model trained for 2 epochs, for 6 epochs it would be compared to the model trained for 7 epochs. For the models without training, we always compared to the base accuracy and loss seen in the table 5.2. The data, which is shown in the following figures, doesn't describe the improvements at the endpoint of the Algorithm, but the improvements at the end of each epoch, compared to the benchmark.

5.2.1. Impact of Training on Mutated Models

Our models 5.1 show slightly better loss and accuracy than the benchmark during training. For the first four epochs, the loss and accuracy of the models are significantly better than the benchmark. We see an improvement in the median for the accuracy of 1.75% and for the loss of 3–4%. Afterwards, we see a decrease of the median to the baseline of 1. After the 11th epoch, we see a drop of the median below the baseline. This holds also true for the loss, where we see the same drop after the 11th epoch.

For some epochs we can see an improvement to 2.5% for the accuracy to an extreme of 7.5%. For the loss, we see an 12% improvement to an extreme of 40%. We would attribute this to the mutation of the model and the breakage of the algorithm, as we can see in the figure 5.2. Because we see a decrease of the data points, per epoch, after the 11th epoch, we limit all figures after this to 10 epochs, for the trained runs.

5.2.2. Effects of Mutation Without Training

As you can see in the figure 5.3, the models without training show a significant improvement in accuracy and loss during the first three epochs. But we see only a median that isn't significantly better than the baseline of 1, for both loss and accuracy. We see a in crease for the extremes stepwise, for each of the first 3 epochs, till we reach an improvement of 25%, but on the other hand a decrease of -17% on the other extreme. The same for the loss, where we see an improvement of 45% to a decrease of -50%.

The three steps we are seeing, can be attributed to the comparison to only the 1st epoch, or 6th epoch respectively, so we are summing up any improvement of the model. The reason for the sharp drop we see in the figure 5.4, we would attribute to the increasing damage we are doing to the model, as we are not training it between the mutations. Because we see a decrease of the data points, per epoch, after the 3rd epoch, we limit all figures after this to 5 epochs, for the not trained runs.

5.2.3. Effects of the extend of pre-Training

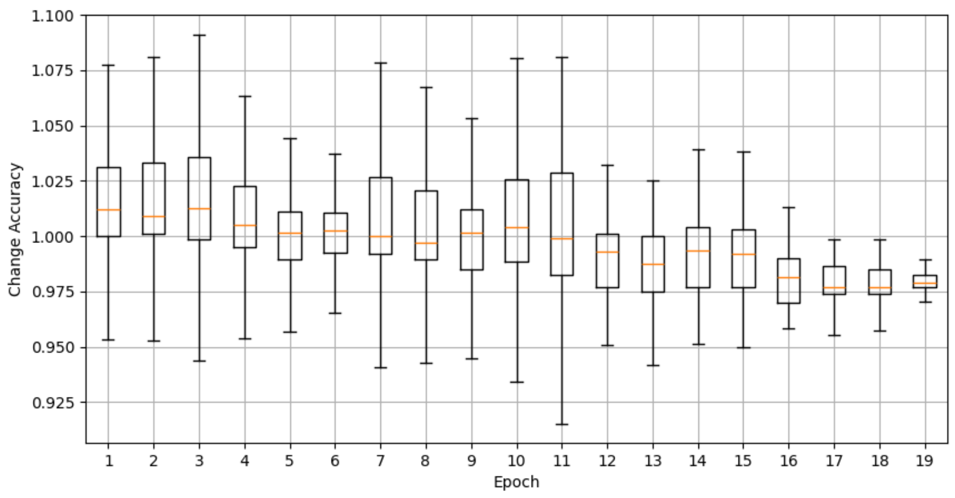
As we can see in the figure 5.5, the models with 1 initial epoch show the largest potential for improvements. While for the not trained runs, the median for loss and accuracy is not significantly better than the baseline, we can see a significant improvement for the extremes. With a maximum improvement of 25% for the accuracy and 45% for the loss, for the 1st epoch. But a decrease of up to -40% for the accuracy and -70% for the loss. For the 6 initial epochs, we see a lower deviation.

For the trained runs, that can be seen in figure 5.6,we see a smaller improvement over all, which is in line with the results of the previous two subsections. But here we also see a better result for the 1 initial epoch, than for the 6 initial epochs. But also a larger deviation for 1 initial epoch, than for the 6 initial epochs.

We would guess that the reason tHat the 1 initial epoch has more room for improvement, then the 6 initial epoch. As we can see in the table 5.2 that the accuracy and loss of the 6 initial epochs is already better than the 1 initial epoch, at the beginning.

5.2.4. Training Dataset Size and Model Performance

As you can see in the figure 5.7, the dataset size doesn't have a significant impact on the loss and accuracy of the models without training, for the full and half dataset. While we see a improvement for the full and half dataset, for the accuracy of often up to 25% and for the loss a slight different result with 50% and 30% respectively. For the quarter dataset we see a more negative result, while an improvement is still possible, we even



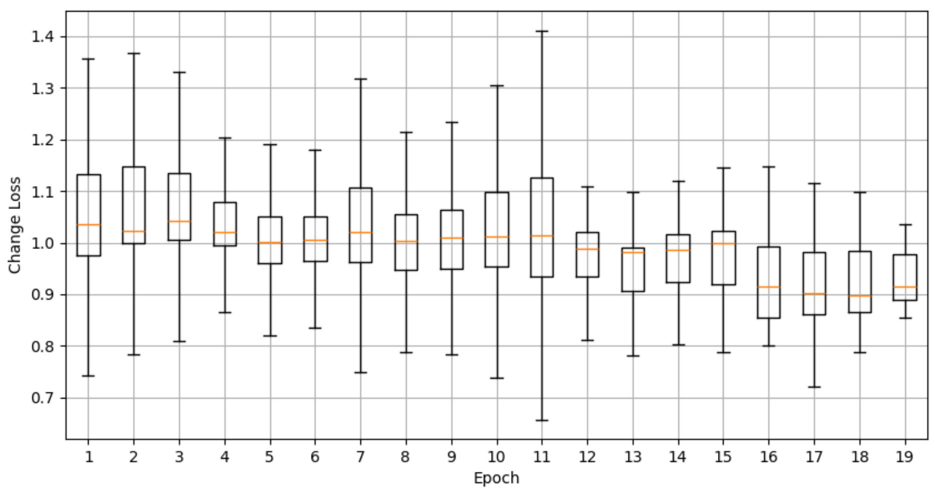


Figure 5.1.: Loss and accuracy of the models, with training

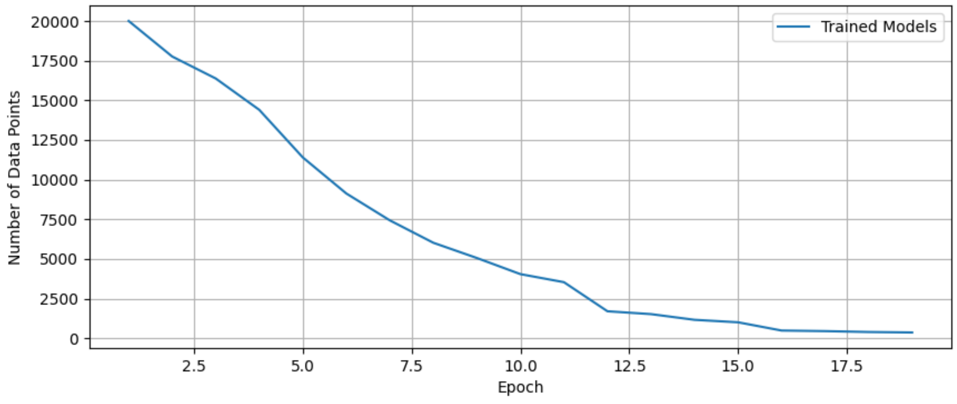


Figure 5.2.: Decay of data points, per epoch, with training

see the 3rd quartile just slightly above the baseline. And the rest all below the baseline, so roughly only a quarter of the models are better than the baseline.

For the trained runs, that can be seen in the figure 5.8, we see a larger difference between the full and half dataset. With the full dataset, we see an improvement of up to 25% for the accuracy and 95% for the loss. We even see the 1st quartile above the baseline, so three quarters of the models are better than the baseline. This holds also true for the half dataset, where we see an improvement of up to 7.5% for the accuracy and a bit more than 25% for the loss. Here also the quarter dataset is the worst, with only half of the models better than the baseline. And even that at most for 5% for the accuracy and around 10% for the loss.

5.2.5. Influence of Suspiciousness Measures

For both trained, as seen in the figure 5.10, and untrained runs, as seen in the figure 5.9, Tarantula appears to be the most promising measure of suspiciousness. Three quarters of the models are better than the baseline, with an improvement of up to 25% for the accuracy and 30% for the loss, for the not trained runs and 22.5% for the accuracy and 90% for the loss, for the trained runs. Ochiai seems also promising, but in contrast to Tarantula, we the third quartile is mostly below the median of tarantula, for untrained runs. Dstar is the worst performing of the three measures, but still positive with a median slightly above the baseline, for the untrained runs. Random performs the worst, as expected. For the trained runs, while we see the improvements, as earlier mentioned, with Tarantula, we see no specific improvement for the other measures. All of them having a median near the baseline, and just a slightly different deviation and both sides of the baseline.

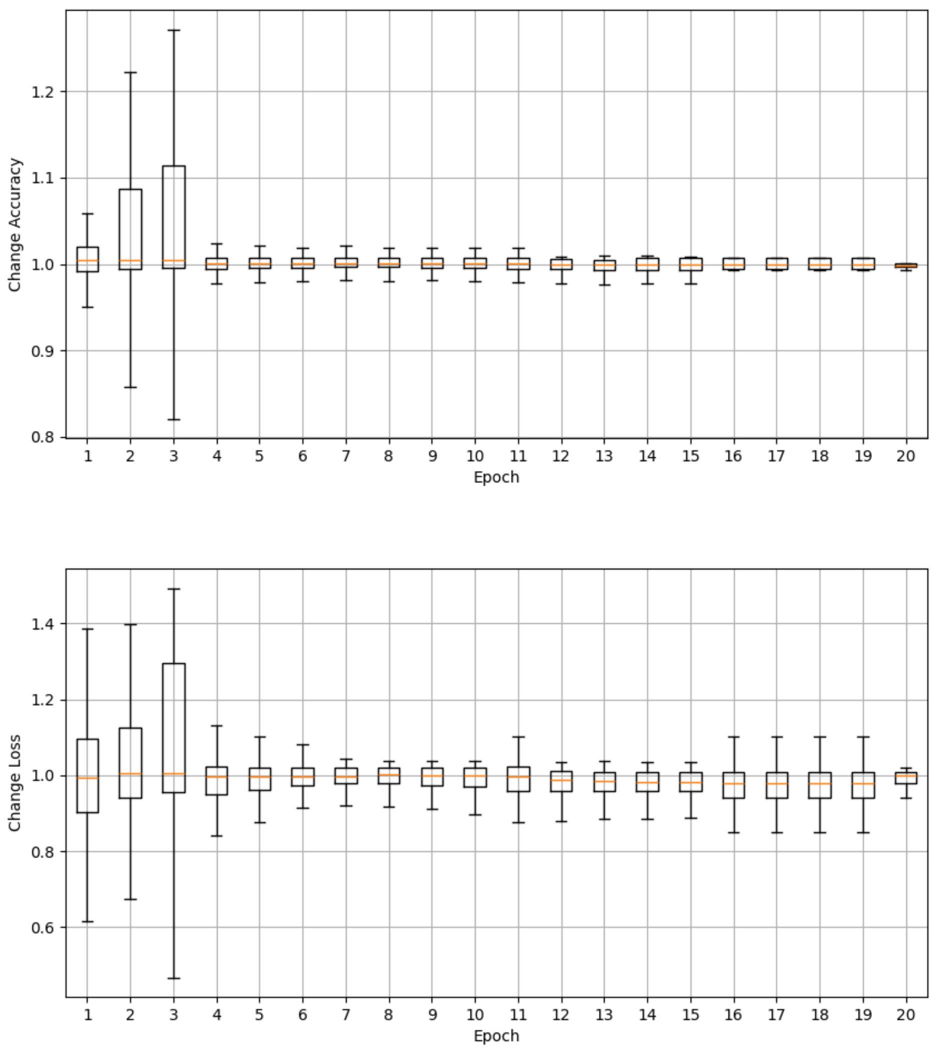


Figure 5.3.: Loss and accuracy of the models, without training

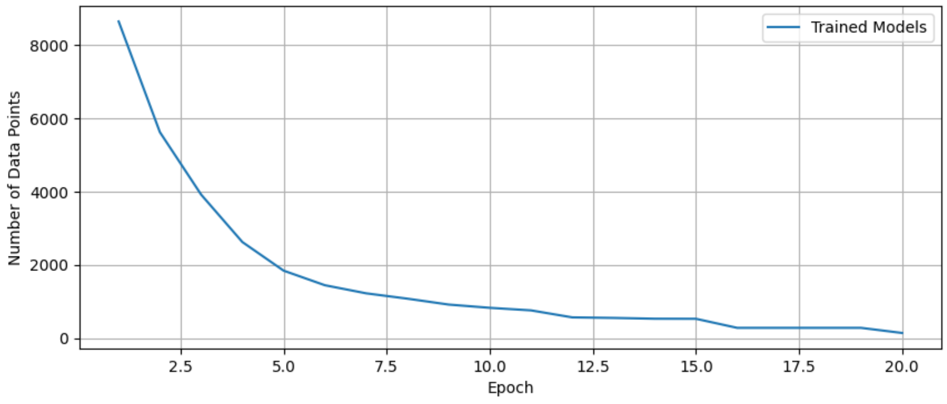


Figure 5.4.: Decay of data points, per epoch, without training

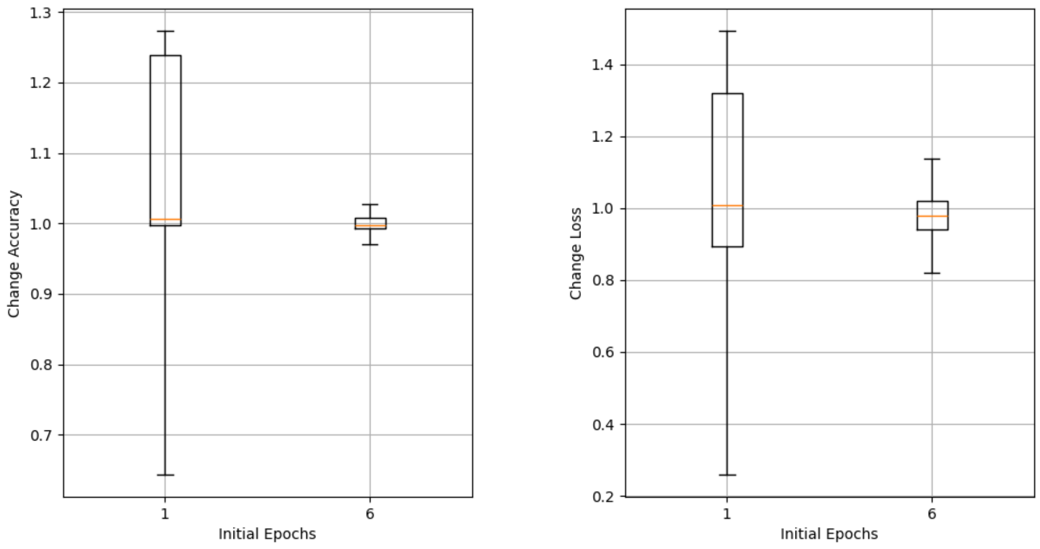


Figure 5.5.: Loss and accuracy, without training, with different initial epochs

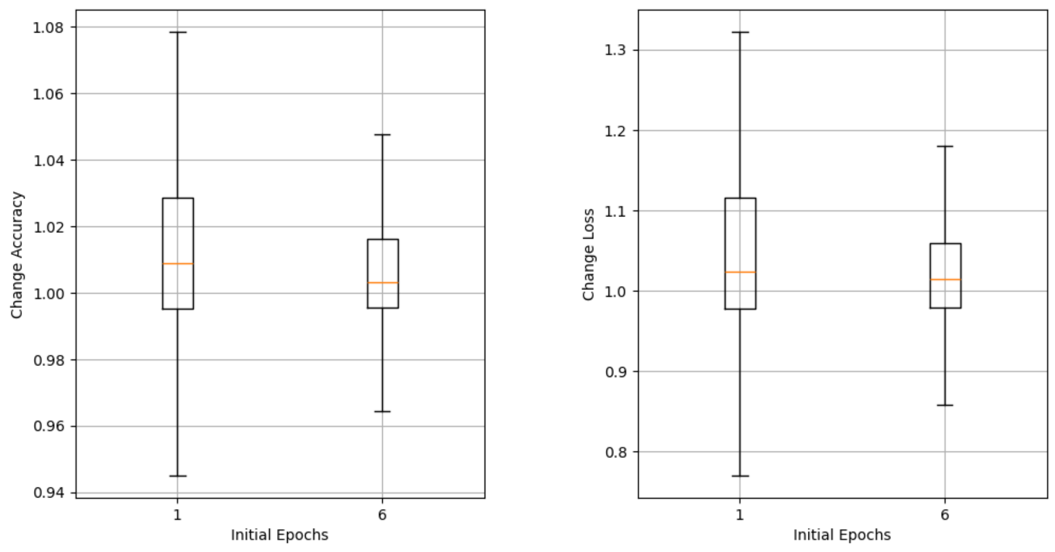


Figure 5.6.: Loss and accuracy, with training, with different initial epochs

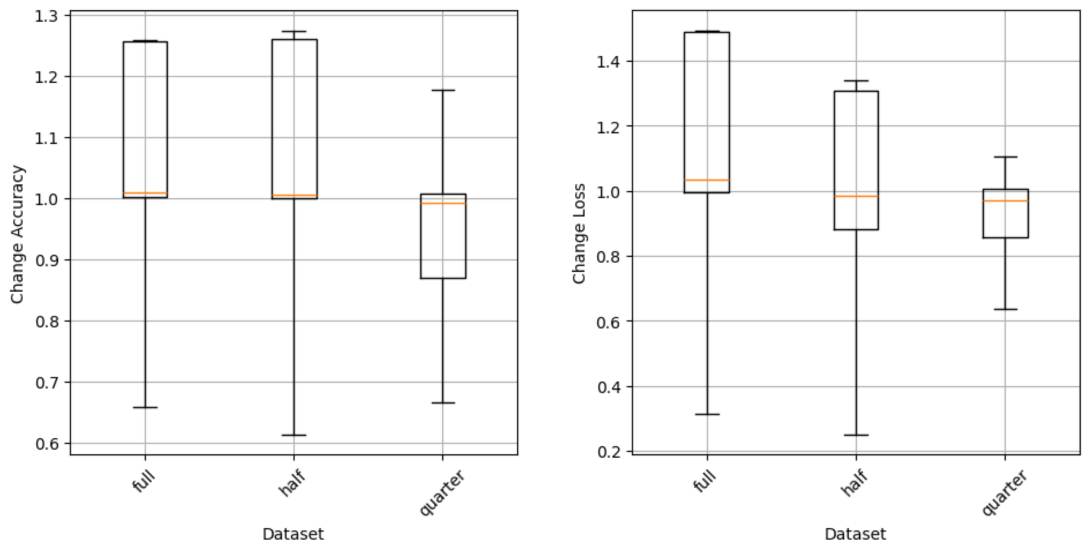


Figure 5.7.: Loss and accuracy, without training, with different dataset sizes

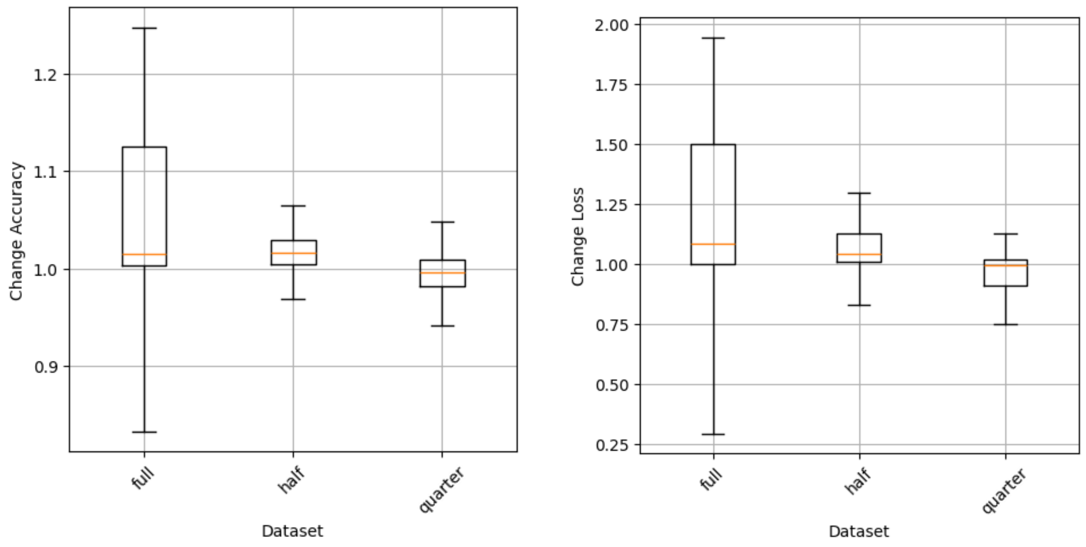


Figure 5.8.: Loss and accuracy, with training, with different dataset sizes

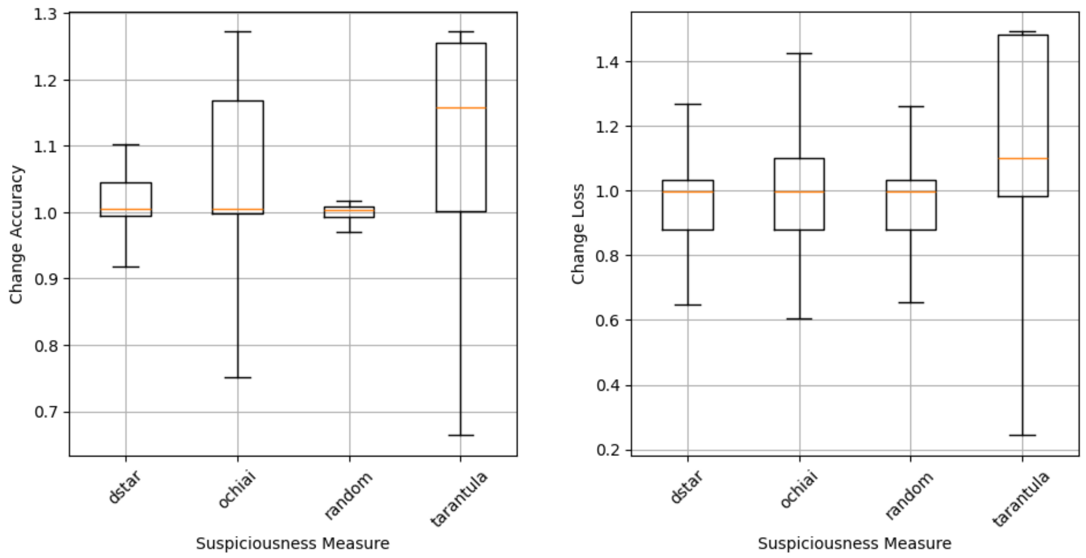


Figure 5.9.: Loss and accuracy, without training, with different suspiciousness measures

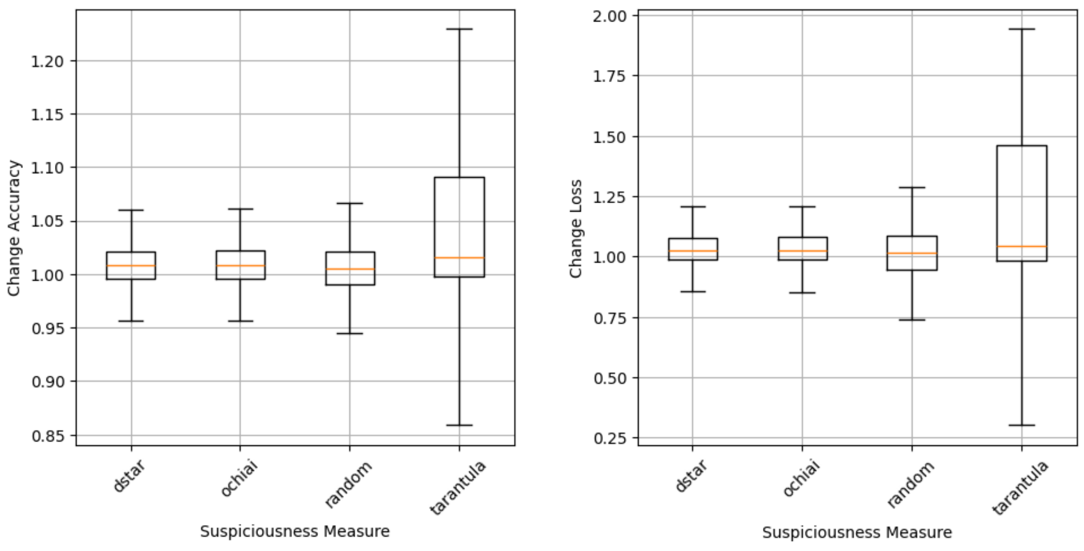


Figure 5.10.: Loss and accuracy, with training, with different suspiciousness measures

5.2.6. CNN vs. DNN Architectural Efficiency

For the non-trained runs, as seen in the figure 5.11, the CNN1 architecture is the most promising, with an improvement of up to 25% for the accuracy and 35–45% for the loss. While CNN2 is promising for the accuracy, with an improvement of up to 10%, it is the worst for the loss, with a worsening tendency. The DNN models are only performing mediocre, with a median slightly above the baseline, but only slight outliers, for both loss and accuracy. The DNN2 is slightly better than the DNN1, in both cases.

For the trained runs, as seen in the figure 5.12, the CNN1 architecture is the most promising, with an improvement of up to 15–25% for the accuracy and 50–90% for the loss. While CNN2 is mediocre for the accuracy, with an even distribution around the median at the baseline, it is the worst for the loss, with a worsening tendency. The DNN models are only performing mediocre, with a median slightly above the baseline, but only slight outliers, for both loss and accuracy.

We aren't seeing us able to draw conclusions from the results, especially the experiments should be repeated with properly designed models and different datasets.

5.2.7. Offset Variations in Loss and Accuracy

For both trained, as seen in the figure 5.13, and untrained runs, as seen in the figure 5.14, the offset doesn't have a significant impact on the loss and accuracy of the models. While seeing the same median for the offset and no offset runs, we see a larger deviation for the offset runs, for both loss and accuracy.

We would attribute this to that the offset, can lead to larger improvements, because we don't stop at local extrema in training, but also we need an even worse performance, for an epoch to break the algorithm.

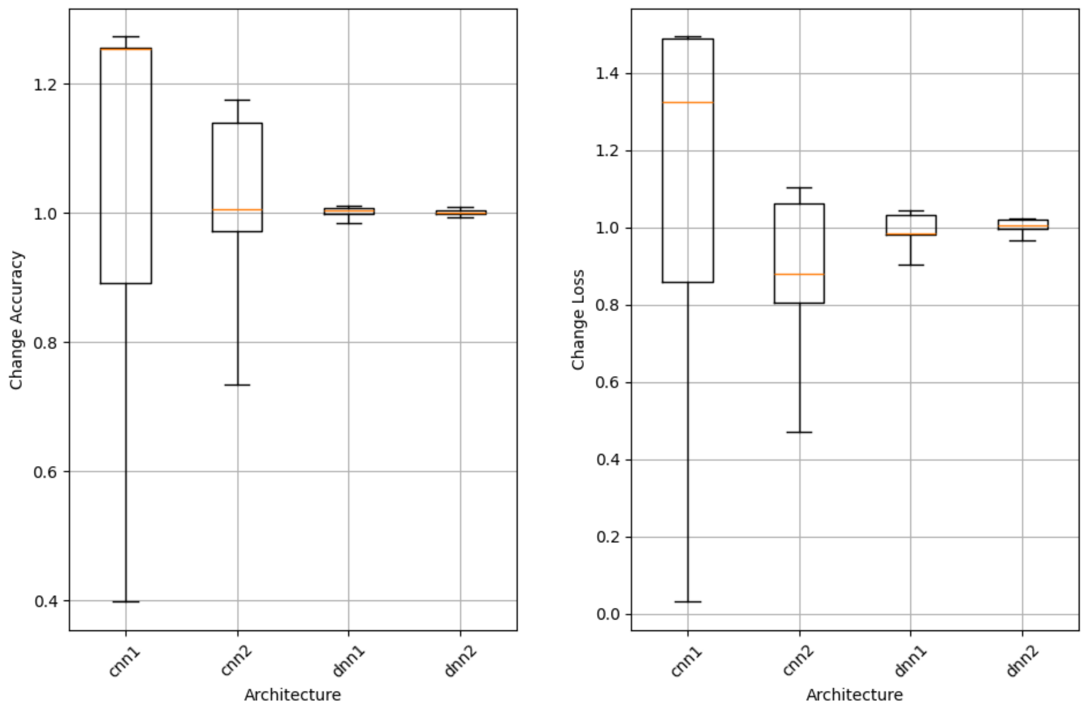


Figure 5.11.: Loss and accuracy, without training, with different architectures

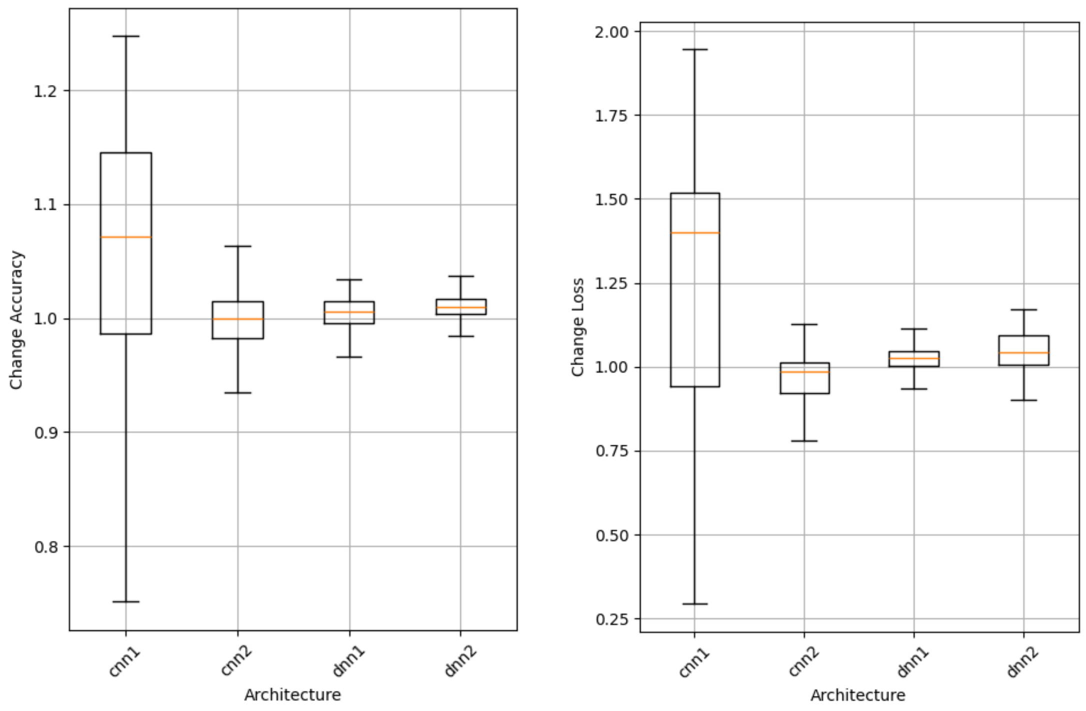


Figure 5.12.: Loss and accuracy, with training, with different architectures

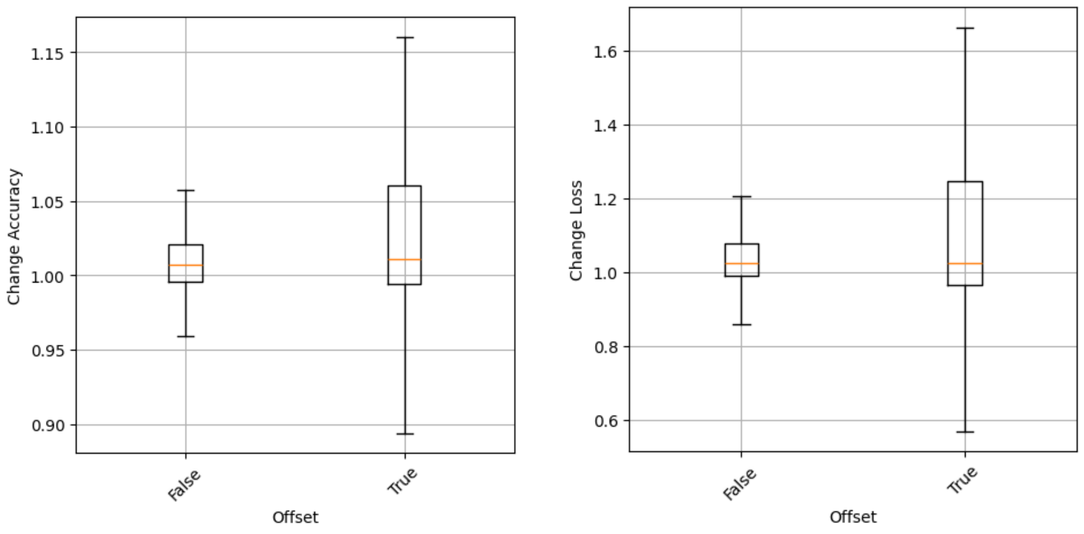


Figure 5.13.: Loss and accuracy, with training, with offsets or not

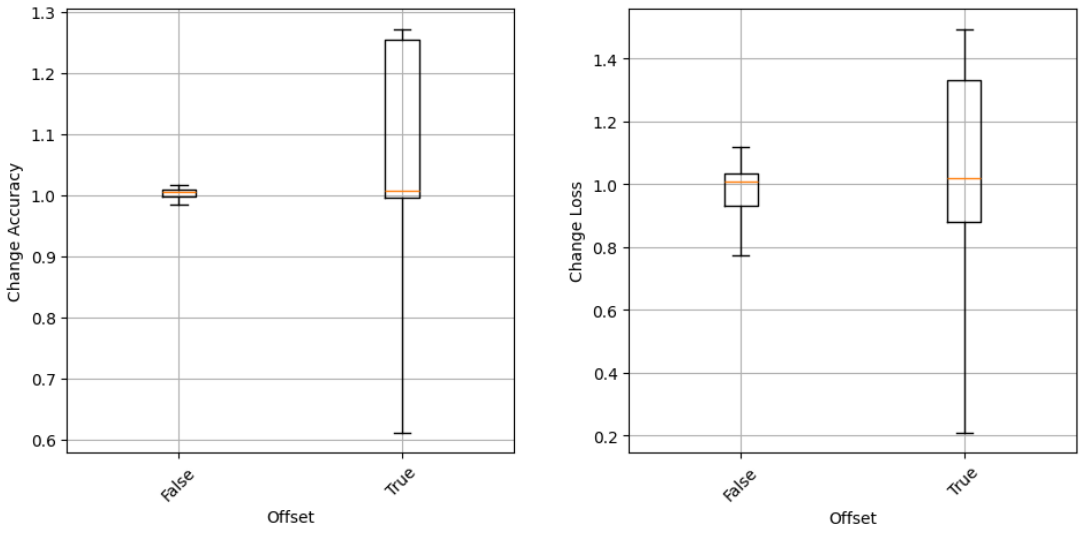
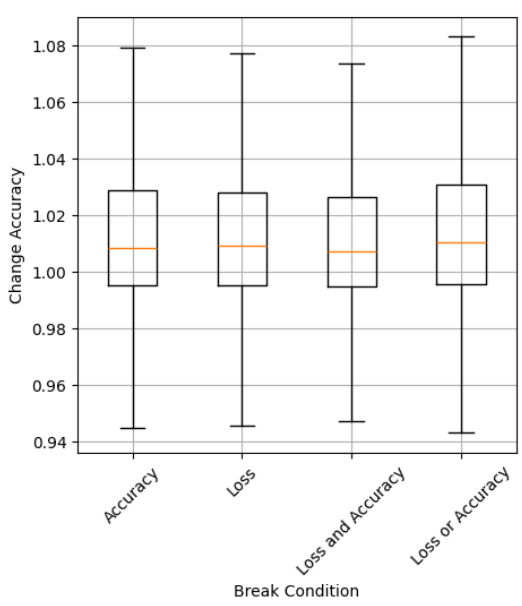


Figure 5.14.: Loss and accuracy, without training, with offsets or not



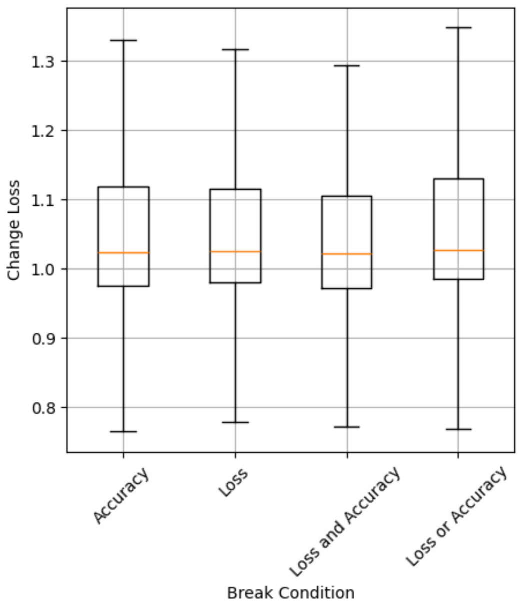


Figure 5.15.: Loss and accuracy, with training, with different break conditions

5.2.8. Break Conditions and Algorithm Performance

For the trained runs, as seen in the figure 5.15, we see no significant differences among the four options for the different break conditions. This holds true for both loss and accuracy. However, for the non-trained runs, as seen in the figure 5.16, we see some differences. Accuracy and the Accuracy and Loss break conditions seem to be the most promising. For both loss and accuracy, which surprised us a bit, because we would have expected the loss to be the most promising for the loss. We see an improvement of up to 25% for the accuracy and 30–50% for the loss, for the Accuracy and Loss break condition, and the Accuracy break condition. While the Loss break condition and Accuracy or Loss break condition are behaving similarly on the loss metric. We see a slightly better performance for the Accuracy or Loss break condition, for the accuracy metric. Which we would attribute to the fact, that the accuracy is the more important metric for the algorithm, as we are using it to decide if we are breaking the algorithm or not. While Loss or Accuracy isn't depending on the accuracy as much as the Accuracy break condition or the Accuracy and Loss break condition. Because also Loss alone can break the algorithm, the other two are depending on the accuracy to break the algorithm, while the last one is also depending on the loss.

5.2.9. Contributions of Different Mutation Functions

For the non-trained runs, as seen in the figure 5.17, we see a significant difference between the weight and bias mutation functions. The bias functions have a median pretty close to the baseline, and a near gaussian deviation around the median. The

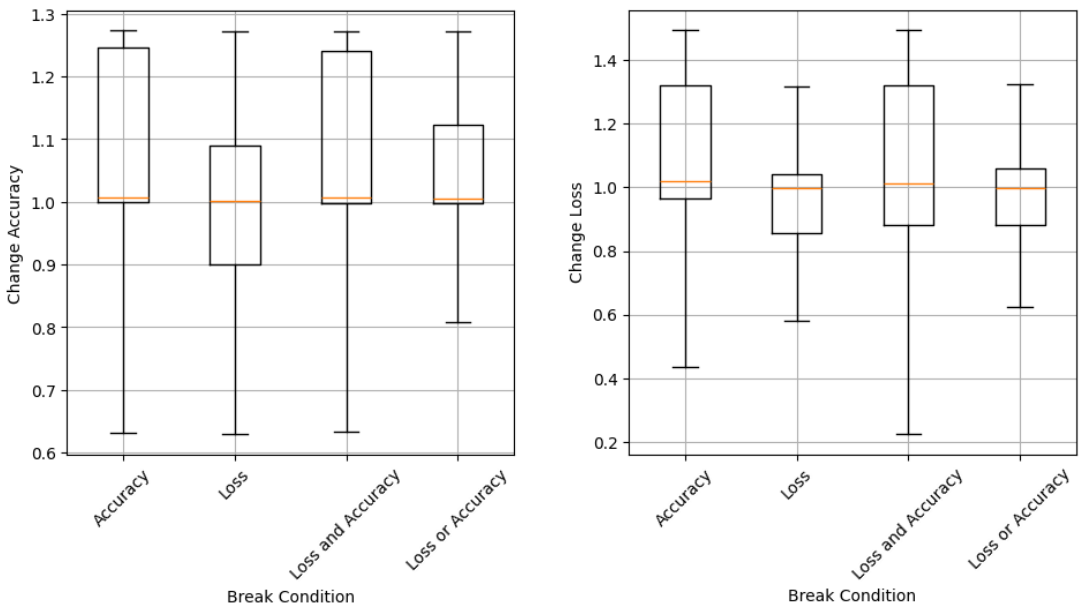


Figure 5.16.: Loss and accuracy, without training, with different break conditions

weight functions are looking much more promising, while we only see a median slightly above the baseline, we see for nearly all function an accuracy improvement of up to 25%, with only the `modify_weight_all_random_by_scalar_gauss` function performing a bit worse. For the loss we similar result, but we see one function standout, the `modify_weight_all_random_gauss` function, which is showing a higher improvement of around 1ß% in comparison to the other functions. For the trained runs, as seen in the figure 5.18, we see a similar result, but the bias functions are performing a bit better. All the functions are working similarly, but with the `modify_weight_all_random_by_scalar_gauss` function performing a bit worse. For both trained and untrained runs, we see no difference between the different mutation values, as seen in the figure 5.19 and 5.20.

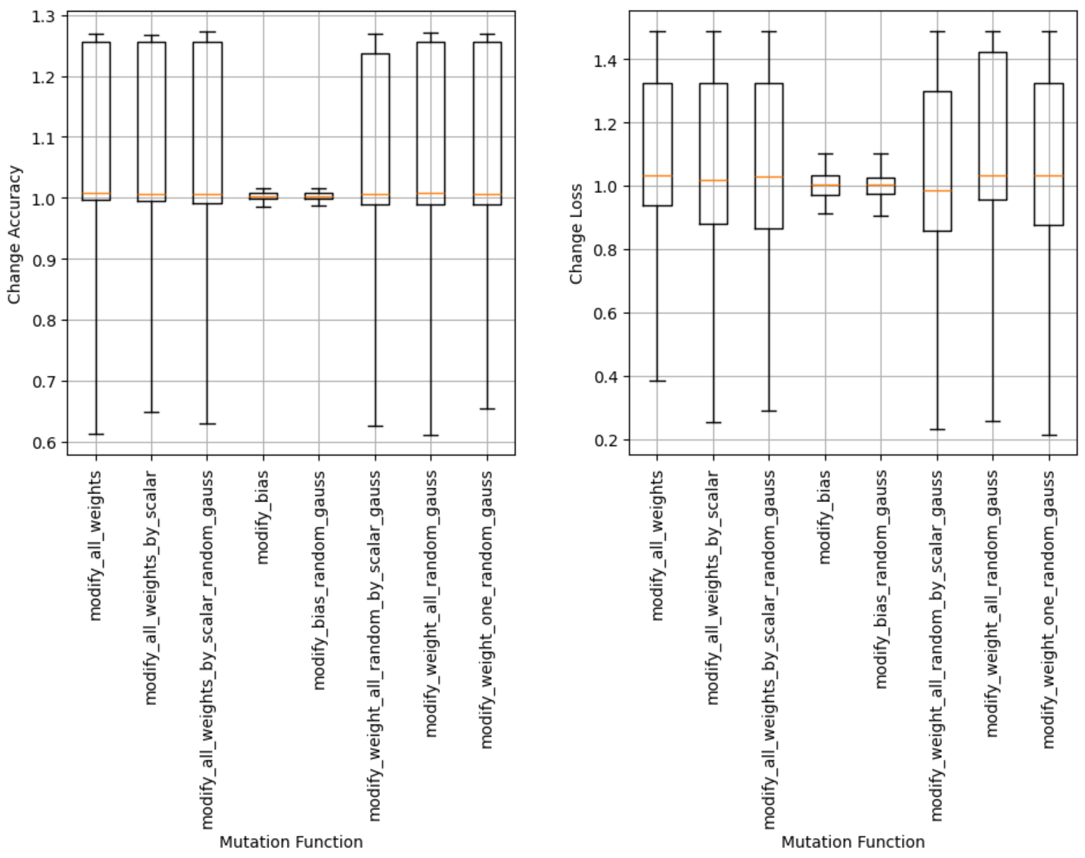


Figure 5.17.: Loss and accuracy, without training, with different mutation functions

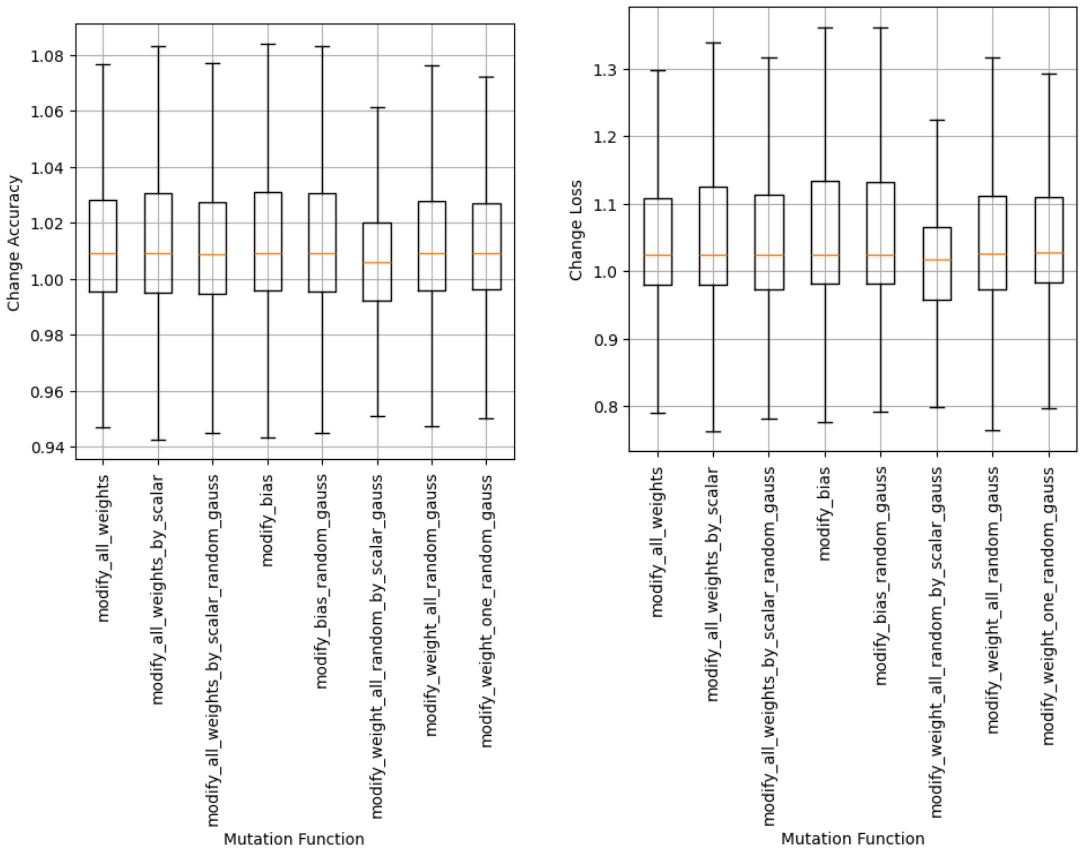


Figure 5.18.: Loss and accuracy, with training, with different mutation functions

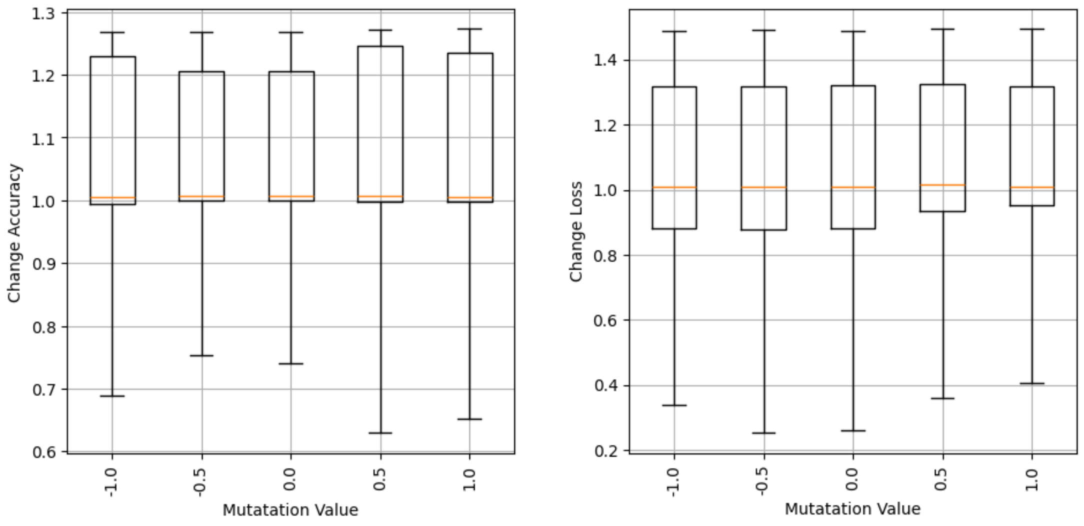


Figure 5.19.: Loss and accuracy, without training, with different mutation values

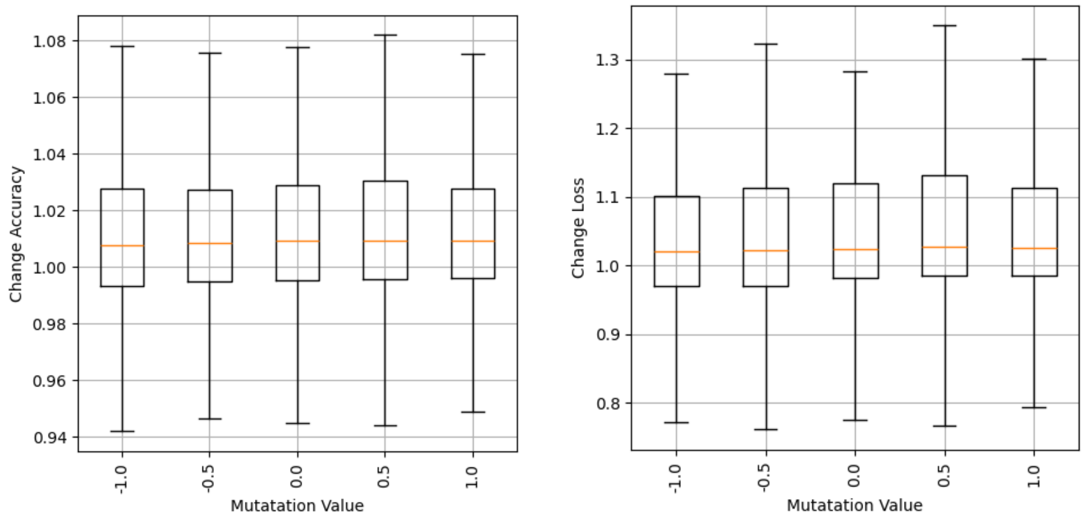


Figure 5.20.: Loss and accuracy, with training, with different mutation values

6. Conclusion and Future Work

In the previous chapters, we introduced a new method for repairing Deep Neural Networks and Convolutional Neural Networks. This method uses a spectrum-based fault localization combined with network mutations to increase network accuracy and decrease network loss. First, we analyse the network to identify any faults. We then use these findings to apply our mutation function. If desired, we can train the network for an additional epoch. Finally, we evaluate the network again. If its performance has declined, we revert to the previous network. If its performance has improved, we keep the new network and repeat the process.

During the first three epochs of our approach, we observed an increase in accuracy of up to 2–5% for the first epoch and up to 11–25% for untrained models. The first epoch showed a decrease of up to 10–40%, and the third epoch showed a decrease of up to 25–40% compared to our benchmark, for the loss. However, there was no further increase in accuracy for subsequent epochs, only a decrease.

If we further trained our models, we get an increase of up to 2.5-7.5% for the accuracy and a decrease of up to 12–35% for the loss. Here we also see a decrease after the third epoch, but we still stay over the baseline of one for the first 10–11 epochs.

The approach performs best on the full Fashion-MNIST dataset and also good on the half data set. For our suspiciousness measures, which we utilize for our fault localization, tarantula seems to be the best performing measure, for the trained models, the other measures aren't working significantly better than the random choosing of neurons. The exception is in the accuracy category for the non-trained models, the Ochiai measure. Our Approach seems to be working best on Convolutional Neural Networks, in contrast, we only see a marginal increase for Deep Neural Networks. For our break conditions, we see no difference in the trained conditions for the trained runs, but we see one for the non-trained ones, where we see the worst performance which depends more on the loss, this holds even true for the loss. Important is also the use of an Offset for both loss and accuracy. Another factor that can be mitigated is the use of the different mutation functions, where we see that the mutation functions aren't leading to significantly different results, with the exception that the weight mutation function are leading to better results than the bias targeting mutation functions.

In future research, it would be beneficial to investigate several aspects. Firstly, it is essential to evaluate the performance of the approach on larger models and more domain specific models and different datasets. This will help to determine a threshold beyond which further modification of the neural network is impossible. For instance, this threshold could be defined as a ratio between the model's modifiable parameters and the modified parameters. Another thing that could be investigated in this instance is the choosing of multiple neurons at a time for the modification.

Due to time constraints, we decided to use the static spectrum matrix for fault localization instead of investigating other critical options. A faster approach for generating the matrix should be considered. This would lead to the possibility to change a neuron, until another more suspicious neuron arises.

One element which we also introduced due to the time constraints of this thesis is the regression for the loss and accuracy offset, to get a faster release of the break condition, hence getting runs with fewer epochs. So the behaviour without the offset would be interesting.

Another possible area of investigation would be the suspiciousness measures, one point would be the usage of suspiciousness measures based on genetic programming which superiority in spectrum-based fault localisation regarding software was already shown by Yoo et al. [51] which could increase the right choosing of the neurons. Another endeavour worth exploring would be the dstar suspiciousness measure, which we used in combination with the star value of 3, hence the investigation of values different to 3 would be needed.

A different idea would be to use the spectrum-based fault localization to not only identify suspicious neurons, but to aggregate the suspiciousness of a layer and then to modify these. For example, we could change the activation function of the layer, the stride of a convolutional layer or pooling layer, the kernel/filter of a convolutional layer, or we could change the size of the pooling layer. An approach, which should be tried is the freezing of an overly trustful layer could be frozen to focus on the training of the other layers.

Or another appropriate layer could be set as a predecessor or successor. For example, a normalization layer, a dense layer or a dropout layer as a predecessor or a regularization layer as a successor. Another idea would be the encapsulation of the layer in a residual block. At last, we could add an attention layer on either side of the suspicious layer.

Another useful feature would be the automatically choosing of an appropriate fix when a condition is met in the suspicious neuron, like an exploding or vanishing gradient for example. Finally, we could modify the activation function of the specified neuron using a custom layer that applies the desired activation function to each neuron, to address things like dying ReLU.

At last, we would find it beneficial to investigate some different methods for the offset in the Algorithm, to get a more precise release of the break condition.

A. Source Code

```python
def test_model(model, X_test, Y_test):
    # Find activations of each neuron in each layer for each input x in
    X_test
    layer_outs = an.get_layer_outs(model, X_test)

    # Print information about the model
    print(model.summary())

    # Evaluate the model
    score = model.evaluate(X_test, np.argmax(Y_test, axis=1), verbose=0)
    print('[loss, accuracy] -> ' + str(score))

    # Make predictions
    Y_pred = model.predict(X_test)

    # Calculate classification report and confusion matrix
    an.calculate_prediction_metrics(Y_test, Y_pred, score)

    # Find test and prediction classes
    expectations = np.argmax(Y_test, axis=1)
    predictions = np.argmax(Y_pred, axis=1)

    classifications = np.absolute(expectations - predictions)

    # Find correct classifications and misclassifications
    correct_classifications = []
    misclassifications = []
    for i in range(0, len(classifications)):
        if classifications[i] == 0:
            correct_classifications.append(i)
        else:
            misclassifications.append(i)

    print("Testing done!\n")

    return correct_classifications, misclassifications, layer_outs,
    predictions
```

Listing A.1: Testing Network, for correct and incorrect classifications

```python
def get_trainable_layers(model):
    trainable_layers = []
    for layer in model.layers:
        try:
            weights = layer.get_weights()[0]
            trainable_layers.append(model.layers.index(layer))
        except:
            pass

    trainable_layers = trainable_layers[:-1]  # ignore the output layer

    return trainable_layers
```

Listing A.2: Testing if layer is trainable

```
def scores_with_foo(trainable_layers, scores, num_cf, num_uf, num_cs,
    num_us, suspicious_num, foo):

    for i in range(len(scores)):
        for j in range(len(scores[i])):
            score = foo(i, j)
            if np.isnan(score):
                score = 0
            scores[i][j] = score

    flat_scores = [float(item) for sublist in scores for item in sublist
    if not math.isnan(float(item))]

    # grab the indexes of the highest suspicious_num scores
    flat_indexes = np.argsort(flat_scores)[::-1][:suspicious_num]

    suspicious_neuron_idx = []
    for idx in flat_indexes:
        # unflatten idx
        i = 0
        accum = idx
        while accum >= len(scores[i]):
            accum -= len(scores[i])
            i += 1
        j = accum

        if trainable_layers is None:
            suspicious_neuron_idx.append((i, j))
        else:
            suspicious_neuron_idx.append((trainable_layers[i], j))

    return suspicious_neuron_idx


def tarantula_analysis(trainable_layers, scores, num_cf, num_uf, num_cs,
    num_us, suspicious_num):
    def tarantula(i, j):
        return float(float(num_cf[i][j]) / (num_cf[i][j] + num_uf[i][j]))
     / \
            (float(num_cf[i][j]) / (num_cf[i][j] + num_uf[i][j]) + float(
    num_cs[i][j]) / (num_cs[i][j] + num_us[i][j]))

    return scores_with_foo(trainable_layers, scores, num_cf, num_uf,
    num_cs, num_us, suspicious_num, tarantula)


def ochiai_analysis(trainable_layers, scores, num_cf, num_uf, num_cs,
    num_us, suspicious_num):
    def ochiai(i, j):
        return float(num_cf[i][j]) / ((num_cf[i][j] + num_uf[i][j]) * (
    num_cf[i][j] + num_cs[i][j])) **(.5)

    return scores_with_foo(trainable_layers, scores, num_cf, num_uf,
    num_cs, num_us, suspicious_num, ochiai)


def dstar_analysis(trainable_layers, scores, num_cf, num_uf, num_cs,
    num_us, suspicious_num, star):
    def dstar(i, j):
        return float(num_cf[i][j]**star) / (num_cs[i][j] + num_uf[i][j])

    return scores_with_foo(trainable_layers, scores, num_cf, num_uf,
    num_cs, num_us, suspicious_num, dstar)
```

Listing A.3: Similarity Coefficients and Ranking

```
def load_classifications(filename, group_index=1):
    filename = filename + '_classifications.h5'
    try:
        with h5py.File(filename, 'r') as hf:
            group = hf.get('group' + str(group_index))
            correct_classifications = group['correct_classifications'][:]
            misclassifications = group['misclassifications'][:]

            print("Classifications loaded from ", filename)
            return correct_classifications, misclassifications
    except (IOError) as error:
        print("Could not open file: ", filename)
        sys.exit(-1)


def save_classifications(correct_classifications, misclassifications,
    filename, group_index=1):
    filename = filename + '_classifications.h5'
    with h5py.File(filename, 'a') as hf:
        group = hf.create_group('group' + str(group_index))
        group.create_dataset("correct_classifications", data=
    correct_classifications)
        group.create_dataset("misclassifications", data=
    misclassifications)

    print("Classifications saved in ", filename)
    return


def load_layer_outs(filename, group_index=1):
    filename = filename + '_layer_outs.h5'
    layer_outs = []

    try:
        with h5py.File(filename, 'r') as hf:
            group = hf.get('group' + str(group_index))
            if group is not None:
                i = 0
```

```python
                while f'layer_outs_{i}' in group:
                    layer_outs.append(group[f'layer_outs_{i}'][:])
                    i += 1
            else:
                print(f"Group 'group{group_index}' not found in file: {filename}")
                return None
            return layer_outs
    except IOError as error:
        print("Could not open file: ", filename)
        traceback.print_exc()
        sys.exit(-1)
    except KeyError as error:
        # This exception will be caught when the dataset doesn't exist
        print("Layer outs loaded from ", filename)
        return layer_outs


def save_layer_outs(layer_outs, filename, group_index=1):
    filename = filename + '_layer_outs.h5'
    with h5py.File(filename, 'a') as hf:
        group = hf.create_group('group' + str(group_index))
        for i in range(len(layer_outs)):
            group.create_dataset("layer_outs_" + str(i), data=layer_outs[i])

    print("Layer outs saved in ", filename)
    return
def save_suspicious_neurons(suspicious_neurons, filename, approach, susp_num, group_index=1):
    filename = filename + '_' + approach + '_SN' + str(susp_num) + '_suspicious_neurons.h5'
    with h5py.File(filename, 'a') as hf:
        group = hf.create_group('group' + str(group_index))
        # for i in range(len(suspicious_neurons)):
        # group.create_dataset("suspicious_neurons_" + str(i), data=suspicious_neurons[i])
        for i, tuple_data in enumerate(suspicious_neurons):
            list_data = list(tuple_data)
            group.create_dataset(f'suspicious_neurons_{i}', data=list_data)

    print("Suspicious neurons saved in ", filename)
    return


def load_suspicious_neurons(filename, approach, susp_num, group_index=1):
    filename = filename + '_' + approach + '_SN' + str(susp_num) + '_suspicious_neurons.h5'
    try:
        with h5py.File(filename, 'r') as hf:
            group = hf.get('group' + str(group_index))
            i = 0
            suspicious_neurons = []
            # while f'suspicious_neurons_{i}' in group:
            #    suspicious_neurons.append(group[f'suspicious_neurons_{i}'][:])
            #    i += 1
            while f'suspicious_neurons_{i}' in group:
                suspicious_neurons.append(tuple(group[f'suspicious_neurons_{i}'][:]))
                i += 1
            print("Suspicious neurons loaded from ", filename)
            return suspicious_neurons
    except IOError as error:
        print("Could not open file: ", filename)
        sys.exit(-1)
def load_spectrum_matrices(filename, group_index=1):
    filename = filename + '_spectrum_matrices.h5'
    scores, num_cf, num_uf, num_cs, num_us = [], [], [], [], []

    try:
        with h5py.File(filename, 'r') as hf:
            group = hf['group' + str(group_index)]
            if group is not None:
                i = 0
                while f'scores_{i}' in group:
                    scores.append(group[f'scores_{i}'][:])
                    num_cf.append(group[f'num_cf_{i}'][:])
                    num_cs.append(group[f'num_cs_{i}'][:])
                    num_uf.append(group[f'num_uf_{i}'][:])
                    num_us.append(group[f'num_us_{i}'][:])
                    i += 1
            else:
                print(f"Group 'group{group_index}' not found in file: {filename}")
                return None
            return scores, num_cf, num_uf, num_cs, num_us
    except IOError as error:
        print("Could not open file: ", filename)
        traceback.print_exc()
        sys.exit(-1)
    except KeyError as error:
        # This exception will be caught when the dataset doesn't exist
        print("Layer outs loaded from ", filename)
        return scores, num_cf, num_uf, num_cs, num_us


def save_spectrum_matrices(scores, num_cf, num_uf, num_cs, num_us, filename, group_index=1):
    filename = filename + '_spectrum_matrices.h5'
    with h5py.File(filename, 'a') as hf:
        group = hf.create_group('group' + str(group_index))
        for i in range(len(scores)):
            group.create_dataset(f'scores_{i}', data=scores[i])
            group.create_dataset(f'num_cf_{i}', data=num_cf[i])
            group.create_dataset(f'num_cs_{i}', data=num_cs[i])
            group.create_dataset(f'num_uf_{i}', data=num_uf[i])
            group.create_dataset(f'num_us_{i}', data=num_us[i])
    print("Spectrum matrices saved in ", filename)
    return
```

Listing A.4: All Saving and loading data functions

```
def run_modification_algorithm(run, modelname, train_images, train_labels
    , test_images, test_labels, analysis_approach,
                               mutation_function, old_loss, old_accuracy,
     train_between_iterations=False, value=0,
                               max_num_iterations=-1,
                               loss_offset=0, accuracy_offset=0,
    regression_loss_offset=False,
                               regression_accuracy_offset=False,
    compare_loss=False, compare_accuracy=True,
                               compare_and_both=False):
    excel_file_path = "/mnt/c/Users/lenna/PycharmProjects/BA_Thesis/data/
    " + modelname + "_results.xlsx"
    df_exists = os.path.isfile(excel_file_path)
    df = pd.read_excel(excel_file_path) if df_exists else pd.DataFrame(
        columns=['run', 'epoch', 'approach_analysis', '
    trained_between_iterations', 'regression_loss_offset',
                 'regression_accuracy_offset', 'compare_loss', '
    compare_accuracy', 'compare_and_both',
                 'max_num_iterations',
                 'mutation_function', 'value', 'old_accuracy', '
    new_accuracy', 'accuracy_offset', 'old_loss',
                 'new_loss', 'loss_offset'])
    tf.random.set_seed(42)
    model = data_managment.get_model(modelname)
    suspicous_neurons = nn_analysis.run_analysis(modelname,
    analysis_approach,test_images,test_labels,max_num_iterations)
    epoch = 0
    data_to_append = []

    while len(suspicous_neurons) > 0:
        mutation_function(model, suspicous_neurons.pop(), value)
        model.compile(optimizer='adam', loss='
    sparse_categorical_crossentropy', metrics=['accuracy'])
        if train_between_iterations:
            model.fit(train_images, train_labels, epochs=1)
        new_loss, new_accuracy = model.evaluate(test_images, test_labels)
        data_to_append.append({'run': run, 'epoch': epoch, '
    approach_analysis': analysis_approach,
                               'trained_between_iterations':
    train_between_iterations,
                               'regression_loss_offset':
    regression_loss_offset,
                               'regression_accuracy_offset':
    regression_accuracy_offset, 'compare_loss': compare_loss,
                               'compare_accuracy': compare_accuracy, '
    compare_and_both': compare_and_both,
                               'max_num_iterations': max_num_iterations,
                               'mutation_function': mutation_function.
__name__,
                                    'value': value, 'old_accuracy': old_accuracy, 'new_accuracy': new_accuracy,
                                    'accuracy_offset': accuracy_offset, 'old_loss': old_loss, 'new_loss': new_loss,
                                    'loss_offset': loss_offset})
        if compare_and_both:
            if (new_loss + loss_offset) > old_loss and (new_accuracy + accuracy_offset) < old_accuracy: break
        else:
            if compare_loss and (new_loss + loss_offset) > old_loss: break
            if compare_accuracy and (new_accuracy + accuracy_offset) < old_accuracy: break
        if regression_loss_offset: loss_offset /= 2
        if regression_accuracy_offset: accuracy_offset /= 2
        epoch += 1
        old_accuracy = new_accuracy
        old_loss = new_loss

    df = pd.concat([df, pd.DataFrame(data_to_append)], ignore_index=True)
    df.to_excel(excel_file_path, index=False)
    return model
```

Listing A.5: Full Modification Algorithm

```
def test_model(model, X_test, Y_test):
    # Find activations of each neuron in each layer for each input x in X_test
    layer_outs = an.get_layer_outs(model, X_test)

    # Print information about the model
    print(model.summary())

    # Evaluate the model
    score = model.evaluate(X_test, np.argmax(Y_test, axis=1), verbose=0)
    print('[loss, accuracy] -> ' + str(score))

    # Make predictions
    Y_pred = model.predict(X_test)

    # Calculate classification report and confusion matrix
    an.calculate_prediction_metrics(Y_test, Y_pred, score)

    # Find test and prediction classes
    expectations = np.argmax(Y_test, axis=1)
    predictions = np.argmax(Y_pred, axis=1)

    classifications = np.absolute(expectations - predictions)

    # Find correct classifications and misclassifications
    correct_classifications = []
    misclassifications = []
    for i in range(0, len(classifications)):
```

```python
        if classifications[i] == 0:
            correct_classifications.append(i)
        else:
            misclassifications.append(i)

    print("Testing done!\n")

    return correct_classifications, misclassifications, layer_outs, predictions


def get_trainable_layers(model):
    trainable_layers = []
    for layer in model.layers:
        try:
            weights = layer.get_weights()[0]
            trainable_layers.append(model.layers.index(layer))
        except:
            pass

    trainable_layers = trainable_layers[:-1]  # ignore the output layer

    return trainable_layers

def contruct_spectrum_matrices(model, trainable_layers, correct_classifications, misclassifications,
                               layer_outs, activation_threshold=0):
    start_time = time.time()
    scores = []
    activated_fail = []  # num_cf
    activated_success = []  # num_cs
    not_activated_fail = []  # num_uf
    not_activated_success = []  # num_us

    for layer in trainable_layers:
        shape = model.layers[layer].output_shape[-1]
        activated_fail.append(np.zeros(shape))
        activated_success.append(np.zeros(shape))
        not_activated_fail.append(np.zeros(shape))
        not_activated_success.append(np.zeros(shape))
        scores.append(np.zeros(shape))

    for layer, layer_index in zip(trainable_layers, range(len(trainable_layers))):
        for test_index, outlayer in enumerate(layer_outs[layer][0]):
            for neuron_index in range(model.layers[layer].output_shape[-1]):
                mean_out = np.mean(outlayer[..., neuron_index])
                if test_index in correct_classifications:
                    if mean_out > activation_threshold:
                        activated_success[layer_index][neuron_index] += 1
                    else:
                        not_activated_success[layer_index][neuron_index] += 1
                elif test_index in misclassifications:
                    if mean_out > activation_threshold:
                        activated_fail[layer_index][neuron_index] += 1
                    else:
                        not_activated_fail[layer_index][neuron_index] += 1

    end_time = time.time()
    print("Time to construct spectrum matrices: ", end_time - start_time)
    return scores, activated_fail, not_activated_fail, activated_success, not_activated_success
```

Listing A.6: Network Testing Functions

Bibliography

[1] Mariusz Bojarski et al. *End to End Learning for Self-Driving Cars.* Apr. 25, 2016. arXiv: `1604.07316[cs]`. URL: `http://arxiv.org/abs/1604.07316` (visited on 02/28/2024).

[2] Y-Lan Boureau, Jean Ponce, and Yann LeCun. "A theoretical analysis of feature pooling in visual recognition". In: *Proceedings of the 27th international conference on machine learning (ICML-10).* 2010, pp. 111–118.

[3] *Build from source on Windows | TensorFlow.* Nov. 21, 2023. URL: `https://www.tensorflow.org/install/source_windows` (visited on 01/11/2024).

[4] Davide Li Calsi et al. "Distributed Repair of Deep Neural Networks". In: *2023 IEEE Conference on Software Testing, Verification and Validation (ICST).* 2023, pp. 83–94. DOI: `10.1109/ICST57152.2023.00017`.

[5] Dawei Cheng et al. "Manifesting Bugs in Machine Learning Code: An Explorative Study with Mutation Testing". In: *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS).* 2018 IEEE International Conference on Software Quality, Reliability and Security (QRS). Lisbon: IEEE, July 2018, pp. 313–324. ISBN: 978-1-5386-7757-5. DOI: `10.1109/QRS.2018.00044`.

[6] François Chollet et al. *Keras.* 2015. URL: `https://keras.io`.

[7] Dan Cireşan, Ueli Meier, and Juergen Schmidhuber. *Multi-column Deep Neural Networks for Image Classification.* Feb. 13, 2012. arXiv: `1202.2745[cs]`. URL: `http://arxiv.org/abs/1202.2745` (visited on 02/28/2024).

[8] Andrew Collette et al. *h5py/h5py: 3.7.0.* Version 3.7.0. May 24, 2022. DOI: `10.5281/ZENODO.6575970`.

[9] Jacob Devlin et al. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding.* May 24, 2019. arXiv: `1810.04805[cs]`. URL: `http://arxiv.org/abs/1810.04805` (visited on 12/29/2023).

[10] Hasan Ferit Eniser. *DeepFault: Fault Localization for Deep Neural Networks.* original-date: 2018-07-04T12:51:18Z. July 31, 2023. URL: `https://github.com/hfeniser/DeepFault` (visited on 10/08/2023).

[11] Hasan Ferit Eniser, Simos Gerasimou, and Alper Sen. "DeepFault: Fault Localization for Deep Neural Networks". In: (2019). Publisher: arXiv Version Number: 1. DOI: `10.48550/ARXIV.1902.05974`.

[12] Kunihiko Fukushima. "Cognitron: A self-organizing multilayered neural network". In: *Biol. Cybernetics* 20.3 (1975), pp. 121–136. ISSN: 0340-1200, 1432-0770. DOI: `10.1007/BF00342633`.

[13] Xiang Gao et al. “Fuzz testing based data augmentation to improve robustness of deep neural networks”. In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering.* ICSE ’20: 42nd International Conference on Software Engineering. Seoul South Korea: ACM, June 27, 2020, pp. 1147–1158. ISBN: 978-1-4503-7121-6. DOI: 10.1145/3377811.3380415.

[14] Xavier Glorot and Yoshua Bengio. “Understanding the difficulty of training deep feedforward neural networks”. In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics.* Vol. 9. Proceedings of Machine Learning Research. Chia Laguna Resort, Sardinia, Italy: PMLR, May 13, 2010, pp. 249–256. URL: https://proceedings.mlr.press/v9/glorot10a.html.

[15] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. “Deep Sparse Rectifier Neural Networks”. In: *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics.* Vol. 15. Proceedings of Machine Learning Research. Fort Lauderdale, FL, USA: PMLR, Apr. 11, 2011, pp. 315–323. URL: https://proceedings.mlr.press/v15/glorot11a.html.

[16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning.* Adaptive computation and machine learning. Cambridge, Massachusetts: The MIT Press, 2016. 775 pp. ISBN: 978-0-262-03561-3.

[17] Kelly J. Hayhurst et al. *A Practical Tutorial on Modified Condition/Decision Coverage.* L-18088. NTRS Author Affiliations: NASA Langley Research Center, Rockwell Collins, Inc., Boeing Co., Federal Aviation Administration NTRS Document ID: 20010057789 NTRS Research Center: Langley Research Center (LaRC). May 1, 2001. URL: https://ntrs.nasa.gov/citations/20010057789 (visited on 01/04/2024).

[18] Dan Hendrycks and Kevin Gimpel. “Gaussian Error Linear Units (GELUs)”. In: (2016). Publisher: arXiv Version Number: 5. DOI: 10.48550/ARXIV.1606.08415.

[19] Geoffrey Hinton et al. “Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups”. In: *IEEE Signal Process. Mag.* 29.6 (Nov. 2012), pp. 82–97. ISSN: 1053-5888. DOI: 10.1109/MSP.2012.2205597.

[20] Hua Jie Lee, Lee Naish, and Kotagiri Ramamohanarao. “Study of the relationship of bug consistency with respect to performance of spectra metrics”. In: *2009 2nd IEEE International Conference on Computer Science and Information Technology.* 2009 2nd IEEE International Conference on Computer Science and Information Technology. Beijing, China: IEEE, 2009, pp. 501–508. ISBN: 978-1-4244-4519-6. DOI: 10.1109/ICCSIT.2009.5234512.

[21] Xiaowei Huang et al. “A survey of safety and trustworthiness of deep neural networks: Verification, testing, adversarial attack and defence, and interpretability”. In: *Computer Science Review* 37 (Aug. 2020), p. 100270. ISSN: 15740137. DOI: 10.1016/j.cosrev.2020.100270.

[22] Viren Jain et al. “Supervised Learning of Image Restoration with Convolutional Networks”. In: *2007 IEEE 11th International Conference on Computer Vision.* 2007 IEEE 11th International Conference on Computer Vision. Rio de Janeiro, Brazil: IEEE, 2007, pp. 1–8. ISBN: 978-1-4244-1630-1. DOI: 10.1109/ICCV.2007.4408909.

[23] Yu-Gang Jiang et al. “Exploiting Feature and Class Relationships in Video Categorization with Regularized Deep Neural Networks”. In: *IEEE Trans. Pattern Anal. Mach. Intell.* 40.2 (Feb. 1, 2018), pp. 352–364. ISSN: 0162-8828, 2160-9292. DOI: 10.1109/TPAMI.2017.2670560.

[24] James A. Jones and Mary Jean Harrold. “Empirical evaluation of the tarantula automatic fault-localization technique”. In: *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering.* ASE05: International Conference on Automated Software Engineering 2005. Long Beach CA USA: ACM, Nov. 7, 2005, pp. 273–282. ISBN: 978-1-58113-993-8. DOI: 10.1145/1101908.1101949.

[25] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization.* Jan. 29, 2017. arXiv: 1412.6980[cs]. URL: http://arxiv.org/abs/1412.6980 (visited on 12/27/2023).

[26] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Advances in Neural Information Processing Systems.* Vol. 25. Curran Associates, Inc., 2012. URL: https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[27] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. “Deep learning”. In: *Nature* 521.7553 (May 28, 2015), pp. 436–444. ISSN: 0028-0836, 1476-4687. DOI: 10.1038/nature14539.

[28] Yann LeCun et al. “Backpropagation Applied to Handwritten Zip Code Recognition”. In: *Neural Computation* 1.4 (Dec. 1989), pp. 541–551. ISSN: 0899-7667, 1530-888X. DOI: 10.1162/neco.1989.1.4.541.

[29] Davide Li Calsi et al. “Adaptive Search-based Repair of Deep Neural Networks”. In: *Proceedings of the Genetic and Evolutionary Computation Conference.* GECCO ’23: Genetic and Evolutionary Computation Conference. Lisbon Portugal: ACM, July 15, 2023, pp. 1527–1536. ISBN: 9798400701191. DOI: 10.1145/3583131.3590477.

[30] Geert Litjens et al. “A survey on deep learning in medical image analysis”. In: *Medical Image Analysis* 42 (Dec. 2017), pp. 60–88. ISSN: 13618415. DOI: 10.1016/j.media.2017.07.005.

[31] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems.* 2015. URL: https://www.tensorflow.org/.

[32] Takao Nakagawa et al. "An Experience Report on Regression-Free Repair of Deep Neural Network Model". In: *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). Taipa, Macao: IEEE, Mar. 2023, pp. 778–782. ISBN: 978-1-66545-278-6. DOI: `10.1109/SANER56733.2023.00090`.

[33] Akira Ochiai. "Zoogeographical Studies on the Soleoid Fishes found in Japan and its Neighbouring Regions-I". In: *NIPPON SUISAN GAKKAISHI* 22.9 (1957), pp. 522–525. ISSN: 1349-998X, 0021-5392. DOI: `10.2331/suisan.22.522`. (Visited on 10/09/2023).

[34] Saeed Parsa. *Software testing automation: testability evaluation, refactoring, test data generation and fault localization.* OCLC: 1374251026. Cham: Springer, 2023. ISBN: 978-3-031-22057-9.

[35] Fabian Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[36] Kexin Pei et al. "DeepXplore: Automated Whitebox Testing of Deep Learning Systems". In: *Proceedings of the 26th Symposium on Operating Systems Principles.* Oct. 14, 2017, pp. 1–18. DOI: `10.1145/3132747.3132785`. arXiv: `1705.06640[cs]`.

[37] *Release Keras 2.3.0 · keras-team/keras.* GitHub. URL: `https://github.com/keras-team/keras/releases/tag/2.3.0` (visited on 03/01/2024).

[38] Janosh Riebesell. *Convolution Operator.* Apr. 9, 2022. URL: `https://tikz.net/conv2d/` (visited on 12/30/2023).

[39] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning representations by back-propagating errors". In: *Nature* 323.6088 (Oct. 1986), pp. 533–536. ISSN: 0028-0836, 1476-4687. DOI: `10.1038/323533a0`.

[40] Jeongju Sohn, Sungmin Kang, and Shin Yoo. "Arachne: Search-Based Repair of Deep Neural Networks". In: *ACM Trans. Softw. Eng. Methodol.* 32.4 (Oct. 31, 2023), pp. 1–26. ISSN: 1049-331X, 1557-7392. DOI: `10.1145/3563210`.

[41] Youcheng Sun et al. "DeepConcolic: Testing and Debugging Deep Neural Networks". In: *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion).* 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion). Montreal, QC, Canada: IEEE, May 2019, pp. 111–114. ISBN: 978-1-72811-764-5. DOI: `10.1109/ICSE-Companion.2019.00051`.

[42] Christian Szegedy et al. "Going Deeper with Convolutions". In: (2014). Publisher: arXiv Version Number: 1. DOI: `10.48550/ARXIV.1409.4842`.

[43] *tf.keras.layers.Dense | TensorFlow v2.14.0.* Sept. 27, 2023. URL: `https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dense` (visited on 12/29/2023).

[44] Shogo Tokui et al. *NeuRecover: Regression-Controlled Repair of Deep Neural Networks with Training History.* Mar. 4, 2022. arXiv: `2203.00191[cs]`. URL: `http://arxiv.org/abs/2203.00191` (visited on 02/25/2024).

[45] Mohammad Wardat et al. *DeepDiagnosis: Automatically Diagnosing Faults and Recommending Actionable Fixes in Deep Learning Programs.* Dec. 7, 2021. arXiv: `2112.04036[cs]`. URL: `http://arxiv.org/abs/2112.04036` (visited on 02/26/2024).

[46] 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.* Vol. 10805. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, pp. 408–426. ISBN: 978-3-319-89959-6 978-3-319-89960-2. DOI: `10.1007/978-3-319-89960-2_22`.

[47] W. Eric Wong and T. H. Tse, eds. *Handbook of software fault localization: foundations and advances.* OCLC: 1345466980. Hoboken, New Jersey, Piscataway, NJ: John Wiley & Sons, Inc ; IEEE Press, 2023. ISBN: 978-1-119-88092-9.

[48] W. Eric Wong et al. "A Survey on Software Fault Localization". In: *IIEEE Trans. Software Eng.* 42.8 (Aug. 1, 2016), pp. 707–740. ISSN: 0098-5589, 1939-3520. DOI: `10.1109/TSE.2016.2521368`.

[49] W. Eric Wong et al. "The DStar Method for Effective Software Fault Localization". In: *IEEE Trans. Rel.* 63.1 (Mar. 2014), pp. 290–308. ISSN: 0018-9529, 1558-1721. DOI: `10.1109/TR.2013.2285319`.

[50] Han Xiao, Kashif Rasul, and Roland Vollgraf. "Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms". In: (2017). Publisher: arXiv Version Number: 2. DOI: `10.48550/ARXIV.1708.07747`. (Visited on 10/10/2023).

[51] Shin Yoo et al. "Human Competitiveness of Genetic Programming in Spectrum-Based Fault Localisation: Theoretical and Empirical Analysis". In: *ACM Trans. Softw. Eng. Methodol.* 26.1 (Jan. 31, 2017), pp. 1–30. ISSN: 1049-331X, 1557-7392. DOI: `10.1145/3078840`.

[52] Bing Yu et al. "*DeepRepair:* Style-Guided Repairing for Deep Neural Networks in the Real-World Operational Environment". In: *IEEE Trans. Rel.* 71.4 (Dec. 2022), pp. 1401–1416. ISSN: 0018-9529, 1558-1721. DOI: `10.1109/TR.2021.3096332`.

[53] Hao Zhang and Wing-Kwong Chan. "Apricot: A Weight-Adaptation Approach to Fixing Deep Learning Models". In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE).* 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). San Diego, CA, USA: IEEE, Nov. 2019, pp. 376–387. ISBN: 978-1-72812-508-4. DOI: `10.1109/ASE.2019.00043`.

[54] Xiaoyu Zhang et al. “AUTOTRAINER: An Automatic DNN Training Problem Detection and Repair System”. In: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). Madrid, ES: IEEE, May 2021, pp. 359–371. ISBN: 978-1-66540-296-5. DOI: `10.1109/ICSE43902.2021.00043`.