Post-Training Mutation Tool for Deep Learning Systems

Master's Thesis submitted to the
Faculty of Informatics of the *Università della Svizzera Italiana*
in partial fulfillment of the requirements for the degree of
Master of Science in Informatics
Major in Artificial Intelligence

presented by
Filippo Casari

under the supervision of
Prof. Paolo Tonella
co-supervised by
Dr. Nargiz Humbatova, Prof. Gunel Jahangirova

January 2024

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Filippo Casari
Lugano, 17th January 2024

To my beloved

If at first you don't succeed; call it version 1.0 ...

Someone

Abstract

Mutation testing has become a popular technique in software engineering. It involves introducing small syntactic errors into systems to create their mutants. This method is especially useful for assessing test suites' ability to detect faults in different software applications. In traditional software systems, decision logic is typically implemented using source code by software developers. In contrast, the behavior of Deep Learning (DL) systems is primarily determined by the training program and the training dataset. These two components are the primary causes of errors in DL systems.

Consequently, this distinction necessitates a specialized approach to mutation testing for DL systems, tailored to address the unique challenges and characteristics inherent in their design and functionality. Currently, two specialized tools stand out as the best in the field of mutation testing on DL systems. The first tool operates in a pre-training mode, injecting faults into the system before training, which results in significant computational demands. In contrast, the second tool functions in a post-training mode, introducing random faults that are improbable in real-world scenarios.

The main goal of this project was to develop a post-training mutation tool for Deep Learning Systems. This tool is designed to address the drawbacks of previous methodologies by implementing more intelligent and efficient mutation techniques on pre-trained neural networks. In this project, we created mutants that imitate real-world situations where models are trained on low-quality data or struggle to learn a specific type of input. To achieve this, we aimed to simulate these scenarios by inducing already trained models (post-training) to misbehave for a particular input data that share certain features. In order to achieve this, we first reduced and clustered input data and then created meaningful mutants that misbehaved on a particular cluster. For this purpose, we implemented two novel approaches to create mutants to address and overcome the drawbacks of current methods.

The first approach is based on novel application of Arachne, a state-of-the-art automated DNN (Deep Neural Networks) repair tool. Using Arachne, we were able to intentionally induce misbehavior in the model by identifying and manipulating key weights associated with particular inputs. We first began with random perturbations to the identified weights and then evolved the process to include our modified version of Arachne's differential evolution algorithm. Indeed, this search algorithm strategically finds an optimal patch for the model's weights to corrupt the model's output for specific inputs while retaining its overall performance on the remaining input data. However, despite a thorough hyperparameter optimization, Arachne's DE algorithm did not yield the expected results. In fact, Arachne's DE failed to find the patches to disrupt the outcome of the model for a certain set of inputs.

We then implemented an alternative strategy called Targeted Misbehavior Retraining (TMR). This approach entailed continuing training the network under test on selected input clusters

with deliberately corrupted labels, effectively generating mutants that misbehaved for the specified cluster without impacting the classification of other clusters or classes. This method proved to be an efficient way to create mutants with specific behaviors.

Finally, the performance and implications of these approaches were analyzed from a statistical viewpoint, where we show that our approach can consistently generate computationally cheap mutants affected by targeted meaningful corruptions.

Acknowledgements

I would like to express my heartfelt gratitude to my supervisor, Prof. Paolo Tonella, for providing me with the extraordinary opportunity to complete my master's thesis. My sincere thanks go to Dr. Nargiz Humbatova and Prof. Gunel Jahangirova for their unwavering support since the beginning of this work. Their professionalism and kindness have been instrumental in helping me navigate and surmount the challenges encountered during this study.

Special thanks to my girlfriend, Isabella, whose daily support and motivation were crucial in my completion of this enriching experience.

I am also deeply appreciative of my parents and my entire family for their continuous encouragement and support throughout these two years of study.

Lastly, I extend my gratitude to both my new and long-standing friends for their companionship during this journey.

Contents

Figures

Tables

Chapter 1

Introduction

In recent years, Deep Learning has become a fundamental component in the advancement of numerous innovative projects and products integral to our daily lives. As the demand for quality and safety in AI-based products escalates, the research community is increasingly focused on exploring diverse methods for assessing their quality. Among these is a technique called mutation testing. In traditional software, this approach deliberately introduces minor syntactic errors into a program to create various faulty versions also called mutants. The underlying principle of mutation testing is that the faults introduced are representative of typical programming mistakes, thus providing a means to evaluate the efficacy of a test suite in detecting such kinds of faults.

Traditional software systems typically have their decision logic embedded in the source code by developers. In contrast, the behavior of DL systems is largely determined by the training data set, training program, and various hyperparameters, which are the primary sources of defects in these systems [17]. Tools like DeepCrime [39] and DeepMutation++ [58] address these differences by proposing a set of mutation operators designed specifically for DL systems. These operators inject faults either before or after the training process, depending on the approach, which implies different requirements for computation resources and different levels of realism of the faults introduced.

The main idea behind this project was to create mutants that mimic a real situation in which models are trained on poor-quality data or when the model struggles to learn a specific type of input. For this reason, our aim was to simulate these scenarios by inducing already trained models (post-training) to misbehave for a particular input data that share certain features. By creating these post-training modified versions of the models, called mutants, we also aimed to achieve two main goals: facilitating smarter and faster post-training mutations specific to DL and reducing the computational cost of creating mutants with respect to pre-training mutations in DL systems. To group data that share common features, we conducted a comprehensive analysis and implementation of various combinations of feature reduction and clustering techniques to identify the most effective method for grouping input data. Then, to create the mutants, we employed two distinct approaches.

The first approach involved using a state-of-the-art tool called Arachne [51] to perturb the weights of the original model after it was trained and saved. This perturbation was tailored to specific sets of input data, with the aim of inducing the model to misbehave on a particular subset while retaining high performance on the remaining inputs. Arachne repairs DL models

that present misbehavior for a certain set of inputs by first localizing the weights of the model that are most impactful for the incorrect behavior, and then patching these weights through a genetic algorithm called Differential Evolution. We adapted Arachne to find the weights most influential in determining the model's outcomes for a cluster identified in the clustering phase. After having localized the weights, we perturbed them in order to create the mutants.

The second approach also utilizes a strategy to cluster the inputs. However, in this case, the model is retrained for a few epochs on a selected cluster with the intention of causing the model misbehavior on this cluster's inputs. The successful application of Arachne to the task of model repair did not produce the expected results for our mutation tool. However, the second strategy that we implemented in the frame of this thesis proved to be successful and offers advancement in the efficiency and effectiveness of mutation testing for DL systems. Our results show that the mutations we generate can successfully make the model misbehave on a particular subset (cluster) within a class of the dataset without excessively compromising the correct model's outcome for the other clusters and classes. We show that for both training and test sets, excluding those inputs within the considered cluster, the model retains a correct behavior. By evaluating the difference in evaluation metric between the original models and the mutants, we can state that our mutants are capable of discriminating between test sets of different qualities.

The contributions of this thesis are:

- an overview of the state-of-the-art approaches for mutation testing for DL systems
- a comprehensive exploration of feature reduction and clustering techniques in order to find relevant clusters within classes of the analyzed dataset
- a novel application of the Arachne tool for weight localization and manipulation of post-training models to create DNN mutants
- the introduction of new and fast post-training mutation for DL systems called Targeted Misbehavior Retraining

This thesis is organized as follows:

- **Chapter 1: Introduction.**
- **Chapter 2: Background.** Here, we explain the tools we adopted in our experimental approaches. Specifically, in this chapter, we briefly explain how Arachne, feature reduction, and clustering techniques work.
- **Chapter 3: Related Work.** We review the literature that is relevant to the topic of this project. We summarize four scientific papers about mutation testing for DL systems: DeepMutation [31], DeepMutation++ [58], MuNN [48], and DeepCrime [39].
- **Chapter 4: Methodology.** In this chapter, the methodology of this project is presented with a specific focus on the decisions we made to achieve our goals. The process of feature reduction and clustering is explained, along with how we used Arachne and DE for this project. Additionally, we explain our TMR method used for the creation of mutants.
- **Chapter 5: Results** In this chapter, we describe experimental setup, and discuss the achieved results.
- **Chapter 6: Conclusion** This chapter concludes the thesis and outlines future work.

Chapter 2

Background

In this chapter, we discuss the existing literature that served as a background for our approach and experiments.

2.1 Arachne

A key challenge in this work involved identifying a reliable method for locating the weights crucial for a Deep Neural Network's (DNN) prediction accuracy on specific inputs. The core aim was to construct mutants by modifying selected neural network weights. These weights are chosen to influence the network's behavior toward a distinct subset of inputs, specifically a certain cluster within a class of the dataset. This precise targeting of weights ensures that the mutants created are meaningful and tailored to DNN's processing of specific input categories. For this reason, in this section we will first introduce Arachne [51], a tool in which a novel technique called the Bidirectional Localisation method has been proposed. This procedure was adopted to discover the important weights of a neural network useful for our mutation approach.
Undoubtedly, DNNs have been adopted in many areas of research and production, such as medical imaging and autonomous driving. The development of Arachne was inspired by the need to quickly address misbehaviors of DNNs in critical environments. Jeongju Sohn *et al.* created **Arachne** [51], a search-based repair tool for Deep Learning Systems. The main idea behind this paper is to correct the misbehavior of already trained DNNs. Hence, the tool localizes the weights most influential for the wrong classifications. In particular, they introduced a new method called Bidirectional localization. Once the weights are localised, they employ a genetic algorithm to form a patch to correct the weights in order to get a desired model performance (see Fig. 2.1).

Another state-of-the-art tool focusing on model repair is Apricot [59]. Apricot iteratively improves deep learning models by adjusting their weights based on insights from reduced deep learning models (rDLMs). These rDLMs are trained on subsets of the original training data. Apricot adjusts the weights of the original model towards those of rDLMs that correctly classify test cases and away from those that misclassify them. The method is particularly effective for Convolutional Neural Networks (CNN) models and offers a unique approach to model repair without requiring additional inputs. However, it incorporates a partial re-training into its repairing process. This procedure could be costly since it involves a lot of computational power and does not guarantee the correction of mispredictions for specific inputs, as illustrated in Fig.

2.2.

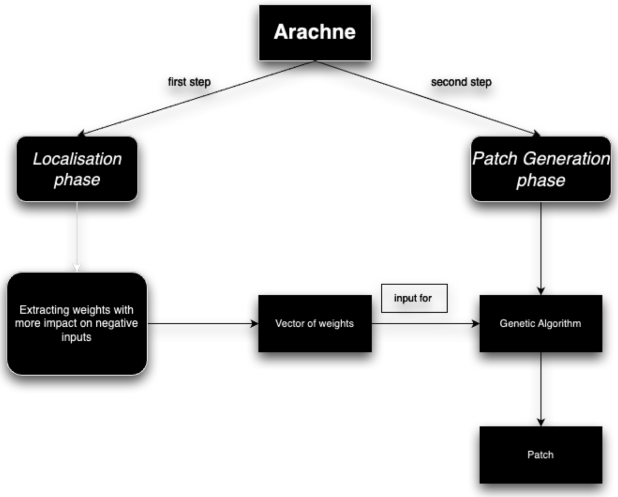


Figure 2.1. Arachne

Arachne, instead of retraining the model, tries to fix the weights that are influential on the misbehaving output. In particular, Arachne employs a two-step process: weight localization and patch generation. In the localization phase, it identifies neural weights likely connected to misclassifications, using both revealing misclassification (negative) and correctly processed (positive) inputs. The intuition is that modifying these weights in a certain way might impact the model behavior towards correction, while not disrupting the model's performance on already correctly classified inputs. In the patch generation phase, Arachne uses the fitness function, informed by the results of both positive and negative inputs, to search for a set of neural weights that correct the DNN's behavior, leveraging Differential Evolution to optimize the localized weights. As shown in Fig. 2.1, Arachne first localizes the weights with more impact on the negative inputs (misclassified samples) and then uses Differential Evolution to find a patch for the network, fixing the incorrect behavior of the model for the set of inputs that the model predicts erroneously. Furthermore, the authors show that Arachne can produce repairs that focus on the targeted misclassifications while minimally affecting other misclassifications with minimal impact on other behavior, and at speeds tens of times faster than Apricot.

2.1.1 Bidirection localisation method

The most critical part of Arachne's process is localizing the weights responsible for the DNN's misbehavior. For this reason, a novel method called Bidirectional Localization was proposed by the authors [51]. This method leverages two measurements: Forward Impact (FI) and Gradient Loss (GL). Forward Impact is a measure of the influence a neural weight has on the final classification outcome, whereas gradient loss quantifies the responsibility a neural weight has for the misclassification. Essentially, it measures how much a particular weight contributes to the error in the output of the neural network. A high gradient loss indicates that the considered weight plays a significant role in the misclassification. The key difference between Forward Impact and Gradient Loss lies in their focus: Gradient Loss focuses on the weight's contribution

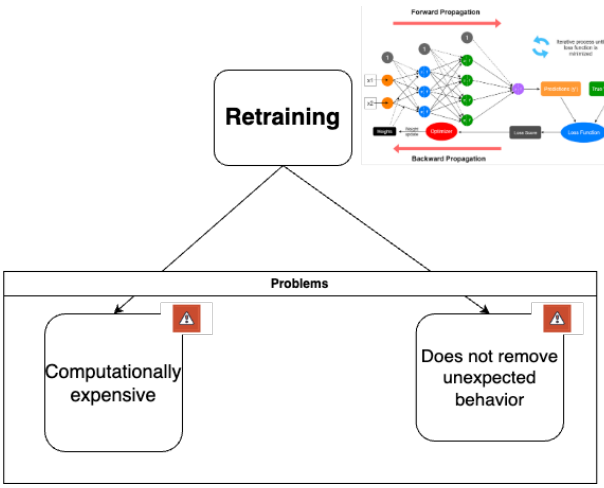


Figure 2.2. Existing retraining issues when using Apricot

to the error, whereas Forward Impact assesses the weight's overall influence on the network's final decision. These two components are treated as competitive objective functions. Gradient Loss is defined as:

$$GradientLoss = \frac{\partial \mathscr{L}}{\partial w} \tag{2.1}$$

Whilst Forward Impact is defined as:

$$ForwardImpact = \frac{o_i w_{i,j}}{\sum_l^{|L|} o_l w_{l,j}} \frac{\partial O}{\partial o'_j} \tag{2.2}$$

Given a neural weight $w_{i,j}$ connecting neurons o_i and o'_j, Forward Impact (Fig. 2.3) assesses the influence of $w_{i,j}$ on the final output [51]. To do this, it first evaluates the influence of this weight on the activation of the output neuron to which it is connected (o'_j) (first multiplier in Equation 2.2), which is calculated as the multiplication between $w_{i,j}$ and the average activation value of the neuron o_i, normalised by the aggregated influence of all the other neurons of the previous layer connected to o'_j. It then multiplies this quantity with the gradient of the final output O with respect to o'_j (second multiplier in Equation 2.2).

As input Arachne takes two sets of data called negative set (I_{neg}) and positive set (I_{pos}.) I_{neg} represents the set of samples that reveal the misbehavior of the DNN, whereas I_{pos} are those samples that are initially correctly classified by the network. However, not all the I_{pos} are taken into account since, in a fully trained DNN, the number of correctly classified samples is usually greater than the number of incorrectly classified samples. Indeed, the authors have sampled an equal number of I_{pos} and I_{neg}.

To circumvent involuntary disruption of the originally proper model behavior, Arachne computes both Forward Impact and Gradient Loss twice, once for I_{neg} and once for I_{pos}. Bidirectional localization (Alg. 1) returns a set of weights, which represents the Pareto front for the combined gradient loss and the forward impact of the positive and negative inputs. The Pareto Front represents a set of solutions that are considered optimal when there are trade-offs between two or more objective functions, such as FI and GL in this case.

Algorithm 1 Bidirectional Localisation method for DNN's weights, taken from Arachne's paper [51]

Input a DNN model (M), a set of weights within the DNN model (W), I_{neg}, I_{pos}, a loss function L

Output a set of weights to repair W_t

$pool \leftarrow []$
$I_{pos} \leftarrow RandomSample(I_{pos}, |I_{neg}|)$
for `w in W` **do**
    $grad\_loss_{neg} \leftarrow ComputeGradientLoss(w, M, I_{neg}, L)$
    $grad\_loss_{pos} \leftarrow ComputeGradientLoss(w, M, I_{pos}, L)$
    $grad\_loss \leftarrow \frac{grad\_loss_{neg}}{grad\_loss_{pos}+1}$
    $forward\_impact_{neg} \leftarrow ComputeGradientLoss(w, M, I_{neg}, L)$
    $forward\_impact_{pos} \leftarrow ComputeGradientLoss(w, M, I_{pos}, L)$
    $forward\_impact \leftarrow \frac{forward\_impact_{neg}}{forward\_impact_{pos}+1}$
    $Add\_tuple(w, grad\_loss, forward\_impact)\ to\ pool$
$W_t \leftarrow ParetoFront(pool)$
return W_t

2.1.2 Differential Evolution

Differential Evolution (DE) (Alg. 2) is a genetic algorithm (a type of metaheuristics) that iteratively improves a candidate solution with respect to a given quality measure [52]. Usually, metaheuristics search in a large space solution and do not ensure finding the optimal solution [13; 32]. They are particularly effective in scenarios where an analytical solution is not feasible, such as when the gradient of the fitness function cannot be computed. This attribute makes them versatile for difficult optimization problems where traditional analytical approaches are limited or impractical.

Arachne's DE algorithm takes as input the target weights W_t, obtained by the Localisation method explained in the previous Section. It then generates initial values for each of the localized weights by sampling from a Gaussian distribution with mean and standard deviation calculated using all the other weights in the same layer. To retain the initial correct behavior and to fix the misclassification of the DNN model, Arachne defines the following fitness function for its DE algorithm:

$$score(i,X) = \begin{cases} 1 & \text{if } label_x(i) = label_{GT}(i) \\ \frac{1}{Loss(i,X)+1} & otherwise \end{cases} \tag{2.3}$$

$$fitness = \sum_{i_p \in I_{pos}} score(i_p,X) + \alpha \sum_{i_n \in I_{neg}} score(i_n,X) \tag{2.4}$$

where i is an input, $label_x(i)$ is the prediction of the model patched with the set of weights X, $label_{GT}(i)$ is the ground truth, and α is a parameter to balance the two components. Of course, increasing α means more priority on correcting the misbehavior, while a smaller α means more focus on conserving the correct behavior. Unlike the localization phase, Arachne DE considers I_{pos} as the entire set of input samples from the dataset initially correctly classified.

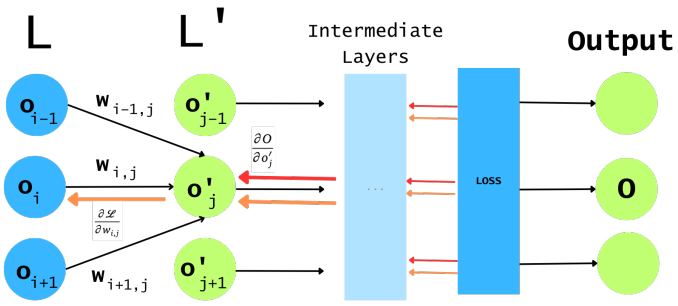


Figure 2.3. Arachne fault localization diagram. The black arrows represent the forward process, whereas the red and orange arrows denote the backpropagation. Black and red arrows show the FI, while the orange arrows show the GL. Both quantities FI and GL estimate the probability of $w_{i,j}$ being localized.

Algorithm 2 Differential Evolution, taken from [51]

Input cross-over rate, **CR**, size of the population **NP**, number of generation **Gen**, fitness function **fitness**, mutation rate **F**

Output best solution so far *best*

```
1:  pop ← Initialize(NP)
2:  for gen in [1, Gen] do
3:      for each X in pop do
4:          X_1, X_2, X_3 = SelectRandom(pop)
5:          j ← randomly select between [1, N]
6:          for i in [1, N] do
7:              r_i ← SelectUniform(0,1)
8:              if i = j or r_i ≤ CR then
9:                  v_i ← x_{1,i} + F(x_{2,i} + x_{3,i})
10:         if fitness(V) ≥ fitness(X) then
11:             X ← V
12: best ← selectBest(pop)
13: return best
```

2.2 Feature Reduction

Feature reduction is a commonly used term to describe techniques for reducing the input space to a smaller domain. The primary objective of these methods is to decrease the number of features so that other machine-learning techniques can work more efficiently on the most significant components extracted by these methods [23]. Furthermore, reducing the number of features enhances the scalability of the entire machine-learning pipeline and mitigates power consumption. In addition, storing only some features among all is a good practice for consuming less memory on devices. Feature reduction could also address the problem of the so-called "curse of dimensionality" [26]. This occurs when the situation involves many dimensions, leading to a sparse data representation.

We employed feature reduction techniques to extract meaningful features and to reduce noise within the data. This can yield more insightful and high-level attributes of inputs than the original input vectors. Consequently, dimensionality reductions, such as Principal Componentes Analysis (PCA) [24], Variational Autoencoder (VAE) [27], and Uniform Manifold Approximation and Projection (UMAP) [36], were introduced in the pipeline of this project. In particular, we decided to use PCA because it is a popular technique for reducing the dimensionality of the data while preserving most of the variability in the data. In addition, PCA is very useful because it can help discard the noise in the data. The use of a Variational Autoencoder in our project was an excellent strategy for tackling complex, high-dimensional data sets. VAEs excel at capturing nonlinear relationships, which is a major advantage over traditional linear techniques. They don't just compress data but learn the underlying patterns of the data. Their encoding-decoding mechanism exposes meaningful features, providing deeper insight into the data structure. Using UMAP allowed us to preserve local details in high-dimensional data. UMAP handles non-linear data well, revealing hidden structures that linear methods such as PCA might miss. This is key for complex datasets where nuances matter. It also strikes a nice balance between showing local and overall data patterns, making it easier to identify and differentiate clusters.

2.2.1 PCA

PCA [47] [24] is a linear transformation that compresses the data points and projects them on a new coordinate system. This process starts by centering each feature, shifting every point so that the mean of each feature is zero and aligned with the origin. The next step involves computing the covariance matrix, which captures the relationships between the features of the dataset. The covariance matrix is symmetric and squared, and it has all the variance of each feature in its diagonal. The next step is computing the eigenvalues and eigenvectors from the covariance matrix to obtain the principal components that are linear combinations of the original features. These components are designed to capture a significant portion of the information in the dataset. Once the principal components are identified, PCA projects the data onto these components, effectively transforming the data into a new space defined by these axes representing the directions of greatest variance in the data. This transformation results in a new representation of the data with reduced dimensions, where the axes correspond to the principal components and capture the most significant patterns of variation in the original dataset.

In Fig. 2.4, there is an example of a PCA application. In the left image, some points are represented in a three-dimensional space. Within the same image, the principal components are illustrated in red. They represent the directions in which the data varies most significantly.

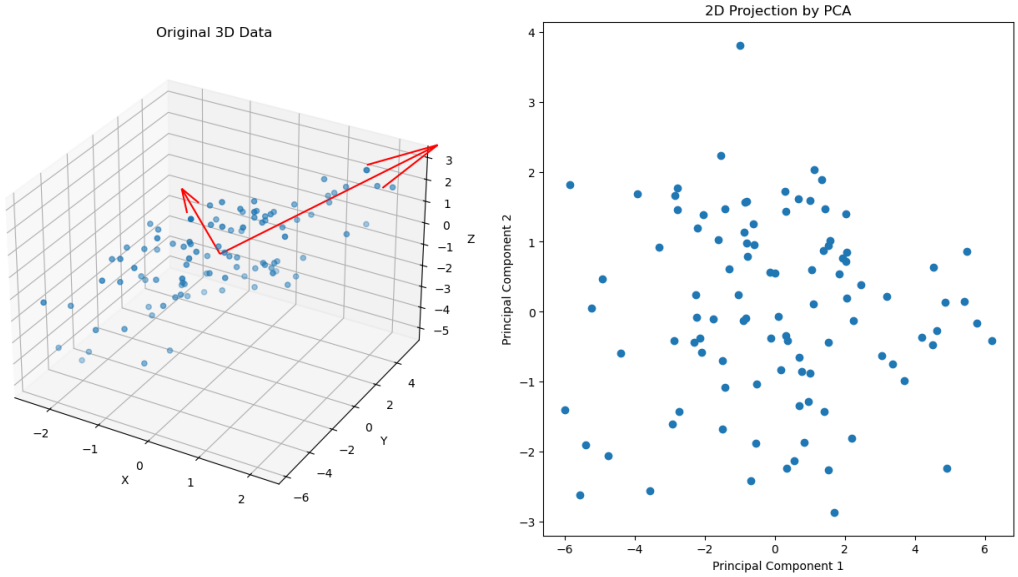


Figure 2.4. Visualizing PCA Dimension Reduction. On the left, the two principal components are illustrated within the original input space (red arrows). On the right, we see a 2D representation, where these components form a new, reduced space.

In the right image, after applying the PCA, the points are represented in two dimensions, preserving most of the variance.

2.2.2 Variational Autoencoder

Variational Autoencoder was presented in work by Kingma *et al.* [28], and it belongs to a more extensive class of autoencoders (AEs). AEs are neural networks that compress the data through an encoder into a low dimensional space (called latent space), and then try to reconstruct the input with a decoder. The latter works to generate or reconstruct an output that is as close as possible to the original input data. In contrast with AEs, which try to encode the input in a fixed vector, VAEs encode the distribution of the input into the latent space. This approach is advantageous for feature reduction as it allows VAEs to capture a finer representation of the data. By modeling the distribution, VAEs account for the variability and uncertainty inherent in real-world data, leading to a richer understanding of the dataset compared to classic AEs. In Fig. 2.5, the general structure of a VAE is shown. The encoder processes the input image x and maps it to a latent space representation, characterized by a mean $\mu_{z|x}$ and a standard deviation $\Sigma_{z|x}$. This latent space is then used to generate a set of latent variables z. The decoder takes these latent variables and reconstructs the input, approximating the original input image in the output $\hat{x}$.

2.2.3 UMAP

As mentioned previously, we employed UMAP [36] for dimensionality reduction. We decided to use UMAP instead of t-SNE (t-distributed Stochastic Neighbor Embedding) because, as experi-

mented by UMAP developers, UMAP is significantly more performant than t-SNE when embedding into dimensions larger than 2. The authors compared UMAP and t-SNE using the MNIST dataset, particularly focusing on the time required for each method. The results revealed that UMAP significantly outperformed t-SNE in terms of speed, requiring considerably less time to complete the task. Given these findings, and the fact that our analysis also involved the MNIST dataset, it was a logical decision for us to choose UMAP over t-SNE for our project.

UMAP is recognized for its ability to maintain both local and global structures of high-dimensional data, which was essential for our goal of reducing the dataset size while retaining its significant characteristics.

UMAP works by first computing pairwise distances between high-dimensional data points using a relevant metric, like cosine or Euclidean distance. After that, a fuzzy simplicial set (a mathematical structure used to model the concept of how close or similar the data points are to each other in their local neighborhood) is created to represent the "nearness" and local neighborhood structure of the data points. UMAP starts with a random embedding and finds a low-dimensional representation that maintains these relationships. This embedding creates a graph that represents the global data structure. Next, data point placements are adjusted based on connections, the edges between the nodes within this graph, through a refinement phase. The process iterates until convergence, determined by a set number of iterations or a change threshold. The final low-dimensional embedding, effectively maintaining neighborhood and global structures, can afterward be utilized for a variety of further tasks such as clustering.

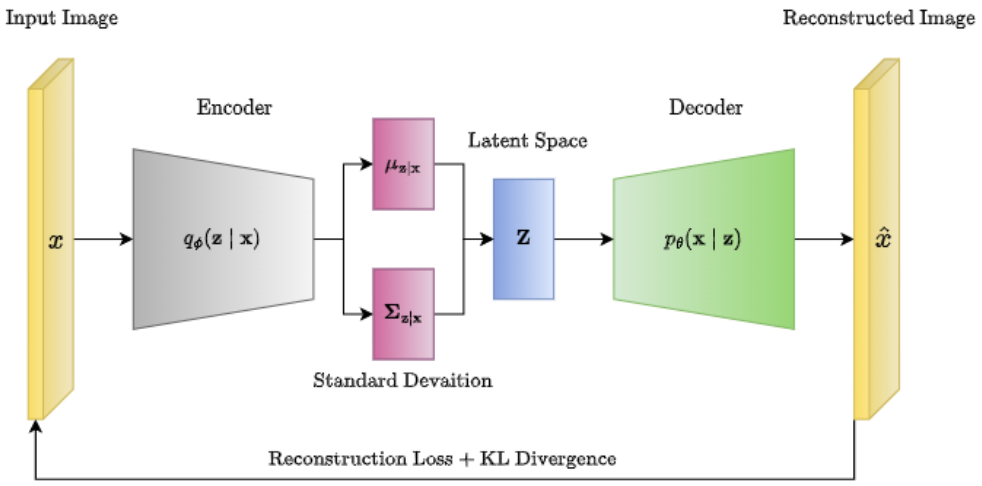


Figure 2.5. VAE structure, image taken from [37]

2.3 Clustering

One of the most challenging and important phases of our work was grouping the samples within a selected class of the dataset according to shared features. After reducing the dimensionality of the data through VAE, PCA, or UMAP the next step was performing clustering on it. Clustering

techniques are a subset of machine learning methods used to group similar data points together without prior labeling or categorization. These techniques are especially useful for exploring and understanding unlabeled data. In essence, clustering algorithms identify patterns and similarities within the data and organize these data points into clusters. Since our samples belong to a class within the dataset but are neither labeled nor part of any subclasses, this strategy was ideal for them.

In our project, we chose K-means, Agglomerative Clustering, Gaussian Mixture Model (GMM), and HDBSCAN for clustering due to their complementary strengths. Indeed, our chosen clustering algorithms represent a diverse collection of approaches: Kmeans is a partitioning method, Agglomerative belongs to Hierarchical methods, GMM is a Distribution-Based method, and HDBSCAN is categorized under density-based methods. This selection strategy was deliberate to harness the unique strengths inherent in each of these different clustering methodologies. K-means is a popular method known for its simplicity and efficiency [2], particularly effective in identifying globular clusters. Globular clusters are clusters that are spherical or round in shape, usually distributed around a central point. The density of the points within a cluster is almost the same in each subregion of the cluster. Agglomerative Clustering, a hierarchical method, is beneficial because it produces a tree-like structure that is straightforward to interpret. GMM is better at handling outliers and analyzing complex and mixed data than KMeans [25]. Lastly, HDBSCAN (Hierarchical Density-Based Spatial Clustering of Applications with Noise) handles clusters of varying densities and is particularly effective in identifying outliers, making it useful for datasets with complex structures and noise. Together, these methods provide a comprehensive toolkit for discovering the underlying patterns in our data, catering to various cluster shapes, sizes, and densities.

2.3.1 KMeans

One of the most popular clustering algorithms is the KMeans [33]. Its final goal is to divide a dataset into discrete groups or clusters according to similarities. It works by first choosing a number of cluster centers at random. Next, each data point is allocated to the closest cluster center using a distance metric, usually the Euclidean distance. Following this initial assignment, each cluster's center is recalculated by the algorithm using the mean of all the points assigned to it. The data points are then reassigned to the closest new cluster center. This iterative process repeats until cluster centers stabilize, ensuring distinct clusters. The main advantages of this algorithm are being fast and guaranteeing convergence. However, it assumes that the clusters have globular shapes, which might not be the case for real-world data. Other limitations are the required parameter K, which is the number of clusters, and the fact that outliers have the ability to shift the position of the centroids. Moreover, KMeans can struggle with clustering data where clusters are of varying sizes and densities [14].

2.3.2 Agglomerative Clustering

Another family of approaches for clustering the data is hierarchical clustering. These clustering algorithms create hierarchical clusters by merging or splitting sequentially. A common clustering algorithm within this category is the agglomerative clustering [22], which is used to arrange data points into clusters according to how related they are. The steps that describe the algorithm are:

1. Start by treating each data point as a single cluster.

2. Find the closest (most similar) pairs of clusters and merge them together to form a new enlarged cluster.

3. Repeat step 2 until the desired number of clusters is achieved.

The outcome is a visual representation of objects in the form of a tree, which is known as a dendrogram. The algorithm takes the linkage rule as a hyperparameter, which is a criterion determining how the distance between sets of observations is computed. The four most common types of linkage rules are single linkage, which merges clusters based on the shortest distance between points in two clusters; complete linkage, which uses the maximum distance between points in two clusters to merge them; average linkage, which merges clusters based on the average distance between all pairs of points in the clusters; Ward's method which merges clusters that result in the minimum increase in total within-cluster variance after the merger.

The advantage of agglomeration clustering is that visual inspection can often be useful in understanding the data's structure. Nevertheless, it is computationally expensive since it requires calculating and recalculating full pairwise distance matrices [54]. Indeed, the closest pair of clusters are merged at each stage of the clustering process, necessitating a recalculation of the distances. It becomes less practical for very large datasets as the number of calculations grows quickly with dataset size.

2.3.3 Gaussian Mixture Model

GMM is another method belonging to clustering algorithms, specifically to the distribution-based clustering family. This probabilistic model assumes that all data points come from a mixture of Gaussian distributions with unknown parameters [12]. GMM works by using the expectation–maximization (EM) algorithm for estimating these parameters. This iterative approach alternates between allocating data points to the most likely Gaussian component (expectation step), which is one of the individual Gaussian distributions, and then adjusting the parameters of the Gaussians (maximization step) to better suit the data. The main advantage of using GMM is that it can work with data in any shape or form [56]. Nevertheless, the number of componentes must be known in advance, before executing the algorithm. The number of components are the number of Gaussian distributions used to model the data.

2.3.4 HDBSCAN

HDBSCAN, presented by Campello *et al.* [7], belongs to the density-based clustering family. Its final goal is finding high-density regions and expanding clusters from them. Unlike KMeans, HDBSCAN does not rely on the assumption that the clusters have a globular shape. Furthermore, it does not require many parameters from users. Users only need to specify the minimum number of samples in a cluster.

The first step in HDBSCAN involves transforming the dataset for hierarchical clustering. This involves building a mutual reachability distance graph, often using a minimal spanning tree (which is a way to connect a group of points with the shortest possible total distance of lines, without forming any loops). This graph is important because it represents the connectivity of points in the dataset, considering the local density. A minimal spanning tree is used here to efficiently connect all points with the minimum possible total distance, reflecting the structure of the data in terms of density. HDBSCAN proceeds by gradually merging points in this graph based on their distance, forming a hierarchy. The resulting single linkage tree in Fig. 2.6 is

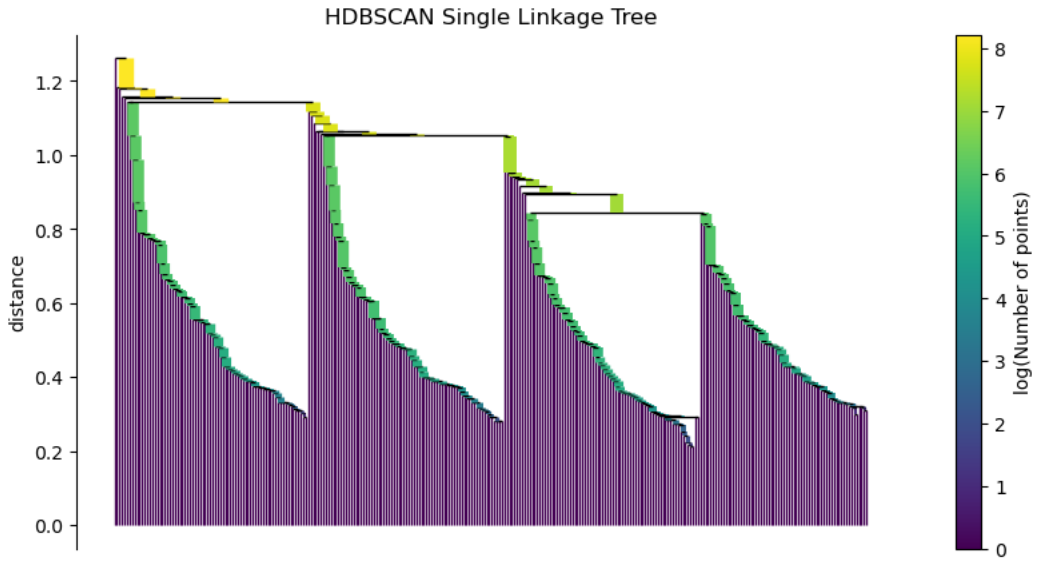


Figure 2.6. Single Linkage Tree produced by HDBSCAN

a visual representation of this process, with each branch indicating a potential cluster. The horizontal axis represents the data points, while the left vertical axis represents the distance at which clusters merge. Each point is considered a single-point cluster at the bottom of the tree. The branches that merge (horizontal line) indicate a potential cluster at the distance level where the horizontal line is placed. The color represents the logarithm of the number of points in the cluster at different levels of the tree. In the figure 2.6, darker colors mean denser regions. Shorter bars indicate clusters that are closer in distance and more similar, whereas higher bars mean that the clusters need a larger distance to be merged. In fact, clusters that merge in the lower part of the tree are denser.

However, this tree can become complex, particularly for large datasets. To address this, HDBSCAN simplifies the tree, pruning it to retain only significant branches, which describe the essential structure of the data in a more manageable condensed tree (Fig. 2.7). The condensed tree works by condensing the vast and intricate cluster hierarchy into a smaller tree with a little bit more information linked to each node, which is known as cluster extraction. Rather than viewing a cluster split as two separate clusters splitting off, HDBSCAN considers it as one persistent cluster that is "losing points." From the hierarchy above 2.6, a cluster split is frequently one or two points breaking off from a cluster. The algorithm proceeds along the hierarchy and examines at each split whether any newly formed clusters have fewer points than the minimum cluster size, which is the hyperparameter required by HDBSCAN. If it is the case that we have fewer points than the minimum cluster size, we declare it to be 'points falling out of a cluster,' and we say at what distance value that occurred. In contrast, we define a split into two clusters that are at least as big as the minimum cluster size as a legitimate cluster split, and we allow that split to remain in the tree. We have a much smaller tree with few nodes after going over the entire hierarchy and doing this. Each node contains information about how the size of the cluster at that node reduces across different distances. This can be shown as a dendrogram, Fig. 2.7, that resembles the one before, Fig. 2.6, with the width of the line once

more denoting the number of points in the cluster. This time, though, as points separate from the cluster, the width changes over the entire length of the line. This stage ensures that the identified clusters are robust and meaningful within the dataset's structure. Within the figure 2.7, the most stable clusters are circled. Points not included in any stable cluster are classified as noise. This classification implies that these points do not belong to any dense region and are likely outliers or anomalies in the data. In real-case scenarios, real data often can contain noise, making HDBSCAN perfect for managing these situations.

e.

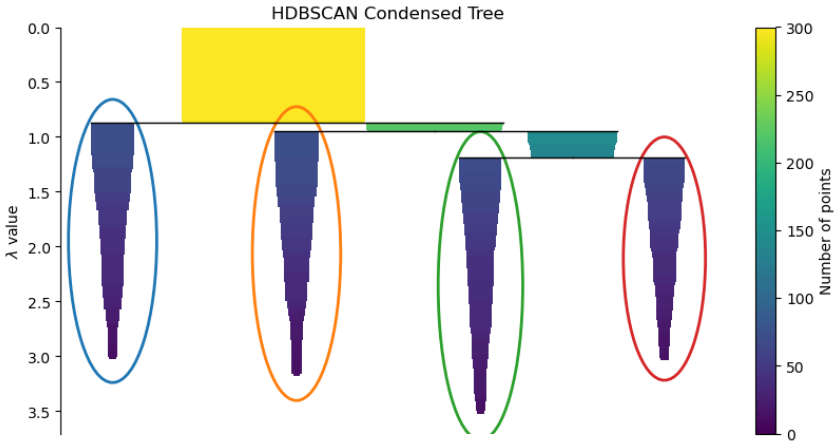


Figure 2.7. Condensed Tree produced by HDBSCAN

Chapter 3

Related Work

3.1 Mutation Testing

Mutation testing is an approach that intentionally introduces faults into the program being tested, such as minor syntactic alterations, to produce a set of defective programs known as mutants. This approach is based on the basic idea that faults utilized in mutation testing are the same errors that programmers typically make. Mutation testing aims to evaluate a test suite's quality based on how well it can identify errors. Each of the produced mutants is subjected to the test suite in order to achieve this. A mutation is considered killed if the result of executing the original program differs from the result for a particular mutant. The mutation score is the ratio of killed mutants to the total number of created mutants. The test suite's quality increases with a higher mutation score.

3.2 Mutation Testing for Deep Learning

The standard method for assessing DL models involves analyzing their performance using a test dataset. The reliability and quality of this test dataset are necessary for establishing confidence in the trained models. If the test dataset is inadequate, even DL models that demonstrate high test accuracy might still be deficient in terms of generality. In this chapter, we present some state-of-the-art tools in mutation testing for DNN systems that address this problem. We will start by discussing a paper named DeepMutation, which aimed to develop a mutation testing tool for deep learning systems that measures test data quality. The second paper, DeepMutation++, is an extension of the first one. It supports and analyzes test data quality for deep learning systems and includes RNNs (Recurrent Neural Networks [16]) in its scope. The third paper, MuNN, examines mutation's impact on neural networks and how neural depth affects mutation analysis. Finally, we will discuss DeepCrime, the first source-level pre-training mutation tool based on real DL faults.

3.3 DeepMutation

The paper [31] proposes a framework to assess the effectiveness of the test set. The authors drew inspiration from conventional software mutation testing techniques. They proposed

source-level and model-level mutation operators that introduce faults into the training process and the pre-trained model. Source-level mutations refer to those faults injected in the source code of the training and those injected into the training data. In contrast, model-level mutations refer to those faults injected into an already-trained DL model. Unlike traditional programs, defects in DNNs are mainly caused by the training set and the training program (as illustrated in Fig. 3.1). The authors intentionally injected faults into the training data and the model to obtain model mutants to experiment against the test set. The authors created these mutants by focusing on the DNN components (such as layers and activation functions) that mostly affect the network's behavior instead of only applying small syntactic changes, for example, into the training source code. The concept behind mutation testing is similar to that of traditional software. The more differences in behavior that can be identified by T between the original DL model M and its mutant versions $\{M'_1, M'_2, \ldots, M'_n\}$, the higher the quality of T is likely to be.

The authors proposed eight source-level mutation operators (listed in Table 3.1) that can be used to simulate potential faults during the data collection or the coding of the training. Developers typically follow a step-by-step process to train machine learning models. The first step involves preparing a program, called P, for training the model and gathering the necessary training data, D. Once the training is complete, a model called M is obtained by running P with D. Before mutation testing, the initial test suite T is run against the program to extract only the passing tests T' (subset of T), which are then used for further mutation analysis. T' consists of the test data points in T that are correctly processed by the original DL model M. Then, to create a mutant, both P and D are slightly modified by applying mutation operators. The modified program and data are P' and D,' respectively. These two modified versions are then used to create the mutant M'. The latter is used to evaluate the quality of test set T.

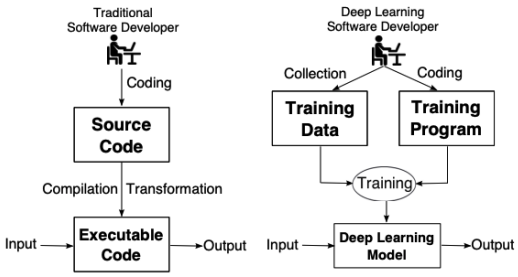


Figure 3.1. Traditional and DL software development. Image taken from DeepMutation [31].

In traditional mutation testing, the evaluation metric is usually the ratio of killed mutants to all mutants. However, according to the authors, this primary metric may not be appropriate for Deep Learning. This is because the amount of samples of T' is usually very high and can easily kill a mutant. They assert that a test sample t' kills class c_i of mutant m if the following conditions are met:

- t' is correctly classified as c_i by the original model
- t' is not correctly classified as c_i by the mutant

where $c_i \in C$, and C represents a set of classes. Next, they introduced a new mutation score for

Fault Type	Level	Target	Operation Description
Data Repetition (DR)	Global	Data	Duplicates training data
	Local	Data	Duplicates specific type of data
Label Error (LE)	Global	Data	Falsify results (e.g., labels) of data
	Local	Data	Falsify specific results of data
Data Missing (DM)	Global	Data	Remove selected data
	Local	Data	Remove specific types of data
Data Shuffle (DF)	Global	Data	Shuffle selected training data
	Local	Data	Shuffle specific types of data
Noise Perturb. (NP)	Global	Data	Add noise to training data
	Local	Data	Add noise to specific type of data
Layer Removal (LR)	Global	Prog.	Remove a layer
Layer Addition (LA_s)	Global	Prog.	Add a layer
Act. Fun. Remov. (AFR_s)	Global	Prog.	Remove activation functions

Table 3.1. DeepMutation Source-level mutation operators

DL classification problems defined as:

$$MutationScore(T', M') = \frac{\sum_{m' \in M'} |KilledClasses(T', m')|}{|M'| \cdot |C|} \tag{3.1}$$

where KilledClasses(T', m') is the set of classes killed by test data T', M' is the set of mutated models, and C is a set of classes.

Predicting the behavioral difference caused by mutation operators is challenging due to the complex nature of the model. Thus, to avoid too many differences between the original model and the mutant, mutants with a high error rate on T' are discarded by the authors of this paper. To evaluate the collective impact of mutation operators, the researchers calculated the Average Error Rate (AER) of the mutants defined as:

$$AveErrorRate(T', M') = \frac{\sum_{m' \in M'} ErrorRate(T', m')}{|M'|} \tag{3.2}$$

As part of their work, the authors also propose model-level mutation operators (listed in Table 3.2) to apply on an already trained model, operating from weight to layer level. As these mutation operators do not require a full retraining of the model, they make the mutation process faster and more feasible.

In their analysis, the authors utilized two of the most popular datasets, MNIST and CIFAR10, which contain ten distinct classes. Additionally, they employed three high-performing models, with the first two trained on the MNIST dataset and the third on the CIFAR-10 dataset. The first experiments were conducted on two data settings: the first sampled 5000 data points from the training set, and the second sampled 1000 data points from the test set. Each setting had a paired dataset, denoted as (T_1, T_2), where T_1 was uniformly sampled from all the classes, while T_2 was not. The authors repeated the experiment five times to avoid bias. They then filtered out failed cases using the original models. Failed cases are those cases that produced an incorrect behavior of the original models.

They created mutants by applying source and model-level mutation operators and then calculated the average error rate and mutation score by running candidate test data against

Mutation Operator	Level	Description
Gaussian Fuzzing (GF)	Weight	Fuzz weight by Gaussian Distribution
Weight Shuffling (WS)	Neuron	Shuffle selected weights
Neuron Effect Block. (NEB)	Neuron	Block a neuron effect on following layers
Neuron Activation Inverse (NAI)	Neuron	Invert the activation status of a neuron
Neuron Switch (NS)	Neuron	Switch two neurons of the same layer
Layer Deactivation (LD)	Layer	Deactivate the effects of a layer
Layer Addition (LAm)	Layer	Add a layer in neuron network
Act. Fun. Remov. (AFRm)	Layer	Remove activation functions

Table 3.2. Model-level mutation operators

them. As mentioned before, they discarded mutants showing very different behavior w.r.t the original model. In all experiments, the uniformly sampled data group had a higher average error rate on the mutant models, suggesting it had more defect detection ability. In both Fig. 3.2 and Fig. 3.3, the low mutation score suggests poor test data quality, possibly due to high dimensionality space. The authors underline that the uniformly sampled data group achieves a higher average error rate on the mutant models, which indicates the uniformly sampled data has higher defect detection ability (better quality from a testing perspective). Furthermore, they noticed that for model C, test data had better quality than training data because it got a higher mutation score even though the training data included more samples than the test data. (training data is composed of 5000 samples while test data is only 1000.) Since, as mentioned, the uniformly sampled test data group had a higher mutation score than the poor-quality test data (represented by the non-uniformly sampled data group), the authors concluded that their mutation operators could differentiate between low and high-quality test datasets. Indeed, while a test dataset commonly has a similar probability distribution to the training dataset, it is frequently independent of it. A good test data set should cover a wide range of functional aspects of the use case for DL software in order to evaluate performance (i.e., generalization) and identify areas where a fully trained DL model is weak. In addition, the experiments were performed on individual classes/labels. The findings revealed that certain classes obtained significantly low mutation scores and AER. This implies that the test data for these classes could still be further improved. According to the researchers, their tool allows for a numerical evaluation of test data quality for each category.

In conclusion, this tool can identify weaknesses in test data and suggest enhancements to cover more defect-sensitive cases.

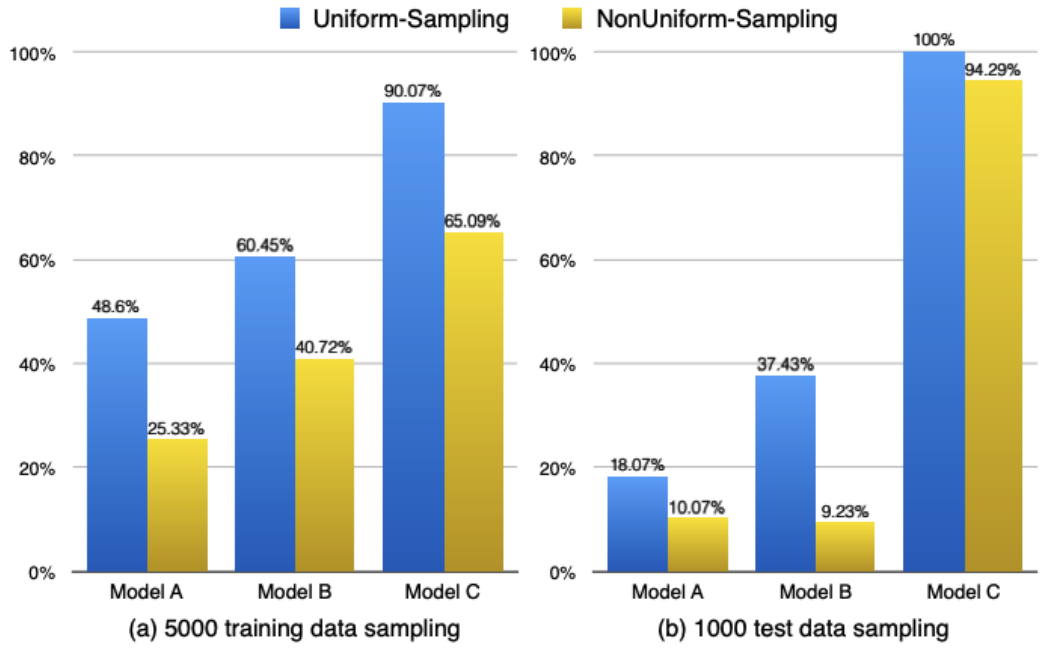


Figure 3.2. The averaged mutation score of source-level mutation testing. Image taken from DeepMutation [31].

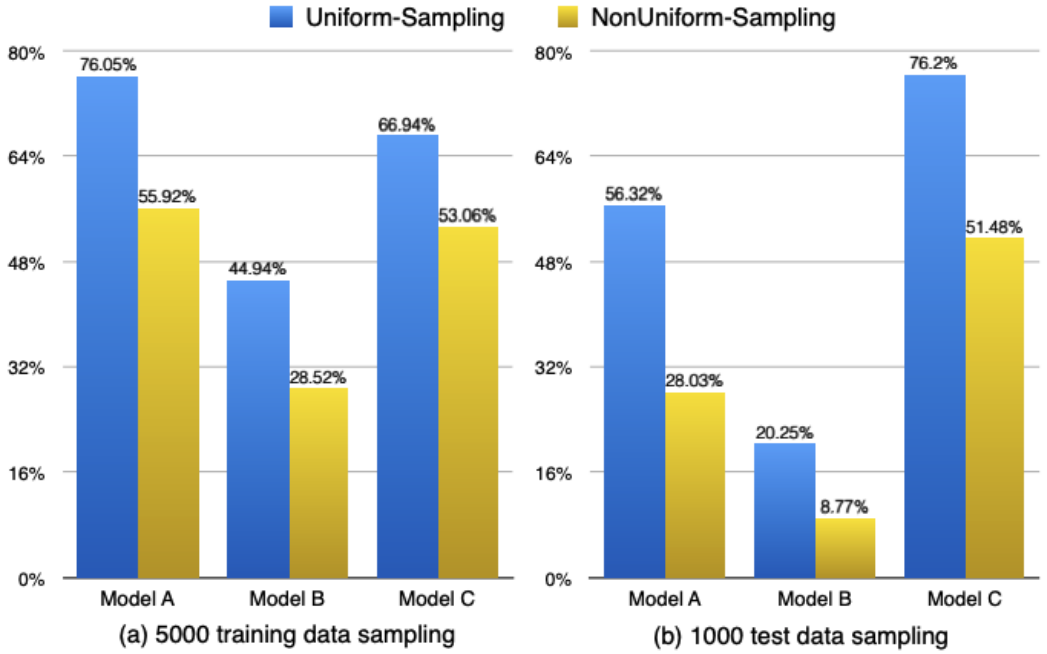


Figure 3.3. The averaged mutation score of model-level mutation testing. Image taken from DeepMutation [31].

3.4 DeepMutation++

Lei Ma *et al.*[58] developed a new tool called DeepMutation++ for DNNs, building on their previous work on DeepMutation. This mutation testing-based tool can analyze feed-forward neural networks (FNNs) and Recurrent Neural Networks (RNNs). DeepMutation++ includes eight model-level operators from DeepMutation, as well as new operators for RNNs. Particularly interesting are the dynamic operators. A dynamic operator mutates the internal runtime status (for example, memory state or gate control state) of an RNN on the fly segment by segment (creating a dynamic mutant). A new dynamic mutant generation mechanism is particularly important, as it can identify vulnerable or weak segments. Segments are chunks of input data processed in an iteration of an RNN. A vulnerable segment refers to a portion of the test input that, when modified or 'mutated' at runtime, can potentially lead to incorrect outputs from an RNN. Instead, the static-level operators are employed for creating mutant models, not in real time. Once these mutants are created, they are saved to a designated location for subsequent mutation analysis. Dynamic operators are introduced to conduct more granular, segment-level analysis. These operators manipulate the runtime state of an RNN, including memory and control gate states, altering the model's behavior in real-time, segment by segment. It's important to note that dynamic-level operators do not save mutants on storage like static ones.

For experiments, they tested their tool on the MNIST dataset, using an FNN, and IMDB dataset for RNN robustness. To use the tool, a user must provide a model and a set of test data for analysis. The tool uses mutation operators to create several mutants, and then compares the behavior of the original model and the mutants against the provided test data. This helps to identify any differences in behavior and improve overall testing. Furthermore, DeepMutation++ can identify on the fly which input segment is weak. For RNNs, the authors of this paper used the mutation operators reported below in Table 3.3.

Level	**Operator**	**Description**
Static	Weight Gaussian Fuzzing (WGF)	fuzz weights
	Weight Precision Reduction (WPR)	reduce weights' precision
State	State Clear (SC)	clear the state to 0
	State Reset (SR)	reset state to previous state
	State Gaussian Fuzzing (SGF)	fuzz state value
	State Precision Reduction (SPR)	reduce state value's precision
Dynamic	Gate Clear (SC)	clear the gate value to 0
	Gate Gaussian fuzzing (GGF)	fuzz gate value
	Gate Precision Reduction (GPR)	reduce gate value's precision

Table 3.3. DeepMutation++ RNN mutation operators

Note that static-level operators are used for generating mutants, which are not dynamic and are stored in a user-specified location, allowing the user to investigate the quality of the test data as a whole. In contrast, dynamic operators can modify the calculation way of the units in the original model to obtain the corresponding new prediction result.

Regarding the metrics for FNNs, DeepMutation++ uses $KScore_1$ for the evaluation of the whole data defined as follows:

$$KS_1(t,m,M) = \frac{|m'|m' \in M \wedge m(t) \neq m'(t)|}{|M|} \tag{3.3}$$

where t is an input, m is a DNN model, and m' is its mutant. Instead, only for RNNs (since the input is segmented), the authors presented a metric for segment-level killing score called $KScore_2$ defined as following:

$$KS_2(t_i, m, M) = \frac{\sum_{m' \in M} ||prob(t_i, m) - prob(t_i, m')||_p}{|M|} \tag{3.4}$$

where t_i is the $i-th$ segment of an input t; M is a set of RNN mutants, and m is an RNN model. This time, m' is the *dynamic* mutant by mutating m with dynamic-level operators. This score describes the prediction probability divergence on the output. Consequently, the larger is $KScore_2$, the less robust is the model against t_i. Several experiments were done, but the most significant are those related to RNNs, showing how $KScore_2$ is correlated with the attacking difficulty (adversarial score). In conclusion, DeepMutation++ can provide a tool for mutation analysis of DNNs and a means for identifying vulnerable inputs and the corresponding segments for RNNs.

3.5 MuNN

In their paper, Weijun Shen et al. [48] introduced a new mutation analysis technique for Deep Neural Networks, which they call MuNN. The authors developed five mutation operators based on the structure of neural networks. They conducted their research on the MNIST dataset to answer two key research questions: (1) "How does mutation affect neural networks?" and (2) "How does neural depth affect mutation analysis?". Typically, mutation operators include *insert*, *delete*, and *change*. The authors focused on the last two methods and presented five mutation operators, as illustrated in Fig.3.4.

Mutation Operator	**Mutation Function**	**Description**
DIN	$\text{op}_{\text{din}}(n)$	Delete Input Neuron
DHN	$\text{op}_{\text{dhn}}(\text{layer}, \text{index})$	Delete Hidden Neuron
CAF	$\text{op}_{\text{af}}(\text{type})$	Change Activation Function
CBV	$\text{op}_{\text{b}}(\text{extend}, \text{ratio})$	Change Bias Value
CWV	$\text{op}_{\text{w}}(\text{extend}, \text{ratio})$	Change Weight Value

Table 3.4. Mutation operators of MuNN

Two models were used for the experiments. The first model ($model_a$) was a fully connected neural network with two hidden layers consisting of 128 and 64 neurons, respectively. The second model ($model_b$) was a CNN with three convolutional layers, each containing 32, 64, and 128 filters with a convolutional kernel size of (3,3). In order to answer the first research question, the first step was to delete 784 input neurons one by one. The results were interesting as they showed that despite this significant reduction in the number of neurons, there was only a slight drop in the performance of the model. The study also highlighted which input neurons were important for the correct prediction of each label and which labels shared the most and least common neurons. This was determined by the number of important neurons that overlap between the labels. For instance, label 0 and label 9 have the maximum shared input neurons influencing the original model. This is due to the fact that 0 and 9 have the highest number of common parts. The authors of the research paper emphasized the importance of having

a strong domain understanding for mutation testing. They suggested that this understanding could help researchers better comprehend the inner workings of neural networks. In the case of classifying labels 0 and 9, we can mutate the model by deleting input neurons critical for this task in order to evaluate the completeness of the test samples. During their experiment, they applied a strategy to the hidden layers where they randomly deleted *n* neurons. Their findings showed that when five or fewer neurons were deleted, the model's performance remained almost unchanged. However, when they deleted 20 neurons, the model's performance was significantly reduced. Furthermore, increasing the proportion of selected parameters results in greater model disturbance due to the increased magnitude of modification. To analyze the effect on the original model, "extent" and "ratio" are calculated. The former measures the magnitude of parameter change, while the latter represents the ratio between the modified and original parameters.
To address the second research question, the researchers trained four neural networks with different numbers of hidden layers: 1, 3, 5, and 10. The findings from the experiments indicate that the depth of a neural network significantly influences the mutation process. The research shows that neural networks with different depths require distinct approaches to mutation. It was observed that certain mutation operators that are effective on shallow networks cannot be directly applied in the same manner to deeper neural networks. This suggests that the complexity and structure of a network, as determined by its depth, play crucial roles in determining appropriate mutation strategies.

3.6 DeepCrime

Humbatova *et al.* [39] introduced a new tool for mutation testing of DNNs, which is unique compared to other papers and research on mutation testing for deep learning systems. Indeed, this tool is based on actual faults, distinguishing it from other tools that rely on simulated faults. They defined 35 DL mutation operators based on previous studies of existing fault taxonomies and tested 24 of them to determine which ones were killable and non-trivial. In particular, the authors proposed 35 mutation operators, with 27 of them acquired from the taxonomy by Humbatova and Jahangirova *et al.* [17], five from the work by Islam *et al.* [21], and three from the study of Zhang *et al.* [60]. The mutation operators are grouped by area of application. Specifically, there are eight groups, namely:

- Training Data: these operators attempt to imitate the issues potentially in the training dataset
- Hyperparameters: these operators mimic the choice of suboptimal values for the hyperparameters
- Activation function: these operators simulate the wrong choice of activation functions for a layer of a DNN model
- Regularisation: these operators produce manipulation of the penalties imposed on layers' kernel or manipulation of the dropout rate and patience
- Weights: these operators are used for altering the initial kernel initializer to a random or user-specified one

- Loss function: these operators are used for altering the initial loss function used to train to a random or user-specified one
- Optimisation function: these operators are used for changing the optimizer or values produced by Gradient clipping
- Validation: disabling the validation data during training

Particularly interesting is also the definition of:

- *killable mutation operator*: if there exists at least one configuration killed by the training data.
- *Contributing Input for Trained Model and Mutant*: When comparing an original model to its mutant, an input x is considered to contribute to "killing the mutant" if its output is different from the original's output (in the case of categorical output) or if it varies from the original's output by δ (in the case of continuous output), assuming that the input x is producing the correct inference y on the original model.
- *non-trivial mutation operator*: A mutant is considered trivial if it has a high triviality score, measured as the expected number of contributing inputs over the total number of inputs in the training set.
- *killing probability*: For some pairs of the original and mutated models, an input x may contribute, but it might not. For this reason, a killing probability was defined as the probability that the random variables $Y_N = N(x)$ and $Y_M = M(x)$ differ.
- *redundancy between mutants*: As well as in traditional mutation testing, an analysis of redundant mutants was made. In particular, a mutant M_1 is said to subsume another M_2 if all inputs that kill the mutant M_1 can kill also M_2.

DeepCrime searches for killable and non-trivial mutants. Using Python's Abstract Syntax Tree (AST) module, Deep-Crime parses the code that constructs and trains the deep learning model under test, and injects faults into it. As it traverses the parse tree's nodes, Deep-Crime identifies the target nodes where each mutation can be introduced into the code.

A statistical comparison was used to introduce a new notion of mutation killing. The comparison was made between the distribution of the random variable Y_N, which represents the output of the original network N, and the distribution of Y_M, which represents the output of the mutant M. This process involved re-training both the original model and its mutant multiple times, specifically n times each. The mutation is considered killed if, for a given test set $TestS$, the difference between the accuracies of the original and the mutants is statistically significant with non-negligible and non-small effect size. The predicate *isKilled* is defined as:

$$isKilled(N, M, TestS) = \begin{cases} true \ \ if \ \ effectsize(A_N(TestS), A_M(TestS)) \geqslant \beta \ \ and \\ \qquad p\_value(A_N(TestS), A_M(TestS)) < \alpha \\ \qquad\qquad false \ \ otherwise \end{cases}$$

Additionally, the MS (Mutation Score) was introduced. It is the proportion of mutation operator instances killed by both the training and test sets out of those killed by the training set.

For the experiments, the authors performed a total of 13400 retrainings. Their study shows that:

- DeepCrime produces a large number of killable, non-trivial mutants
- Mutation operators applied with extreme parameter values generate redundant mutants.
- DeepCrime's pre-training mutation operators are more sensitive to test set quality changes than DeepMutation++'s post-training mutation operators.

To conclude, DeepCrime operators were extracted from the existing studies on real DL faults, ensuring an approach based on real-world issues seen in deep learning, and can more effectively distinguish between weaker and stronger test sets compared to DeepMutation++.

3.7 Limitations of the current state-of-the-art tools of mutation testing for DL systems

The main limitations of these state-of-the-art techniques for mutation testing for DL systems are:

- The pre-training tool DeepCrime, while making meaningful changes, is computationally expensive and hard to apply in practice since it involves retraining
- The post-training tools DeepMutation and DeepMutation++, while being faster than pre-training tools, introduce random changes to DNNs in order to create mutants, which makes these mutations not realistic and less sensitive to the change in test data quality than pre-training ones

Chapter 4

Methodology

The main objective of this project was to create a mutation tool that can overcome the limitations of existing methods by proposing an alternative post-training approach. We aim to introduce smarter and fast mutation operators specifically for deep learning.

Our approach involves corrupting the model's behavior on a specific set of inputs from the dataset while retaining the model performance on the remaining data. This strategy simulates the cases in which inadequate training data are gathered in order to train the model, or the model doesn't learn properly certain features or specific types of input.

In order to have a useful separation between samples of the dataset that differ from each other based on their features, we applied clustering within each class of the dataset. Then, the clusters have been used as input for our mutation tool.

We decided to adopt two main strategies to build the mutants of a DNN model. The first approach used Arachne's Bidirectional Localisation in order to find the weights that are impactful on the model's outcome for a specific cluster. We localize the model's weights and add Gaussian noise to corrupt it deliberately to misbehave on a specific cluster. In addition, we employed Arachne' DE with a modified fitness function to perturb the localized weights in a more meaningful way. In fact, by applying DE to the localized model weights, the perturbation of these weights is more effective and targeted compared to the more random approach of Gaussian noise. DE's strategy of utilizing differences between existing solutions to explore the weight space results in a more structured and potentially more impactful exploration, leading to potentially better mutants. In addition, we explored an alternative approach. This second strategy still utilizes the power of clustering but seeks to address the creation of mutants for DNNs from a different angle with respect to the first approach. We present a technique that we've named "Targeted Misbehavior Retraining." In particular, in this approach, the full-trained DNN model is loaded, and then it is trained for a few epochs on a particular cluster within a class using incorrect labels in order to cause misbehavioral outcomes. Afterward, the model is trained again for one more epoch on the remaining data within the same class the cluster was taken from. This method ensures that the model misbehaves for a particular subset of input that shares specific features, without affecting the correct behavior of the network for the other subsets.

In this chapter, we will discuss the details of the approaches used in this study. In particular, the first part is about the strategy used to find an optimal combination of preprocessing, feature reduction, and clustering algorithms for grouping samples within the classes of the dataset. The

second part is about the localization of the weights with more impact on the output, giving a selected cluster as input. The third part examines the logic behind the creation of mutants and our "Targeted Misbehavior Retraining" method. The fourth section describes how we used a customized DE fitness function to find a patch that can disrupt a DNN's correct behavior for a specified input cluster. Lastly, we will explain the statistical techniques regarding the results obtained with this mutation tool.

4.1 Subjects

For this study, we have decided to use MNIST dataset as it is one of the most frequently used in the related literature experiments [31; 58; 39]. MNIST stands for "Modified National Institute of Standards and Technology" database and contains samples of hand-written digits. The samples are images of 28x28 pixels in greyscale containing numbers from 0 to 9. The entire dataset is composed of 60000 samples for the training set and 10000 samples for the test set. The selection of this dataset is due to the fact that it is widely used in machine learning and has been tested for diverse applications, making it suitable for our approach. In future work, we will extend the application of our approach to other subjects.

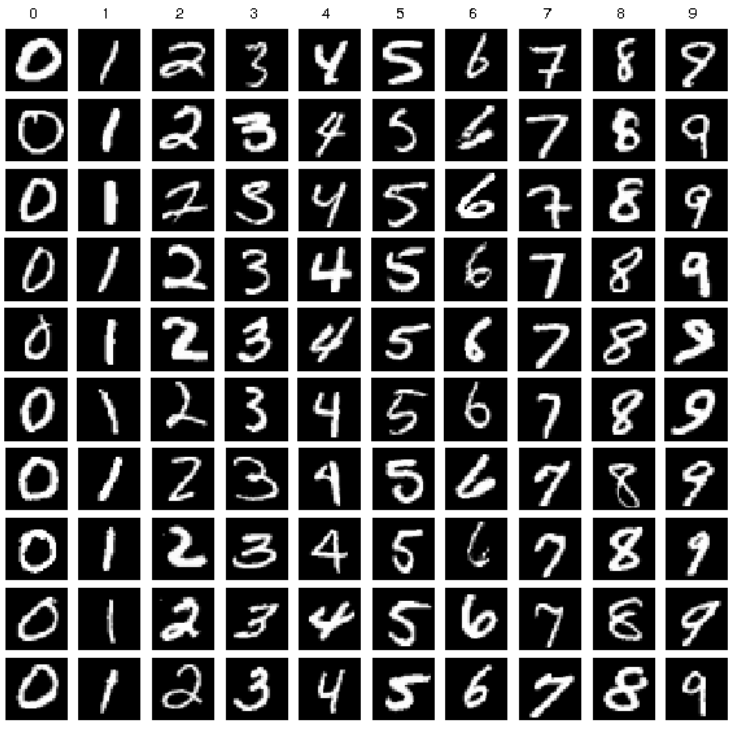


Figure 4.1. Example of images taken from the dataset [57]

4.2 Feature Reduction and Clustering

The first part of this project involved grouping the inputs based on the similarity of the features from the MNIST dataset. In order to do this, we consider several combinations of preprocessing methods, feature reduction techniques, and clustering algorithms (4.1). We apply each combination of approaches listed in Tab. 4.1 to each identified cluster of inputs in the classes of MNIST dataset. "None" means that no transformation on the MNIST images was applied. Each cell in tab. 4.1 represents a distinct step in the pipeline of the data processing and clustering. These steps, categorized under preprocessing and clustering, are foundational elements that are sequentially combined to form a complete workflow. Essentially, each step in one category (like thresholding in preprocessing) can be combined with steps in subsequent categories (for example, with Normalization for scaling, VAE for reduction, and KMeans for clustering). This allows for many combinations, where each unique sequence of steps forms a different pipeline. For example, one pipeline starts with thresholding, followed by MinMax scaling, then PCA reduction, and conclude with KMeans clustering.

Preprocessing			**Clustering**
thresholding	**Scaling**	**Reduction**	
Threshold 7.8%	Normalization	VAE	KMeans
None	Standard	VAE single class	Agglomerative
	MinMax	PCA	Gaussian Mixture Model
	Dividing by 255	UMAP	HDBSCAN
	None		

Table 4.1. Pipeline

4.2.1 Thresholding

Pourmohammad *et al.* [43] have shown that thresholding provides good results for the task of reducing the dimensionality of input data. It is evident from Figure 4.2 that there exist pixels with minimal activity across the entire dataset. Pixels with minimal activity mean that there are pixels that represent areas of the image that remain dark most of the time across the dataset without giving much information about the images. To assess the significance of each pixel in the overall dataset, we can compute their average value as follows:

$$\mu_t = \frac{1}{N_t} \sum_{d=0}^{9} \sum_{k=1}^{N_d} x_k \tag{4.1}$$

In this equation, x_k represents sample vector, and μ_t is the mean vector. N_t denotes the total number of data samples, while N_d corresponds to the number of data samples per digit.

To simplify the data, any pixels with an activity level below some pre-defined threshold tr are removed from the image (Fig. 4.2). We opted, as the authors of this paper suggested [43], for a threshold of about 7.8% of total pixel activity, which removes roughly 50% of original data. This decision was motivated by the fact that we could save computational resources by reducing the number of pixels (features) before applying any feature reduction technique.

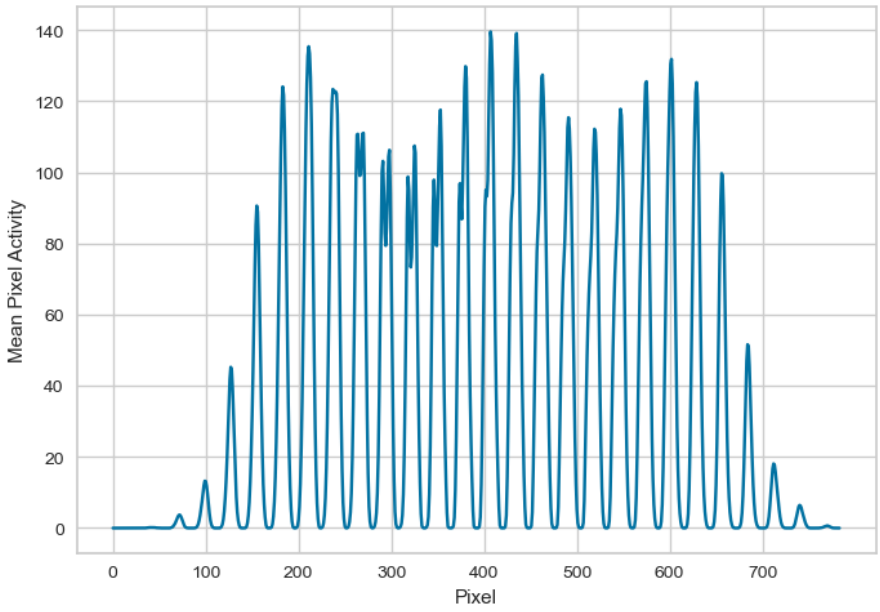


Figure 4.2. Mean pixel activity MNIST dataset

4.2.2 Feature Scaling

Before applying any feature reduction or clustering, several approaches are usually used to preprocess the input for the MNIST dataset [5]. For this reason, the first step was to try diverse transformations of the input images:

- dividing the pixels by 255 (maximum value for a pixel represented by 2 ^ 8 bits)
- Computing a single mean and standard deviation for all features across the entire dataset, treating all samples as a single set. All features were scaled based on the collective mean and standard deviation of the entire dataset.
- using a standard scaler, we scale the data's features independently. Thus, we computed the mean and standard deviation for each feature separately across the dataset. We then subtracted the mean and divided each pixel by this standard deviation (feature independent).
- using a MinMax scaler that reduces each pixel into a range [0, 1]. MinMax scaling works by subtracting the max value of each individual feature and by dividing this quantity by the difference between the maximum and minimum values. MinMax scaling indeed is defined as:

$$X_{\text{scaled}} = \frac{X - X_{\text{min}}}{X_{\text{max}} - X_{\text{min}}} \tag{4.2}$$

where X is a pixel value.

4.2.3 Dimensionality Reduction

Dimensionality reduction is a key step of this work, and in general in machine learning applications, to decrease the amount of computation demanded by subsequent steps such as clustering and to remove the noise within the data. Indeed, this step can diminish the feature vector representing a sample and, thus, the amount of data the clustering algorithms would need to utilize.

In order to reduce the feature dimension of the dataset, we employed PCA (section 2.2.1) (Principal Component Analysis) and VAE (section2.2.2) (Variational Autoencoder). PCA applies a linear transformation of the data to a lower dimensional space where most of the variation in the data can be described. Nevertheless, PCA cannot capture non-linear and complex relationships in the data. Thus, to as alternative approach, in this study, a VAE was trained to encapsulate the non-linear structure of the data while reducing its dimensionality. The term "VAE single class" within the table 4.1 refers to the VAE that was trained on a single class of the dataset. For this reason, we trained ten VAE, each one specifically on a specific class. In contrast, "VAE" in the same table refers to the VAE trained on the entire dataset.

In addition to PCA and VAE, we also incorporated UMAP [36] (section 2.2.3) (Uniform Manifold Approximation and Projection) as a tool for dimensionality reduction. UMAP is particularly effective in preserving both the local and global structure of the data in a lower-dimensional space. This makes UMAP a powerful technique for visualizing clusters or groups within the data, which is especially beneficial for datasets with complex, non-linear structures.

4.2.4 Clustering Algorithms

A crucial phase of our project entailed grouping samples within each class using clustering techniques. This approach served as an effective mean to group samples with similar features and facilitated creation of mutation operators based on corrupting the behavior of a model for a specific sets of inputs. By clustering, we could ensure that each subset of samples within a class shared common characteristics, enhancing the precision and relevance of our mutation operators.

To cluster the data within a class, four popular algorithms were employed: KMeans [33], Agglomerative clustering [22], HDBSCAN [7], and Gaussian mixture model [12].It is worth noting that algorithms can exhibit varying degrees of sensitivity to the shape and density of data. Therefore, it is essential to take into account the specific characteristics of the data. For example, the Kmeans groups the samples, assuming the clusters to be globular. In contrast, HDBSCAN is suitable to handle more complex shapes, such as non-convex clusters.

On the other hand, Gaussian mixture models assume that the data are generated by a finite number of Gaussian distributions with unknown parameters. Instead, Agglomerative clustering operates in a "bottom-up" fashion. Initially, every object is treated as a separate cluster (leaf). At each stage of the algorithm, the two most similar clusters are merged to form a larger cluster (node). This process is repeated until all points belong to a single large cluster.

The selection of different clustering algorithms can produce various clusters with different characteristics according to the data. The choice of different clustering algorithms was motivated by the goal of exploring a variety of approaches. Based on the set of evaluation metrics, the best-performing clustering can be selected.

4.2.5 Hyperparameters

Thoroughly assessing the performance of clustering algorithms on a given dataset after the reduction step necessitated the exploration of various hyperparameters. Numerous hyperparameters (Tab. 4.2) have been experimented with to identify the optimal combinations that resulted in optimal scores on the metrics.

Clustering	**Number of clusters**	**Min cluster size**	**Linkage**	**Covariance Type**
KMeans	from 2 to 5			
Agglomerative	from 2 to 5		ward, complete, average	
GaussianMixture	from 2 to 5			full, tied, diag, spherical
HDBSCAN		from 10 to 200, step size of 10		

Table 4.2. Selection of hyperparameters for clustering algorithms

In Tab. 4.2, we show the selected hyperparameters for our clustering step. Specifically, "number of clusters" is a hyperparameter that tells the algorithm how many clusters (and centroids in the case of the KMeans) must be constructed. Since we didn't have prior knowledge of how many clusters we needed, we tried different numbers of clusters. We restricted the range to 2 to 5 clusters because we did not expect too much variability within a class. The term "Min cluster size" as used in the context of Hierarchical Density-Based Spatial Clustering of Applications with Noise (HDBSCAN), pertains to the minimum number of sampled data points that must be present in a group for it to qualify as a cluster. This parameter enables the algorithm to distinguish between noise and actual clusters in a dataset. A group that fails to meet the minimum sample requirement is classified as noise. Lower minimum cluster sizes can reveal more nuanced and detailed patterns, potentially identifying clusters that might otherwise be missed or considered noise. On the other hand, larger minimum cluster sizes tend to generalize more, grouping data into larger clusters and possibly categorizing subtle structures as noise. "Linkage" refers to which linkage criterion to use. The linkage criterion is a parameter that decides which distance measure should be used. The algorithm aims to merge pairs of clusters that minimize a specific criterion. "Ward" works well for producing clusters of equal size since it concentrates on minimizing the variance within each cluster. On the other hand, "complete linkage" takes into account the greatest separation between points within two clusters, resulting in compact clusters that are more susceptible to outliers. By calculating the average distance between every pair of points in two clusters, "average linkage" provides reduced sensitivity to outliers and strikes a compromise between the sensitivity of the other two methods. For GMM clustering, we have chosen different types of Covariance matrix. Since every Gaussian component can have a different size, shape, and orientation, the 'full' covariance offers the greatest flexibility and is appropriate for complex datasets with intricate cluster shapes. When clusters are expected to have similar spread and orientation, the 'tied' covariance, where all components share the same matrix, ensures uniformity in cluster shapes across the dataset. The 'diagonal' covariance, which offers a compromise between flexibility and parameter simplicity and is helpful for moderately complex data, permits each component to have its own axis-aligned shape while restricting the orientation. Lastly, the most restricted type is the spherical type, which has one variance for every component and is effective for data that is more evenly distributed or simpler. It forces spherical shapes. The benefit of trying these different types in GMM is their ability to capture various data complexities, from highly intricate to simple cluster structures.

4.2.6 Clustering Evaluation Metrics

We adopted the Sihuette score [45] to evaluate our clustering results obtained using clustering algorithms, except HDBSCAN. The Sihuette score provides insight into the separation between the resulting clusters and the cohesion inside the clusters (how close the points are within a cluster). For HDBSCAN evaluation, we used the Density-based Validity Index (DBCV) [38]. This choice is motivated when we will describe DBCV in detail.

Silhouette score

Silhouette score ([45]) is a metric that measures the similarity of a sample to its own cluster compared to other clusters. For each point, i within a cluster C_I, $a(i)$ corresponds to the measure of how good point i is allocated to its cluster, and it is defined as:

$$a(i) = \frac{1}{|C_I| - 1} \sum_{j \in C_I, j \neq i} d(i,j) \tag{4.3}$$

where $|C_I|$ is the number of samples within the cluster and $d(i,j)$ is the distance between points i and j. Then, $b(i)$ is defined as the smallest mean distance between i and all the points of other clusters (in which i is not present):

$$b(i) = min_{J \neq I} \frac{1}{|C_J| - 1} \sum_{j \in C_J} d(i,j) \tag{4.4}$$

Finally, the silhouette score is defined as :

$$s(i) = \frac{b(i) - a(i)}{max\left\{b(i), \quad a(i)\right\}} \tag{4.5}$$

After having computed the clusters and all the silhouette scores for each sample, an overall silhouette score is calculated as the mean of these scores for each clustering result.

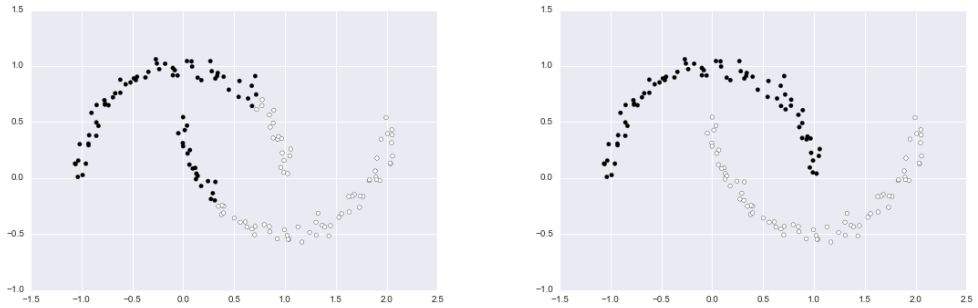
$$S_{clustering} = \frac{\sum_{i \in X} s(i)}{|X|}, \tag{4.6}$$

where $|X|$ is the number of considered samples. The range of this metric is $[-1, 1]$, with the best value (best clustering) equal to 1 and -1 the worst.

DBCV

As was mentioned in the work by Moulavi *et al.* [38], the Sihuette score works very well for globular clusters, but they may fail if the shape of the clusters is non-convex and not deal with noise objects. Instead, the authors proposed a new metric called DBCV (density-based cluster validation index) that computes the least dense region inside a cluster and the densest region between the clusters which are used to measure the within and between cluster density connectedness of clusters. Given that HDBSCAN establishes the clusters on the points' densities, we adopted DBCV as a metric to evaluate this particular clustering approach. The value of this metric ranges from -1 to 1, with higher values indicating better clustering.

An example of DBCV's application is provided in Fig. 4.3. The difference in performance observed between HDBSCAN (Fig. 4.3b) and KMeans (Fig. 4.3a) when using the silhouette



(a) DBCV = -0.71, silhouette score = 0.49 (b) DBCV = 0.60, , silhouette score = 0.32

Figure 4.3. Different clustering results using KMeans (a) and HDBSCAN (b). By visually inspecting the figure, HDBSCAN seems to cluster the data better with respect to KMeans. However, according to the silhouette score, it would be the opposite. Instead, DBCV suggests that HDBSCAN indeed performs better. This indicates that DBCV can evaluate better than other metrics the clustering results produced by HDBSCAN [9].

score on moon-shaped data highlights the sensitivity of clustering algorithms to the inherent structure of the data. Moon-shaped data is characterized by non-convex and crescent-shaped clusters, which poses a challenge for certain algorithms such as KMeans, affecting their ability to produce well-defined clusters. On the other hand, HDBSCAN performs better according to the Density-Based Cluster Validation (DBCV) score, suggesting that density-based metrics can be more effective in capturing the quality of non-convex clusters.

4.2.7 Post-Clustering Filtering

After clustering results had been obtained with metrics scores, a filter was applied in order to remove those combinations (a combination is a sequence containing a method from the thresholding, a method from our selected scaling, an algorithm from our chosen future reductions, and finally an algorithm from our considered clustering) whose results led to a single-cluster (in the cases of Gaussian Mixture model, KMeans, or Agglomerative clustering) or to only noise (in the case of HDBSCAN). Indeed, a single cluster does not provide any insights or differentiation within the data, which was our goal after clustering.

As the final step of this post-processing filtering, other combinations were removed in order to have meaningful final results after clustering the samples. In particular, we removed those combinations that did not meet two size criteria. A combination was discarded if both these conditions were met: the largest cluster contained more than 80% of the total number of samples, and the smallest cluster contained less than 5% of the total number of samples. This criterion was aimed at achieving a more balanced distribution of samples across clusters. This procedure is shown in Fig. 4.4.

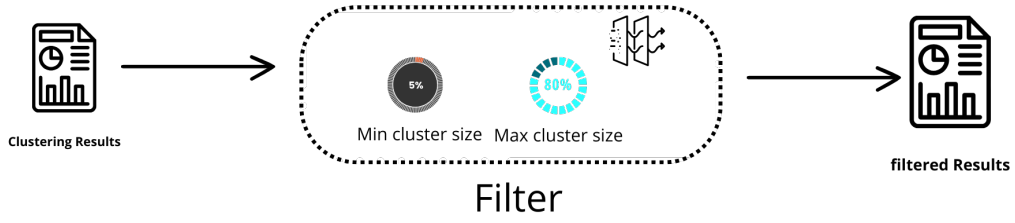


Figure 4.4. Post-Clustering filtering diagram

4.2.8 Optimal Combination Selection via Rank Scoring

In Tab. 4.3, we provide an example of the post-clustering filtered results. Considering the first column, the first row represents the class used; the second row shows the size of the clusters; and the third, fourth, and fifth rows illustrate the combination used. The remaining rows show the metrics, the parameters for the clustering, and finally, the time spent to find these clusters. In Tab. 4.3, vae_s is the abbreviation of VAE trained on a single class.

class	1	2	3	4	5
size clusters	[2955 1320 2467]	[2574 1749 2419]	[2580 1760 2402]	[2262 812 1616 2052]	[1987 1161 1643 1951]
features	10	10	10	10	10
thresholding	False	False	False	False	False
scaling	normalization	normalization	normalization	normalization	normalization
reduction	vae_s	vae_s	vae_s	vae_s	vae_s
clustering	kmeans	gauss mix	gauss mix	kmeans	gauss mix
silhouette	0.45	0.436	0.434	0.414	0.398
num clusters	3	3	3	4	4
params	...	...	...	...	...
time fit clustering	0.005	0.01	0.01	0.006	0.02

Table 4.3. Example of filtered results. Starting from the second column, each column represents a combination of thresholding, scaling, reduction, and clustering. In addition, the metric to evaluate the clustering are reported as well as the time (in seconds) used for clustering the data. "Params" refers to the hyperparameters used by the clustering algorithms.

Since the results produced by the tested clustering algorithms were applied independently to each class in the dataset, we would end up with different results for each class. This means that the best combinations for a class do not imply that they are the same for the other labels.

We used two distinct metrics for two separate groups of pipelines: (1) Silhouette for all pipelines excluding HDBSCAN and (2) DBCV for all pipelines, including those that incorporate HDBSCAN. This choice was made because Silhouette works very well for globular clusters, but they may fail if the shape of the clusters is non-convex and does not deal with noise objects. Non-convex clusters can have irregular shapes where connecting two points in the cluster with a straight line might pass outside of the cluster. DBCV, as explained in [9], properly handles the clusters formed by HDBSCAN, which clusters the data points based on their density.

Therefore, a ranking strategy was adopted to find the best combination among the classes. First, we sorted the results for each class by ordering the results based on the evaluation metrics. We organized the combinations of our clustering analysis based on the evaluation metrics by sorting them according to their performance scores. Specifically, we arranged the combina-

tions for the Silhouette scores in descending order, indicating that higher values for this metric signify better clustering performance. For further analysis, we considered only the three best combinations per class, i.e., the three ordered combinations that have led to the three highest metric scores.

Let us define $score_c(x)$ as the score of combination x for class c. The rank value $score_c(x)$ is defined as follows:

$$score_c(x) = \begin{cases} 3 & \text{if combination } x \text{ is the first best combination for class } c, \\ 2 & \text{if combination } x \text{ is the second best combination for class } c, \\ 1 & \text{if combination } x \text{ is the third best combination for class } c \\ 0 & \text{otherwise} \end{cases} \tag{4.7}$$

Assuming there are N classes, let us now define the overall rank as follows:

$$\text{overallRank}(x) = \sum_{c=1}^{N} \text{score}_c(x) \tag{4.8}$$

This is the total rank score of a specific combination (x), considering it across all classes. In this way, we can compute the best combination for all the classes, defined as:

$$best\ combination = \underset{x}{\text{argmax}}\ overallRank(x) \tag{4.9}$$

This combination, consisting of a specific preprocessing, reduction, and clustering step (with relative parameters), will be used in the *input selection* phase. Moreover, to get an approximation of what we would lose in terms of precision using this strategy, we computed the loss between the best metrics' score for a particular class and the metric score we have by using $best\ combination$. This loss is defined as follows:

$$loss_c = \max_{b \in B_c}(f(b)) - f(best\ combination) \tag{4.10}$$

where b is in B_c, B_c is the set of combinations for class c, and $f(\cdot)$ is a metric score specific for the cluster evaluation. In Tab. 4.4, we show the loss in precision when we select the best combination across the classes. We can see that we did not lose any precision for certain classes, like in the case of classes 4 and 8. Moreover, we adopted this ranking selection for two distinct pipelines: one is composed of those combinations that include KMeans, Agglomerative clustering, and GMM, whereas the other is composed of only those combinations that include HDBSCAN as a clustering algorithm. This is due to the fact that these two pipelines were evaluated by different metrics. Indeed, HDBSCAN clustering was evaluated using DBCV, while the results produced by other clustering were evaluated by silhouette score. We will present the two optimal pipelines in the chapter Experimental Evaluation 5.

Class	Score Best Combination	Best Score	Loss
0	0.005944	0.007240	0.001296
1	0.000000	0.018680	0.018680
2	0.0	0.005735	0.005735
3	0.0	0.005251	0.005251
4	0.002248	0.002248	0.0
5	0.0	0.041002	0.041002
6	0.0	0.004189	0.004189
7	0.0	0.008626	0.008626
8	0.005156	0.005156	0.0
9	0.004467	0.005010	0.000543

Table 4.4. Example of loss computed between the DBCV of best combination among classes and the best DBCV for a specific class. This example was taken from the results obtained through PCA and HDBSCAN. We reported this specific example because it is evident that in two cases, for classes 8 and 4, we did not lose any precision.

4.3 First Approach - Arachne

This section describes the methodology behind our weights localization approach and mutant creation using the Arachne tool.

4.3.1 Input Selection & Weights Localisation

Identifying the weights most impactful for the model's predictions was important for generating mutations within the DNN. In order to use the Arachne tool to do so, we needed to select a set of inputs to feed Arachne. The input selection phase consisted of picking inputs to provide Bidirectional localization method from the Arachne framework. Arachne selects an equal number of positive and negative inputs in the case in which the number of positive inputs is greater than the number of negative ones. Nevertheless, in order to preserve the accurate model's behavior for the remaining clusters (positive inputs), we have chosen to allow the localization method to accept arbitrary input sizes.

Positive inputs typically represent correctly classified samples for which we want to preserve the behavior of a model, while negative inputs are those that a DNN misclassifies and should be corrected after the application of Arachne. However, our approach diverged from the standard application. We aimed to identify those weights within the DNN that are important for the model's predictions for a particular cluster. Instead of using misclassified samples as negative inputs, we used samples within a cluster that were correctly classified, as shown in Fig. 4.5. This decision was made in order to let the model misbehave on correctly classified samples. From the previous phases, we ended up with a number of clusters for each class. From a specific cluster, we selected correctly classified samples to serve as our "negative" inputs—although, in our unique context, these were not really "negative" as Arachne defined them. We have chosen to maintain the established notation (negative and positive inputs) of Arachne for consistency in our methodology. The "positive" inputs were the correctly classified samples taken from all remaining clusters of the same label. This process was repeated for each cluster and each class.

The original Aarche's weight localization method requires a specific layer for analysis to

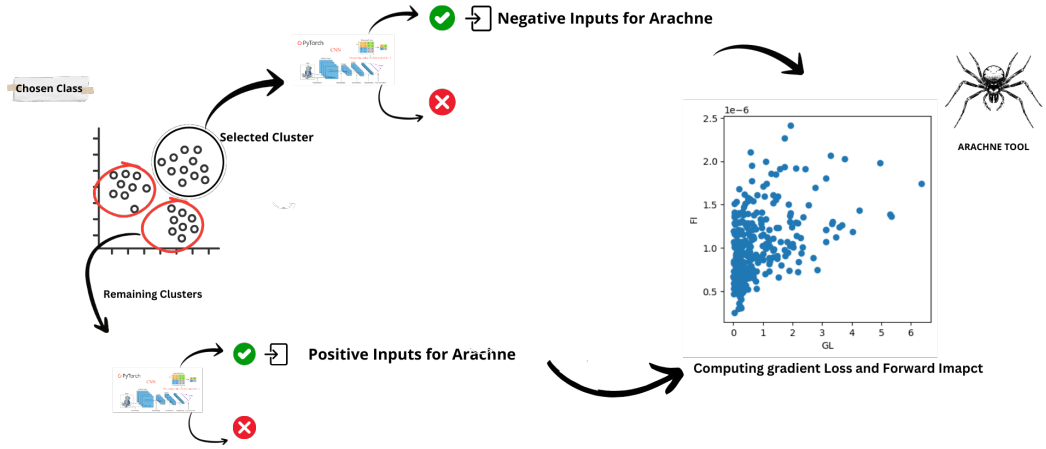


Figure 4.5. Input Selection and Weight Localization with Arachne: In this process, we first select a cluster from a specific class. For the 'Negative inputs', we use only those samples that our DNN model has correctly classified. Likewise, the 'Positive inputs' are comprised of correctly classified samples from other clusters within the same class of the dataset. The plot above on the right illustrates each weight's GL and FI values for a given layer.

be provided along with a set of negative and positive inputs. The algorithm returns a set of weights from this specific layer. We chose to utilize the Arachne Localization technique, paying close attention to the Deep Neural Network's (DNN) dense layer and its first two convolutional layers. We purposefully excluded the last layer from our analysis because we were worried that manipulating the last layer would cause an overly simplistic mutation. Corrupting the last layer would have amounted to compromising the critical decision-making layer far too easily. Our goal was to take a more sophisticated approach to mutation-building so that we could produce a trivially killable mutants.

4.3.2 DNN model

To conduct our experiments, we needed an already trained and saved Deep Neural Network (DNN) with reasonably high performance in order to effectively engage in the input selection and weight localization phases. To this end, we have adopted a Convolutional Neural Network (CNN) model that was used for the MNIST digit classification task in the related literature [39]. We have trained 20 instances of this model to mitigate the effects of randomness that can influence the outcomes of our study and to allow the use of the notion of statistical killing (discussed in Chap. 3).

Randomness in DNN studies can arise from various sources beyond weight initialization, such as the non-deterministic nature of the training algorithms, the order of input data presented during the learning process, and the stochastic nature of optimization techniques. Additionally, hardware-specific factors, such as GPU parallelism, can introduce variability, as can

the random selection of mini-batches during training.

By training multiple instances of our CNN model, we aimed to average out these sources of randomness, thereby obtaining a more robust and generalized understanding of the model's behavior. This approach allowed us to ensure that the insights gained from input selection and weight localization were not artifacts of random chance but were reflective of the true nature of the model's operation.

4.3.3 Mutants Creation - Gaussian Noise

In order to create mutants, we perturbed the weights of our already-trained DNN instances that were identified by the Arachne's localization method as shown in Alg. 3. The perturbation is represented by Gaussian noise with mean (μ) of 0 and variance (σ) of 1. This perturbation is applied to each of the localised weights.

Algorithm 3 MUTATE Function

```
1: function MUTATE(instance_DNN, weights_to_change)
2:     mutant ← COPY(instance_DNN)              ▷ Create a copy of the instance
3:     for all weight in weights_to_change do
4:         mutantWeight ← PERTURB(weight)       ▷ Perturb the weight
5:         UPDATEWEIGHT(mutant, mutantWeight)   ▷ Update the weight in the mutant
6:     return mutant                            ▷ Now mutant's weights are perturbed
```

The "Build Mutants" algorithm (Alg. 4) is designed to generate a list of mutated instances (or "mutants") of the DNN. It does so by iterating over a predefined number of DNN instances (20 in our experiments) and the selected classes of the dataset. Within each class, the algorithm iterates over the generated input clusters. "Localise weights" function calls the localisation method proposed by Arachne and identifies the weights with more impact on the outcome of the network based on correctly classified samples (I_{neg} and I_{pos}). The algorithm then applies targeted mutations to the identified weights. These mutations are based on selected inputs and the chosen layers in the network. The aim is to create a variety of mutants for each combination of instance, class, and cluster. The resulting mutants are then compiled into a list for further analysis. In order to obtain a more targeted and effective perturbation, we employed Arachne's Differential Evolution. This was an alternative solution to the random Gaussian noise for the creation of mutants.

4.3.4 Differential Evolution Patch for Selective Misprediction in DNNs

As an alternative to introducing random changes to the weights, we leveraged Arachne's Differential Evolution (DE) technique for automatic patch generation. DE is a genetic algorithm that is used by Arachne to find a patch of weights that repair the model by finding a solution that fixes the model's misbehavior for a particular set of inputs. To generate patches capable of causing mispredictions for a specific subset within a dataset class, we fed the DE algorithm a set of weights identified by Arachne as highly influential on the negative impact subset I_{neg} and less influential on the positive impact subset I_{pos}. Arachne's DE patches DNNs, which did not align directly with our goals. To address this, we adapted the fitness function of the DE process to align with our specific objective of inducing selective mispredictions. In particular, we defined the fitness function as follows:

Algorithm 4 Build Mutants

```
1: function BUILDMUTANTS(numberOfInstances, targetLayers)
2:     mutantsList ← []
3:     for instanceId ← 0 to numberOfInstances − 1 do
4:         instance ← LOADINSTANCE(instanceId)
5:         for class ← 0 to numClasses − 1 do
6:             for all cluster in clusters[class] do
7:                 I_neg, I_pos ← SELECT_INPUTS(instance, cluster)
8:                 for targetLayer in targetLayers do
9:                     weightsToChange ← LOCALISE_WEIGHTS(instance, targetLayers, I_neg, I_pos)
10:                    mutant ← MUTATE(instance, weightsToChange)
11:                    APPEND(mutantsList, mutant)
12:    return mutantsList
```

$$score_{neg}(i,X) = \begin{cases} 1 \text{ if } label_x(i) \neq label_{GT}(i) \\ Loss(i,X),\ otherwise \end{cases} \tag{4.11}$$

$$score_{pos}(i,X) = \begin{cases} 1 \text{ if } label_x(i) = label_{GT}(i) \\ \frac{1}{Loss(i,X)+1},\ otherwise \end{cases} \tag{4.12}$$

$$\text{fitness} = \sum_{i_p \in I_{pos}} score_{pos}(i_p,X) + \sum_{i_n \in I_{neg}} score_{neg}(i_n,X) \tag{4.13}$$

With respect to the original fitness function definition presented in the Arachne tool, we changed the score used for negative inputs $score_{neg}$. In our approach, this score is 1 if a negative input is misclassified by the model. Otherwise, if the negative input is still correctly classified by the model, the score will be proportional to the loss function. In this way, we could assign a higher score, and consequently, a higher fitness function, to those sets of weights, individuals in DE nomenclature, that improve the ability of the model to misbehave on the negative inputs. We kept the original score computed for the positive inputs since we wanted the model to correctly behave for those inputs belonging to the remaining clusters.

Unfortunately, we could not run the experiments on all 20 instances while using DE patch generation due to the amount of time and resources required by one run of DE (18 hours using only CPU). Hence, we run DE for one instance only.

Fig. 4.6 shows the process of mutant creation. In particular, on the left are the clusters within a class of the dataset. A plot illustrating the corresponding FI and GL values for each weight belonging to the target model's layer is on the bottom. the mutant creation process, illustrated in the center, involves perturbing the weights localized by Arachne.

4.3.5 Weight Clipping

We followed a procedure similar to that described in the Arachne paper, where the authors defined a particular search domain for new weights within a predefined range. We computed bounds to cap the weights using their formula 4.14, making sure they stay within these boundaries. In order to keep the weights from reaching unreasonably high or low values, this precau-

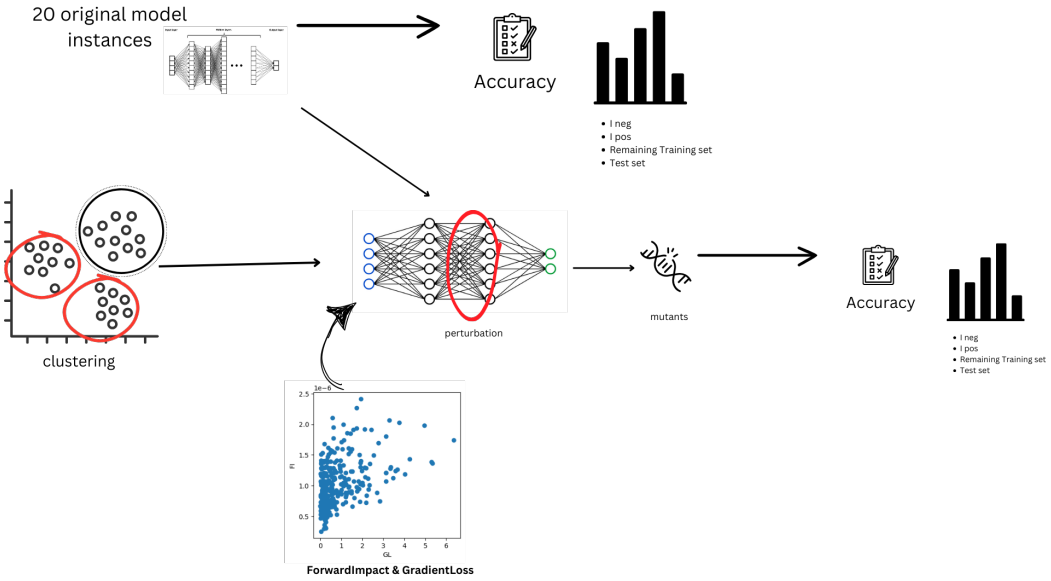


Figure 4.6. Overview mutants' creation process for the first approach using Arachne's Bidirectional Localisation to find the weights to perturb with random noise or DE algorithm.

tion is essential. Given the set of weights W extracted from a layer L:

$$
\begin{aligned}
lowerBound_L &= \min(W), \\
upperBound_L &= \max(W), \\
lowerBound_L &= \begin{cases} 2 \times lowerBound_L & if\ lowerBound_L < 0, \\ \frac{lowerBound_L}{2} & otherwise, \end{cases} \\
upperBound_L &= \begin{cases} 2 \times upperBound_L & if upperBound_L > 0, \\ \frac{upperBound_L}{2} & otherwise, \end{cases} \\
bounds &= (lowerBound_L, upperBound_L).
\end{aligned}
\tag{4.14}
$$

4.4 Second Approach - Targeted Misbehavior Retraining

As an alternative to Arachne, we have designed an approach we named "Targeted Misbehavior Retraining." Given a selected cluster, we intentionally corrupted its inputs by changing their labels to wrong ones. This way, we can simulate real-world faults such as low quality of data. This approach involves the pre-trained DNN models (the same used for the Arachne experiment) for N additional epochs using only the samples in a specific cluster, with the goal of inducing misbehavior in the model specifically for that cluster. In addition, in order to retain the correct behavior for the inputs within the remaining clusters, we trained the model for an additional epoch on these inputs (belonging to the remaining clusters, denoted as positive inputs).

These mutants are designed to specifically mispredict a particular subset within a class.

As in the previous approach, only the samples that did not reveal misbehavior for the considered model DNN are used as input for our retraining. Thus, only correctly classified samples from the clusters were taken into account. We use the term C_i to refer to a specific clus-

Algorithm 5 Targeted Misbehavior Retraining function

```
1: function TARGETED MISBEHAVIOR RETRAINING FUNCTION(model, C_i, OtherClusters, class𝒞)
2:     N                                          ▷ N is the number of retraining epochs
3:     L_mis ← secondMostProbableLabel(model, C_i)
4:     mutant ← retrain(model, C_i, L_mis, N)
5:     // Retrain the mutant for another epoch on the other clusters
6:     mutant ← retrain(mutant, OtherClusters, 1)
7:     // Evaluate the mutant model
8:     accuracyCi ← evaluate(mutant, C_i, 𝒞)
9:     accuracyOther ← evaluate(mutant, OtherClusters, 𝒞)
10:    // Return the final accuracies and the mutant model
11:    return accuracyCi, accuracyOther, mutant
```

ter produced by our clustering algorithms on a class $\mathscr{C}_i$ of the dataset and use $\bigcup_{j \neq i} C_j$ (or $otherClusters$) to refer to the other clusters from class $\mathscr{C}_i$. As shown in Alg. 5, to corrupt the ground truth labels of a cluster, we extracted as "misleading labels" L_{mis} those classes that the model predicts as the second most probable classes in its predictions by taking as input the set of C_i. Retraining the model with the second most likely label to gently skew behavior on a particular cluster C_i without radically altering its general learning patterns. The idea is that the intervention we introduce should be gentle enough not to significantly alter the model's preexisting behavior for other clusters of the selected class or the remaining classes in the dataset. To ensure that our results are not affected by the randomness inherent to model training, we have applied this strategy to 20 instances of the already trained model.

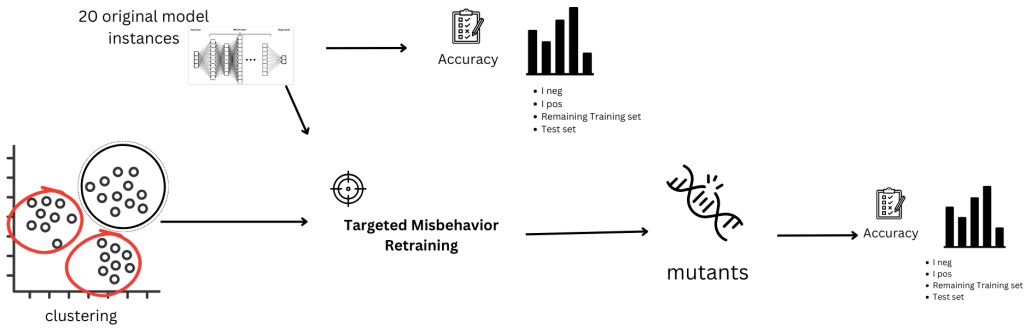


Figure 4.7. Overview mutants' creation process for the second approach using our Targeted Misbehavior Retraining.

Fig. 4.7 shows the process of our TMR approach. On the left, it illustrates the clustering of the data belonging to a class of the dataset. Then, the clusters are given to TMR as input. TMR then creates the mutants, which are used for further evaluation.

4.5 Evaluation

We evaluated our approaches by examining the percentage of mispredictions on both positive and negative inputs to effectively gauge the efficiency of our mutants (Eq. 4.15). A higher misprediction rate for the selected input cluster (set negative defined as I_{neg}) indicates a more effective mutation, highlighting its successful impact. At the same time, a lower misprediction rate for positive inputs is desirable, as it signifies the retention of high accuracy across the remaining clusters and proves that impact is targeted. We refer to $\#corrupted\ I_{neg}$ as the number of negative inputs misclassified by the mutated model and to $\#I_{neg}$ as the total number of inputs in the set negative (same for I_{pos}).

$$\begin{aligned} MispredictionRatio_{neg} &= \frac{\#\ corrupted\ I_{neg}}{\#\ I_{neg}} \\ MispredictionRatio_{pos} &= \frac{\#\ corrupted\ I_{pos}}{\#\ I_{pos}} \end{aligned} \tag{4.15}$$

To determine the extent to which the mutations impacted the model's overall behavior, we computed the accuracies for both original and mutated models on I_{neg}, I_{pos}, the training set (excluding the class considered during clustering), the test set (excluding the class considered during clustering) and the test set of the class considered for clustering.

Next, we employed statistical tests to determine if there was a significant difference between the accuracies of the original models and their mutated counterparts. To achieve this, we calculated both the p-value and the effect size. In particular, we employed the non-parametric Wilcoxon signed-rank test in our analysis [55] for computing the p-value and Cohen's d for computing the effect size [10]. Specifically, we considered the two groups to be statistically different if the p-value was below 0.05 and, concurrently, the effect size was greater or equal to 0.5. First of all, we should mutate the model enough to produce significant changes on the set negative (inputs that we want a mutated model to misclassify). Ideally, there should be no statistically significant difference observed on the inputs belonging to set positive. Additionally, we aimed to verify that there was no statistical significance in the test and training sets when excluding the targeted class. Conversely, our goal was to observe statistical differences in the test set that included only the clustered class. This entire evaluation process is illustrated in Fig. 4.8. It shows that the accuracies of 20 original model instances are compared to the accuracies of the 20 corresponding mutants.

As a final step of the evaluation, we check whether the mutants that we have generated are sensitive to the change in the quality of test data. To do so, we compare the average drop in accuracy of the 20 instances of original and mutated models calculated on the strong and the weak test sets. We have adopted the test set for the MNIST dataset from the replication package of DeepCrime [18].

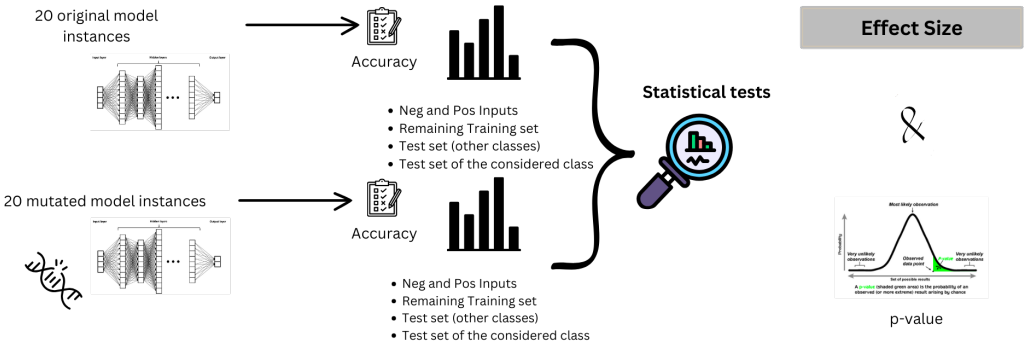


Figure 4.8. Evaluation pipeline. We evaluated the accuracies of both original and mutated models. We then computed their accuracies on I_{neg}, I_{pos}, the training set (excluding the class considered during clustering), the test set (excluding the class considered during clustering) and the test set of the class considered for clustering.

Chapter 5

Experimental Evaluation

In this chapter, we discuss the experimental procedure and the results obtained during our study.

5.1 Experimental Setup & Procedure

In this section, we will report the libraries, the hardware, and the model structures used in our experiments.

5.1.1 Libraries and Hardware

For this project, we used Python [53] due to its extensive libraries for machine learning. In particular, we used the following libraries:

- Pytorch [41] to implement VAE
- TensorFlow [1] and Keras [8] for the creation of mutants, the training and testing, and class of the original model
- numpy [15] for mathematical operations
- matplotlib [19] for displaying the results and generating plots
- sci-kit learn [42] for the use of PCA and clustering algorithms
- hdbscan module [30] to use DBCV implementation
- umap module to use UMAP [35]
- pandas [40] for working with the data

All the experiments were conducted using an Apple MacBook Pro M1 2020 equipped with an Apple M1 chip with an 8-core CPU, 8-core GPU, and 16-core Neural Engine.

5.1.2 Training VAE

We implemented our own VAE as shown in Tab. 5.1. Specifically, we used a convolutional VAE since we needed to deal with images from MNIST dataset. This decision was motivated by the ConvVAE's superior capacity to extract local features and spatial hierarchies from image data by utilizing convolutional layers, which are naturally well-suited to handle the structural complexities found in handwritten digits. This architectural change is anticipated to improve the model's capacity to learn intricate patterns, which will ultimately result in the creation and reconstruction of MNIST images with greater accuracy.

Layer	Type	Configuration
conv1	Conv2d	1, 16, kernel_size=(5, 5), stride=(2, 2)
conv2	Conv2d	16, 32, kernel_size=(5, 5), stride=(2, 2)
linear1	Linear	in_features=512, out_features=300, bias=True
mu	Linear	in_features=300, out_features=5, bias=True
logvar	Linear	in_features=300, out_features=5, bias=True
linear2	Linear	in_features=5, out_features=300, bias=True
linear3	Linear	in_features=300, out_features=512, bias=True
conv3	ConvTranspose2d	32, 16, kernel_size=(5, 5), stride=(2, 2)
conv4	ConvTranspose2d	16, 1, kernel_size=(5, 5), stride=(2, 2)
conv5	ConvTranspose2d	1, 1, kernel_size=(4, 4), stride=(1, 1)

Table 5.1. Our Convolutional VAE layers

As explained in the Methodology chapter 4, we used the VAE to reduce the dataset's features and remove the noise. We trained our model for 100 epochs. As shown in Fig. 5.1, one can easily see that our model performs reasonably well in reconstructing the original image.

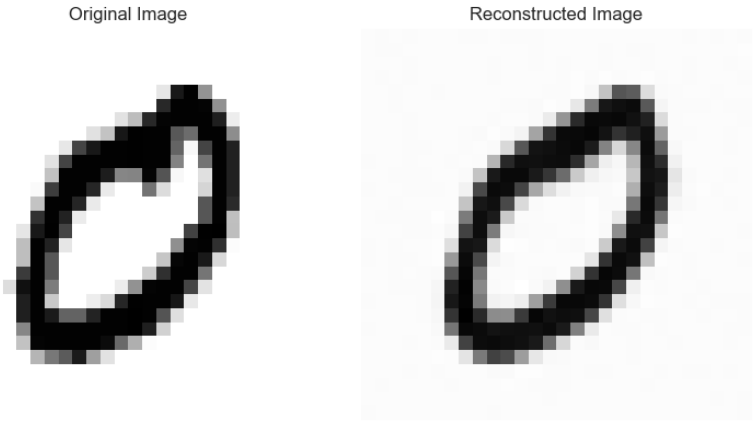


Figure 5.1. Original image and reconstructed image by VAE

Our primary motivation for adopting this was to attain successful feature reduction. In order to condense the essential features of the MNIST images into a more manageable format, we deliberately encode the input image into a 10-dimensional latent space. Through experi-

mentation, we found that a 10-dimension latent space VAE was a good compromise between accuracy, reconstruction and speed.

Then, we implemented 10 different VAEs (10-dimensional latent space), denoted with VAE_s in the methodology section, each trained on a single class of the dataset.

5.1.3 DNN Model

In this project, we used a simple CNN model trained on the MNIST dataset [18]. We trained it for 12 epochs. We ended up with an accuracy of about 98% on the test set. In Tab. 5.2, we provide details about the model architecture and the number of trainable parameters.

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 26, 26, 32)	320
re_lu (ReLU)	(None, 26, 26, 32)	0
conv2d_1 (Conv2D)	(None, 24, 24, 64)	18,496
re_lu_1 (ReLU)	(None, 24, 24, 64)	0
max_pooling2d (MaxPooling2D)	(None, 12, 12, 64)	0
dropout (Dropout)	(None, 12, 12, 64)	0
flatten (Flatten)	(None, 9216)	0
dense (Dense)	(None, 128)	1,179,776
re_lu_2 (ReLU)	(None, 128)	0
dropout_1 (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 10)	1,290
softmax (Softmax)	(None, 10)	0
Total params		1,199,882
Trainable params		1,199,882
Non-trainable params		0

Table 5.2. DNN Model Summary

5.1.4 Configuration

We created 720 mutants, Tab. 5.3, using Arachne's localization. Indeed, we had: 20 original model instances, three classes from the dataset, three clusters per class, four different targeted layers . We focused our targeting on three specific layers of our model: layer zero and two, which are the two convolutional layers, as well as the second last layer, which consists of 128 neurons. "All layers" refers to targeting all these layers. Instead, in our approach using TMR, we created a total of 160 mutants since we had 20 original model instances, three classes, three clusters for classes 0 and 5, and two clusters for class 7.

Mutants			
Instance_ID	Label_ID	Cluster_ID	Mutants files .h5 format
Instance_0	Label_0	Cluster_0	all_layers, layer_0, layer_2, layer_7
		Cluster_1	all_layers, layer_0, layer_2, layer_7
		Cluster_2	all_layers, layer_0, layer_2, layer_7
	Label_5	Cluster_0	all_layers, layer_0, layer_2, layer_7
		Cluster_1	all_layers, layer_0, layer_2, layer_7
		Cluster_2	all_layers, layer_0, layer_2, layer_7
	Label_7	Cluster_0	all_layers, layer_0, layer_2, layer_7
		Cluster_1	all_layers, layer_0, layer_2, layer_7
		Cluster_2	all_layers, layer_0, layer_2, layer_7
...			
Instance_19	Label_0	Cluster_0	all_layers, layer_0, layer_2, layer_7
		Cluster_1	all_layers, layer_0, layer_2, layer_7
		Cluster_2	all_layers, layer_0, layer_2, layer_7
	Label_5	Cluster_0	all_layers, layer_0, layer_2, layer_7
		Cluster_1	all_layers, layer_0, layer_2, layer_7
		Cluster_2	all_layers, layer_0, layer_2, layer_7
	Label_7	Cluster_0	all_layers, layer_0, layer_2, layer_7
		Cluster_1	all_layers, layer_0, layer_2, layer_7
		Cluster_2	all_layers, layer_0, layer_2, layer_7

Table 5.3. Mutants' configuration in our approach using Arachne

5.1.5 Customizing Arachne's Bidirectional Localization

Problem Formulation. While employing Arachne's code for Bidirectional Localisation in our study, we encountered an issue where the Forward Impact (FI) metric consistently registered as zero. Upon investigation, we identified the root cause as our oversight in accommodating models whose final layer is a softmax activation layer. This issue becomes particularly pronounced in well-trained models exhibiting high accuracy, as in our case. In such models, the softmax layer's output tends to be constant. Given that a component of the computation of the FI is the gradient of the model's output with respect to the output of a specific neuron, the unvarying output of the softmax layer leads to a zero gradient. This anomaly significantly impacts our ability to accurately identify the crucial weights within the network that influence the output for a given sample. Indeed, both I_{neg} and I_{pos} were correctly classified by the network. In well-trained models, such as our pre-trained models, the softmax output can become invariant for a given input that the model classifies with high confidence. This is because the softmax function, defined as $\frac{\exp(x-max(x))}{\sum\exp(x-max(x))}$, exponentially scales the logits and normalizes them. As a consequence, any large negative logits become exponentially smaller than the positive logits and are thus effectively negligible when the softmax is applied. This scaling can cause the logits that do not correspond to the correct class to approach zero, as seen in Fig. 5.2. This happens of course when there is a very huge difference between the logits (so the logit value corresponding to the correct class is very large compared to the other logit values). This phenomenon led to the network generating a constant output value, irrespective of variations in the input, consequently resulting in the gradient $\frac{\partial O}{\partial o}$ being reduced to zero.

*Proposed Solution.*Instead of using the final output of the network, denoted as O, we chose

to use the logits computed before the application of the softmax layer when dealing with a network that has a softmax layer for the final layer. Then, to prevent numerical instability and avoid issues with exploding gradients, we normalize these logits.

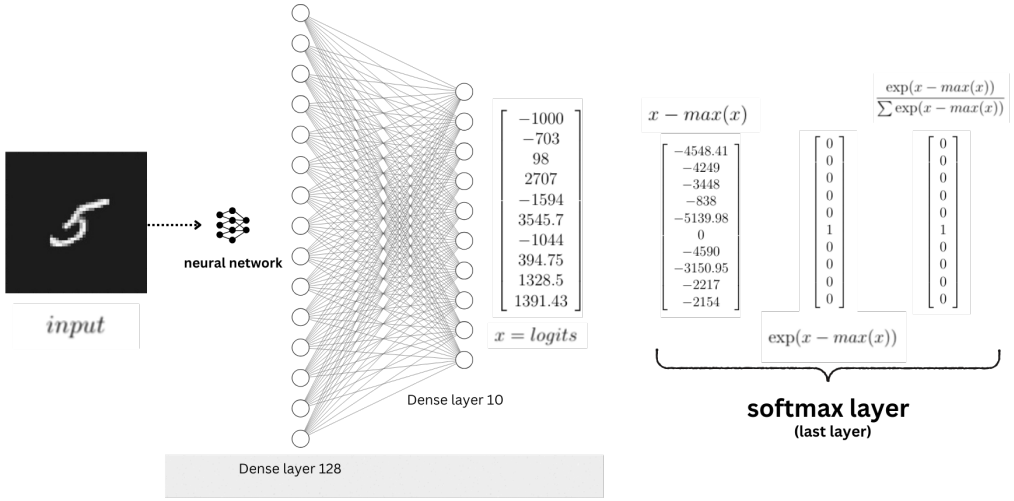


Figure 5.2. An example of how to compute the output of a network given a correctly classified sample and a high confidence level. Note that subtracting the maximum value of the logits from the logits is a technique used to avoid numerical instability during computation.

5.1.6 DE parameters

In all experiments we used: population size of 100 individuals and 25 generations. Additionally, we have incorporated an early stopping mechanism. If the fitness function does not improve for 10 consecutive generations, the process will halt.

Furthermore, we opted to keep the original population initialization method (used by the original Arachne tool), which involves initializing individuals in proximity to model layer values. Specifically, we calculated the mean and standard deviation of the weights within each layer (considering all the weights in that layer), and then generated initial values by sampling from a normal distribution with these parameters.

5.2 Clustering Results

Tab. 5.4 presents an example of results from our experimental approach, containing various combinations of preprocessing, feature reduction, and clustering techniques. The results show the specific class under consideration, the detailed pipeline employed to achieve the final clustering, and the dimensionality feature reduction. Additionally, the table enumerates the sample count within each cluster. It also provides a comprehensive evaluation of the clustering effectiveness through pertinent metrics, details the parameters utilized in the clustering process,

and notes the time duration (in seconds) required to fit the clustering model. In total, we tried 17040 combinations of preprocessing, feature reduction, clustering, and hyperparameters. This number comes from considering two combinations for the first step of preprocessing (thresholding), four types for the second step of preprocessing (feature scaling), four types of feature reduction, and 71 combinations of clustering algorithms and hyperparameters. We excluded combinations where thresholding and either *VAE* or VAE_s were used because VAEs were trained with 28*28 input sizes and could not work with different input dimensions.

class	clusters size	Reduction	Preprocessing	num_features	clustering	silhouette	# clusters	params	time fit clustering (s)
0	[3365 2558]	vae_single_label	normalization	10	kmeans	0.329	2	{..}	0.028
0	[2778 1241 1904]	vae_single_label	normalization	10	kmeans	0.242	3	{..}	0.005
0	[1609 1171 1602 1541]	vae_single_label	normalization	10	kmeans	0.219	4	{..}	0.007
0	[1327 1537 1093 1389 577]	vae_single_label	normalization	10	kmeans	0.198	5	{..}	0.006
0	[2360 2078 1485]	pca	normalization	62	agglomerative	0.080	3	{..}	0.004
1	[2955 1320 2467]	vae_single_label	normalization	10	kmeans	0.456	3	{..}	0.006
1	[2262 812 1616 2052]	vae_single_label	normalization	10	kmeans	0.414	4	{..}	0.006
1	[1969 453 1133 1308 1879]	vae_single_label	normalization	10	kmeans	0.383	5	{..}	0.009
2	[2729 3229]	vae_single_label	normalization	10	kmeans	0.282	2	{..}	0.004

Table 5.4. Some of the achieved results by preprocessing, feature reduction, and clustering for each dataset class.

5.2.1 Post-clustering Filtering results

We found that there was a discernible disparity in cluster sizes among the different clustering outcomes after evaluating our data. In order to tackle this, we employed a filtering criterion to enhance the quality of our analysis, as explained in the methodology chapter. After this filtering, we got 4621 combinations in total, including those combinations that incorporate HDBSCAN. A portion of our findings, after applying the aforementioned filter, are shown in Tab. 5.5 and Tab. 5.6. It is significant to underline that the first cluster in the case of HDBSCAN is reserved for outliers. Thus, we applied our filtering procedure only to the other clusters, excluding this outlier group.

class	clusters size	num_features	preprocessing	reduction	clustering	DBCV	n_clusters	min_cluster_size	time fit clustering
5	[4459 697 265]	105	standard	pca	hdbscan	0.04	3	15	3.158
5	[4538 657 226]	105	standard	pca	hdbscan	0.04	3	20	3.194
5	[4626 606 189]	105	standard	pca	hdbscan	0.03	3	25	3.224
5	[4689 571 161]	105	standard	pca	hdbscan	0.03	3	30	3.251
5	[4763 146 512]	75	dividing_by_255	pca	hdbscan	0.03	3	25	1.976

Table 5.5. An example of the clustering results obtained using HDBSCAN after applying the filter.

5.2.2 Optimal Combination Selection via Rank Scoring

After filtering the results, we selected a single combination that worked best for all the classes in our dataset and resulted in optimal metric scores. Even with the filtering, we still had more than 4000 combinations to choose from. Following the implementation of our ranking selection process, as detailed in the methodology chapter, we arrived at two distinct sets of results.

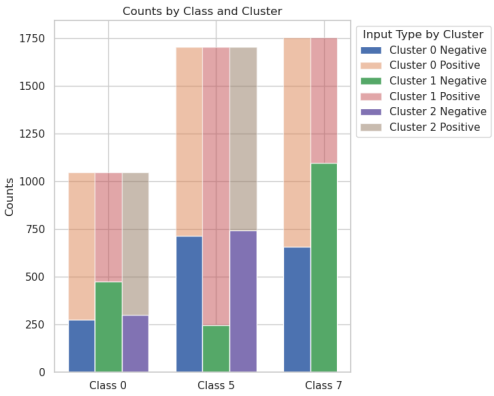
class	clusters size	num_features	second step	third step	clustering	silhouette	n_clusters	time fit clustering
1	[2955 1320 2467]	10	normalization	vae_single_label	kmeans	0.456	3	0.006
1	[2574 1749 2419]	10	normalization	vae_single_label	gaussian_mix	0.437	3	0.017
1	[2580 1760 2402]	10	normalization	vae_single_label	gaussian_mix	0.435	3	0.017
1	[2262 812 1616 2052]	10	normalization	vae_single_label	kmeans	0.414	4	0.006
1	[1987 1161 1643 1951]	10	normalization	vae_single_label	gaussian_mix	0.399	4	0.025

Table 5.6. An example of the clustering results obtained by KMeans, Agglomerative clustering, and Gaussian Mixture model after applying the filter.

One set emerges from the combinations utilizing KMeans, Gaussian Mixture models, or Agglomerative clustering, while the other set is derived from the use of HDBSCAN. In particular, for the former pipeline, we obtained the best combination: as preprocessing Normalisation, as dimensionality reduction, VAE trained on all the classes, as clustering KMeans with the number of clusters parameter equal to 3. Instead, for the second pipeline that includes HDBSCAN (minimum cluster size parameter equal to 200) as a clustering algorithm, we obtained the best combination: Normalization as preprocessing and UMAP as feature reduction.

To align with our ultimate objective of utilizing these clusters for identifying neural network weights via the Arachne tool, we strategically chose to focus on a subset of three classes (0, 5, and 7) for more detailed analysis. This decision was driven by several factors. Firstly, concentrating on a smaller set of classes allows for more in-depth and precise investigation, particularly important when working with sophisticated tools like Arachne. Secondly, narrowing our focus to these classes enables a more efficient allocation of resources and time, important for the intensive computational demand by both Arachne and TMR. We choose these particular classes because the digits they represent are of different shapes that might help us to get more insightful results.

(a) These inputs are extracted using KMeans clustering (after reducing the dimensionality with VAE) and filtering them by taking only those correctly classified by the DNN model.



(b) These inputs are extracted using HDBSCAN clustering (after dimensionality reduction performed by UMAP) and filtering by taking only those correctly classified by the model. Anomalies found by HDBSCAN are not considered clusters.

Figure 5.3. Number of negative and positive inputs for each class.

We obtained, by using our ranking approach, three clusters for classes 0, 5, and 7 using VAE and KMeans. In contrast, we obtained three clusters for classes 0 and 5 and two clusters for class 7 using the best combination composed of UMAP and HDBSCAN. We filtered samples within the cluster based on our trained DNN models. Samples that showed misbehavior were discarded. This way, we kept only the correct-classified samples to use for our mutation tool. In both figures Fig. 5.3a and Fig. 5.3b, the number of I_{neg} (one selected cluster), and I_{pos} (remaining clusters) are shown for the two respective pipelines.

5.3 Gaussian Noise Weights Perturbation

By using our customized Arachne's Bidirectional Localization method, we introduced Gaussian noise to the determined weights. However, the experimental outcomes were not favorable. Based on the results provided in Tab. 5.7 (clusters obtained using UMAP and HDBSCAN) and 5.8 (clusters obtained using VAE and KMeans), we can observe that introduction of this noise led to a low ratio of mispredictions for both positive and negative inputs (less than 1 % for both sets on average). Indeed, the percentage of I_{neg} mispredicted is very low indicating that this approach did not work.

class	cluster	target_layer_idx	I neg mispredicted %	I pos mispredicted %
0	0	0	0.092	0.078
		2	0.037	0.006
		7	0.037	0.000
		all_layers	0.148	0.026
	1	0	0.074	0.150
		2	0.021	0.009
		7	0.011	0.009
		all_layers	0.042	0.061
	2	0	0.017	0.088
		2	0.000	0.007
		7	0.000	0.007
		all_layers	0.050	0.000
5	0	0	0.070	0.097
		2	0.007	0.051
		7	0.014	0.015
		all_layers	0.049	0.031
	1	0	0.162	0.165
		2	0.041	0.062
		7	0.000	0.014
		all_layers	0.143	0.082
	2	0	0.218	0.047
		2	0.089	0.016
		7	0.041	0.005
		all_layers	0.048	0.031
7	0	0	0.123	0.202
		2	0.023	0.046
		7	0.031	0.028
		all_layers	0.077	0.041
	1	0	0.124	0.054
		2	0.032	0.008
		7	0.156	0.000
		all_layers	0.133	0.031

Table 5.7. Results of adding Gaussian noise to the weights targeted by Arachne's Localisation. The clusters are found by applying UMAP and HDBSCAN. The values in the last two columns are an average across all the 20 model instances. The column "target layer idx" displays the layer of the DNN that was targeted during the Localisation phase.

class	cluster	target_layer_idx	I neg mispredicted %	I pos mispredicted %
0	0	0	0.168	0.165
		2	0.041	0.025
		7	0.012	0.008
		all_layers	0.023	0.014
	1	0	0.073	0.080
		2	0.027	0.047
		7	0.015	0.016
		all_layers	0.024	0.029
	2	0	0.139	0.227
		2	0.000	0.043
		7	0.000	0.017
		all_layers	0.167	0.201
5	0	0	0.217	0.303
		2	0.067	0.070
		7	0.049	0.023
		all_layers	0.165	0.168
	1	0	0.197	0.242
		2	0.115	0.145
		7	0.029	0.054
		all_layers	0.282	0.242
	2	0	0.353	0.219
		2	0.041	0.026
		7	0.030	0.046
		all_layers	0.092	0.069
7	0	0	0.185	0.171
		2	0.061	0.063
		7	0.047	0.043
		all_layers	0.021	0.046
	1	0	0.515	0.424
		2	0.081	0.098
		7	0.027	0.075
		all_layers	0.027	0.182
	2	0	0.243	0.275
		2	0.139	0.147
		7	0.029	0.028
		all_layers	0.058	0.043

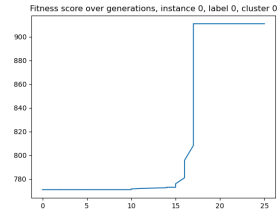
Table 5.8. Results of adding Gaussian noise to the weights targeted by Arachne's Localisation. The clusters are found by applying VAE and KMeans. The values in the last two columns are an average across all the 20 model instances. The column "target layer idx" displays the layer of the DNN that was targeted during the Localisation phase.

5.4 DE Results

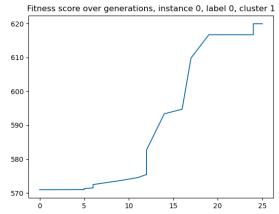
Alternatively to Gaussian noise, following the localization phase, we proceeded to execute the Differential Evolution (DE) algorithm using these localized weights. Our investigation consisted of four distinct runs in total: two runs for each approach (VAE+KMeans and UMAP+HDBSCAN), with one run involving weights clipping and the other without.

It's crucial to emphasize that, in the case of Differential Evolution (DE), we employed only a single instance of our trained Deep Neural Networks due to the extensive time required by the DE algorithm.

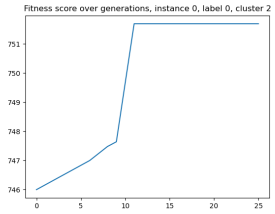
In Fig. 5.4 and Fig. 5.5 that show the behavior of our fitness function across multiple generations, we observe a clear trend of convergence. This consistent convergence over successive generations provides an evidence of the effective of our fitness function.



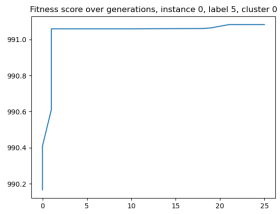
(a) Cluster 0, class 0



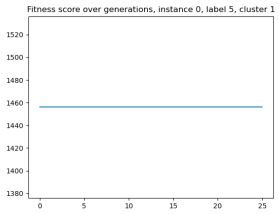
(b) Cluster 1, class 0



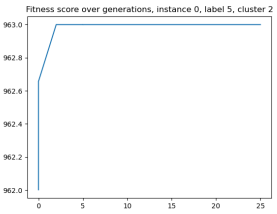
(c) Cluster 2, class 0



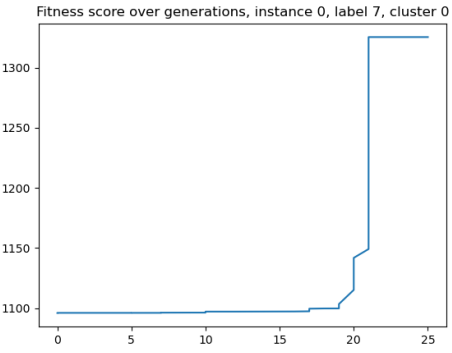
(d) Cluster 0, class 5



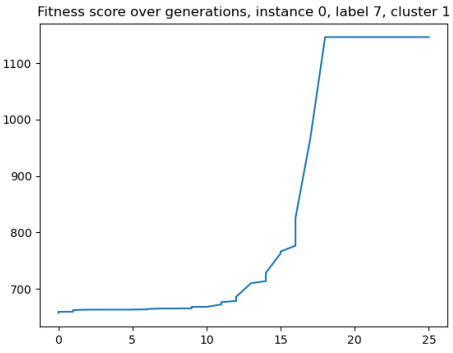
(e) Cluster 0, class 5



(f) Cluster 0, class 5

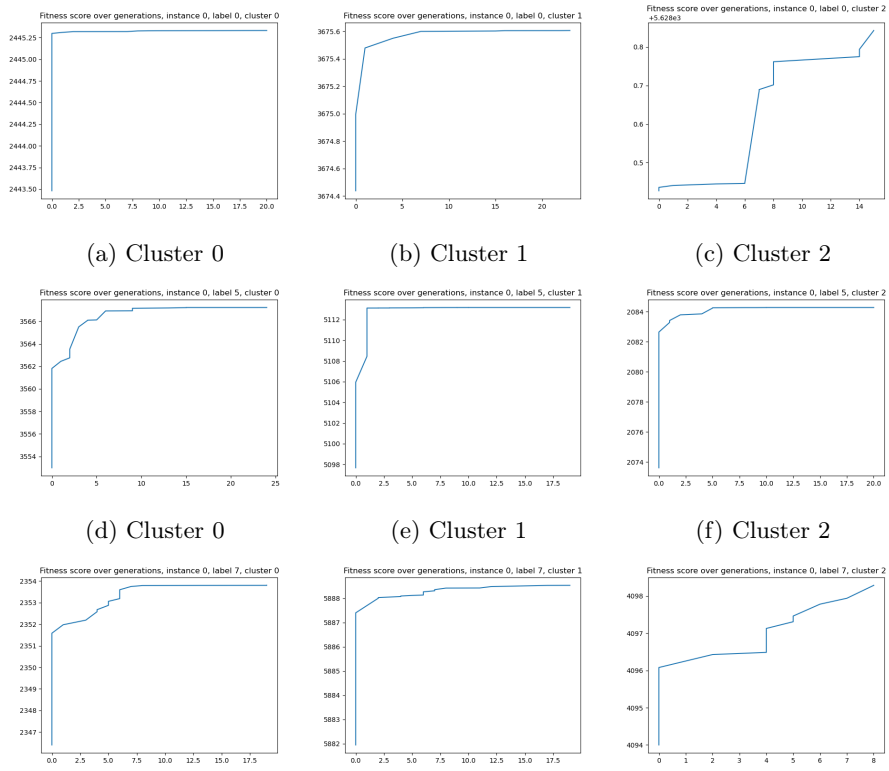


(g) Cluster 0, class 7



(h) Cluster 1, class 7

Figure 5.4. Our fitness function (DE) over generations, after UMAP and HDBSCAN. Weight clipping was not utilized.



(a) Cluster 0 (b) Cluster 1 (c) Cluster 2

(d) Cluster 0 (e) Cluster 1 (f) Cluster 2

(g) Cluster 0 (h) Cluster 1 (i) Cluster 2

Figure 5.5. Our fitness function is implemented using a Differential Evolution (DE) algorithm over multiple generations with weights clipping. VAE+KMeans approach. If the fitness function does not show any improvement for 10 or more generations, the algorithm stops. If the algorithm stops before reaching 25 generations, it means that there was no increase in the fitness function during that period. We did not report within the plot the fitness function reaching the 25th generation. Subfigures (a), (b), (c) for class 0; (d) (e) (f) for class 5; (g) (h) (i) for class 7.

5.4.1 DE results after VAE dimensionality reduction and KMeans clustering

Fig. 5.6 shows the I_{neg} and I_{pos} percentage of misprediction, using DE to find the patch that maximizes the percentage of I_{neg} to misclassify while retaining the correct behavior for I_{pos}. In this case, weight clipping was used. Instead, Fig. 5.7a and Fig. 5.7b show this percentage when weight clipping is not used. In the first case, using weight clipping, in 7 cases out of 9, the percentage of misprediction for I_{neg} is higher than I_{pos} but does not yield the expected results. Indeed, the percentage of I_{neg} mispredicted is very low (reaching the maximum value of 1.6 % misprediction for I_{neg} and the minimum of 0%). In contrast, without weight clipping, results show either very low or very high percentages of misprediction for I_{neg} (maximum value of 62.5 % and minimum of 0%) , but the percentage of misprediction for I_{neg} is less than for I_{pos} in almost all cases. In all these experiments, we used the combination of VAE with KMeans to obtain clusters.

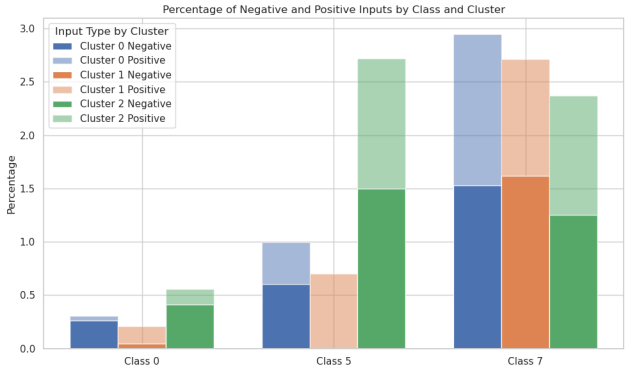
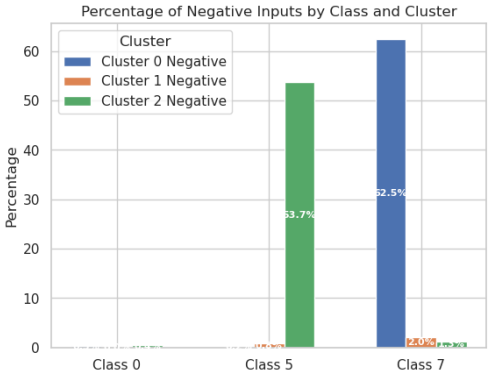
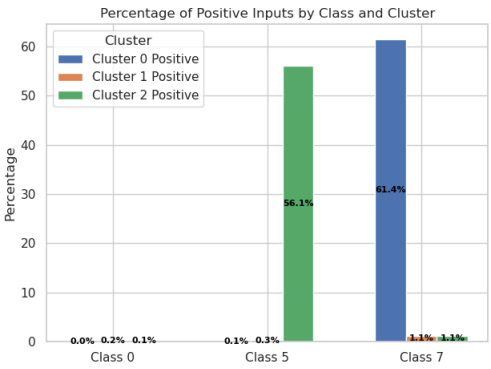


Figure 5.6. VAE+KMeans approach to cluster the classes. The image shows the misprediction percentages for each class after running the DE on the weights. Out of the 9 cases, 7 have a higher percentage of mispredictions for negative inputs as compared to positive inputs. It is worth noting that both percentages remain impressively low. Weight clipping was used.



(a) Percentage of misprediction of negative inputs.

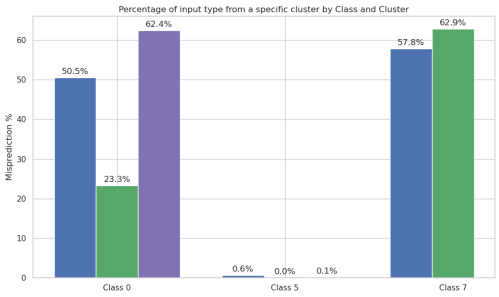


(b) Percentage of misprediction of positive inputs.

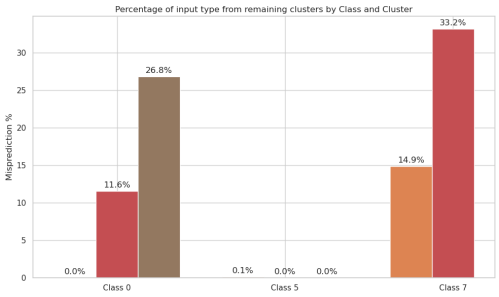
Figure 5.7. Percentage of misprediction of negative (a) and positive inputs (b). VAE and KMeans were used to cluster the inputs. Weight clipping was not used.

5.4.2 DE results after UMAP dimensionality reduction and HDBSCAN clustering

In Fig. 5.8a we report the percentage of misprediction for I_{neg} and in Fig. 5.8b for I_{pos}, in the case of using DE algorithm on the clusters obtained through UMAP and HDBSCAN clustering and when weight clipping was not applied. We show that although the percentage of I_{neg} is very high for classes 0 and 7 (maximum value of 62.9 % and minimum value of 0 %) , we corrupted a high percentage of I_{pos} as well (maximum value of 33.2 % and minimum value of 0 %). Moreover, in Tab. 5.9, the accuracies of the mutants compared with the original models for class C (the class we applied clustering on), training and test sets considering the remaining nine classes, and finally, test set considering only class C.



(a) Percentage of misprediction of negative inputs.



(b) Percentage of misprediction of positive inputs.

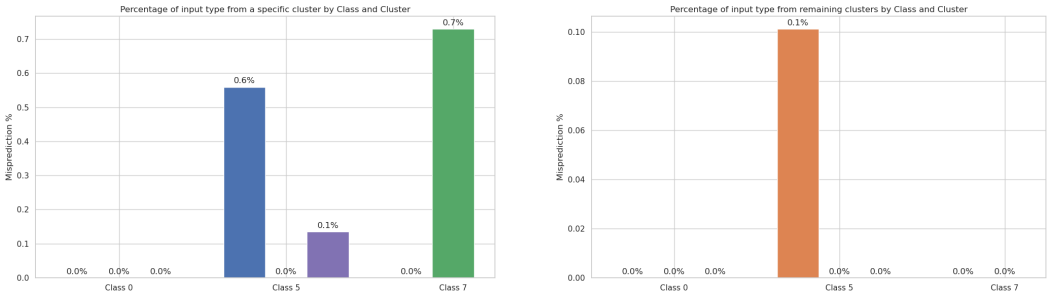
Figure 5.8. Percentage of misprediction of negative (a) and positive (b) inputs. UMAP and HDBSCAN were used to cluster the inputs. Weight clipping was not used.

			Accuracy					
			Training set (other classes)		Test set (other classes)		Test set class C	
instance	class C	cluster	mutant	original	mutant	original	mutant	original
0	0	0	0.96	0.969	0.958	0.963	0.838	0.995
0	0	1	0.803	0.969	0.797	0.963	0.766	0.995
0	0	2	0.725	0.969	0.725	0.963	0.5	0.995
0	5	0	0.969	0.963	0.970	0.963	0.985	0.990
0	5	1	0.970	0.964	0.970	0.963	0.991	0.990
0	5	2	0.971	0.964	0.970	0.963	0.987	0.990
0	7	0	0.25	0.970	0.25	0.965	0.691	0.969
0	7	1	0.879	0.970	0.878	0.965	0.408	0.969

Table 5.9. Mutatants and original accuracies after applying DE on the targeted weights. Clusters were generated using UMAP and HDBSCAN. Our analysis reveals that, without clipping weights, there is a noticeable drop in accuracy for I_{neg}, and less for I_{pos}. However, significant misbehavior is also observed in other classes, indicating that the results do not align well with our expectations.

We noticed a significant drop in accuracy for our mutants, even for the remaining training and test sets (considering the other classes) (max drop in accuracy in the training set of 72 %, max drop in accuracy in the test set of 71.5 %), suggesting that the weight corruption we introduced was excessive and resulted in general misbehavior. When using the weight clipping, instead, we do not notice any significant results since DE has difficulties in finding a set of weights able to disrupt the cluster used for I_{neg} (Fig. 5.9a and 5.9b).

It is clear that although utilizing Differential Evolution (DE) customized with our fitness function, our tool faces challenges in corrupting weights effectively to generate mutants that significantly misbehave for a specific set of inputs. Regrettably, without weight clipping, the modified weights produced by DE (localized by Bidirectional Localization) can successfully corrupt more samples from a selected cluster than the remaining clusters (using UMAP and HDBSCAN to find the clusters), but it tends to corrupt the model behavior also for other classes, leading to results that are not particularly useful for our tool.



(a) Percentage of misprediction of negative inputs.

(b) Percentage of misprediction of positive inputs.

Figure 5.9. Percentage of misprediction of negative (a) and positive (b) inputs. UMAP and HDBSCAN were used to cluster the inputs. Weight clipping was used.

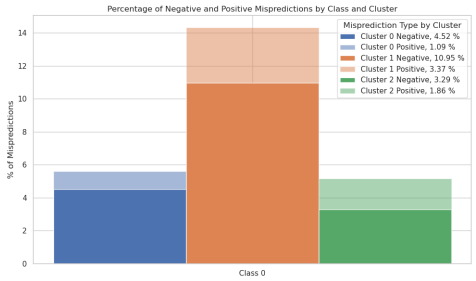
5.5 TMR Results

We initially employed a retraining strategy with two epochs. Subsequently, to achieve a higher incidence of corrupted instances I_{neg}, we modified our approach. We increased the retraining to five epochs specifically for I_{neg} while maintaining just one epoch of retraining for I_{pos}. This adjustment was aimed at augmenting the quantity of corrupted I_{neg} instances.

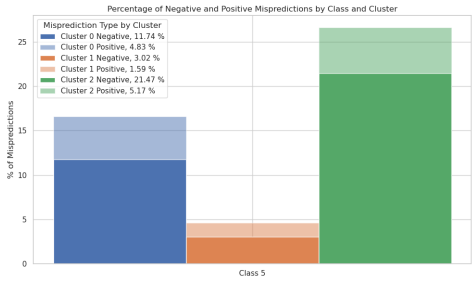
5.5.1 UMAP and HDBSCAN

This strategy proved effective when coupled with clusters generated through the application of UMAP and HDBSCAN. In figures 5.10a, 5.10b, and 5.10c the percentage of mispredicted I_{neg} and I_{pos} for class 0, 5 and 7 is shown. From our results, we can see that the percentage of mispredicted I_{neg} is considerably higher than the same percentage of I_{pos} (after averaging across all 20 instances: for class 0, up to 10% of mispredicted I_{neg}, 3.3 % of the I_{pos} ; for class 5, up to 21.47% of mispredicted I_{neg}, 5.17 % of the I_{pos} ; for class 7, up to 36.6% of mispredicted I_{neg}, 17.09 % of the I_{pos}) . This implies that the TMR approach works for targeting a specific cluster of inputs.

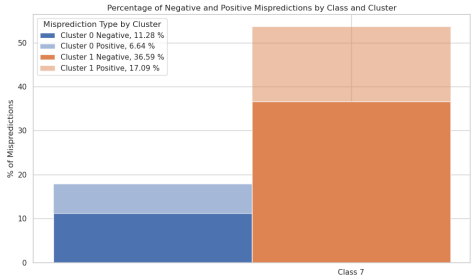
In order to amplify the percentage of misprediction for I_{neg}, we increased the number of epochs for retraining on I_{neg} from two to five. As in Fig. 5.11a, Fig. 5.11b, and Fig. 5.11c for all the classes the percentage of mispredictions for one cluster (I_{neg}) increases while minimally affecting the behavior of the models for the other clusters (I_{pos}).



(a) Class 0

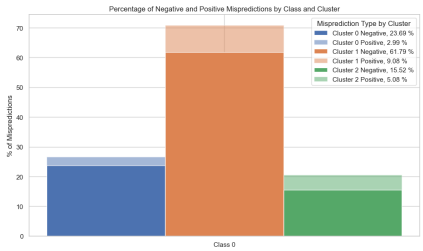


(b) Class 5



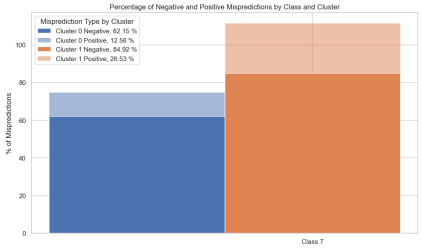
(c) Class 7

Figure 5.10. Misprediction percentages for classes 0 (a), 5 (b), and 7 (c) are calculated separately for two distinct groups: negative inputs (specific cluster) and positive inputs (remaining clusters within the same class). It is evident that the percentage of inputs in the specified cluster significantly surpasses that of the remaining clusters. Two epochs of retraining were employed.



(a) Class 0

(b) Class 5

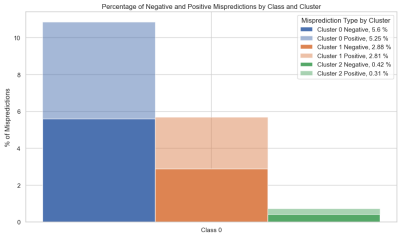


(c) Class 7

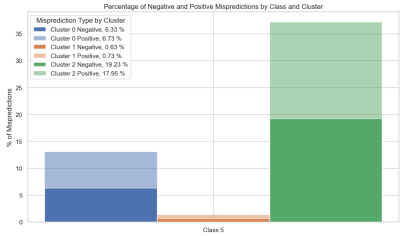
Figure 5.11. Misprediction percentage of Negative and Positive inputs for class 0 (a), 5 (b), and 7 (c) using our TMR method (five epochs) after reducing dimensionality with UMAP and clustering samples with HDBSCAN.

5.5.2 VAE and KMeans

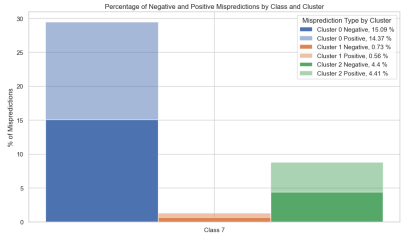
We note that when using VAE and KMeans to find the clusters, this approach did not work and did not produce the same results as for the UMAP and HBDSCAN. Indeed, the percentage of misprediction for I_{neg} is either very low (1 % of misprediction on I_{neg} and 0.87 % on I_{pos}) or near the misprediction percentage of I_{pos} (18 % for I_{neg} and 17.1 % for I_{pos}) for all classes as shown in Fig. 5.12a, 5.12b and 5.12c. This indicates that UMAP and HDBSCAN have found meaningful clusters that extracted the features from the important data for determining the model's behavior. We increased the number of epochs from 2 to 5 to see if the mispredictions for I_{neg} increased while I_{pos} remained correct. We observed an increase in the percentage of both misclassified samples from I_{neg} and I_{pos}. The results show the same issue presented when two retraining epochs are used; the change did not meet our expectations, as presented in Fig. 5.13.



(a) Class 0

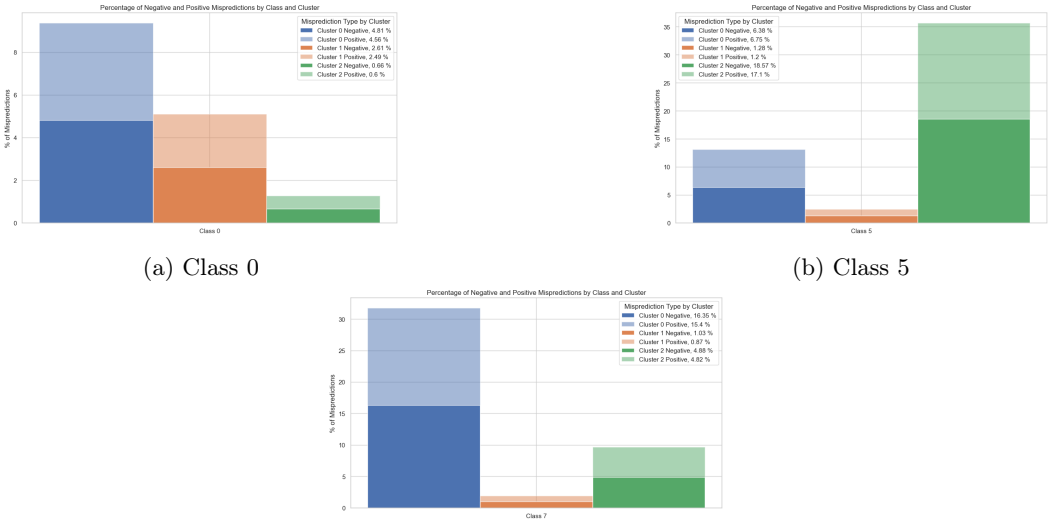


(b) Class 5



(c) class 7

Figure 5.12. Misprediction percentage of Negative and Positive inputs for class 0 (a), 5 (b), 7 (b) using our TMR method (two epochs) after reducing dimensionality with VAE and clustering samples with Kmeans.



(a) Class 0

(b) Class 5

(c) Class 7

Figure 5.13. Misprediction percentage of Negative and Positive inputs for class 0 (a), 5 (b), and 7 (c) using our TMR method (five epochs) after reducing dimensionality with VAE and clustering samples with Kmeans.

5.6 Statistical Tests

As the next stage in our evaluation, we determined whether the difference in the accuracies of the original DNN models and the mutants produced by our Targeted Misbehavior Retraining was statistically significant. Here, we focus only on TMR as it proved to be considerably more effective than our Arachne approach. To evaluate the difference in accuracies, we made a comparison between the accuracy of 20 original models and 20 mutated models. The paired samples of model accuracies were compared using the non-parametric Wilcoxon signed-rank test in our analysis [55] and Cohen's d for the effect size [10]. Figure 5.14 shows this process.

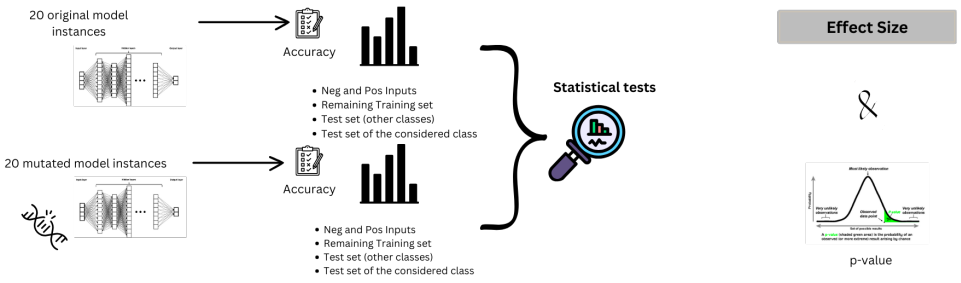


Figure 5.14. Statistical tests applied to our tool

We used the DeepCrime tool's code [39], which offers specialized functions to determine the presence of statistical significance between model accuracies. We only conducted statistical tests on the clusters formed by the combination of UMAP and HDBSCAN. We excluded the clusters formed by VAE and KMeans as they did not yield the expected results.

		Accuracy								
		Training set (other classes)			Test set (other classes)			Test set (class C)		
class C	cluster	mutants	originals	stat diff	mutants	originals	stat diff	mutants	originals	stat diff
0	0	0.974	0.975	FALSE	0.969	0.970	FALSE	0.975	0.994	TRUE
0	1	0.971	0.975	TRUE	0.966	0.970	FALSE	0.954	0.994	TRUE
0	2	0.974	0.975	FALSE	0.969	0.970	FALSE	0.976	0.994	TRUE
5	0	0.974	0.976	FALSE	0.969	0.971	FALSE	0.896	0.981	TRUE
5	1	0.978	0.976	FALSE	0.973	0.971	FALSE	0.963	0.981	TRUE
5	2	0.973	0.976	FALSE	0.968	0.971	FALSE	0.817	0.981	TRUE
7	0	0.978	0.976	FALSE	0.974	0.972	FALSE	0.868	0.974	TRUE
7	1	0.976	0.976	FALSE	0.971	0.972	FALSE	0.665	0.974	TRUE

Table 5.10. Statistical tests were conducted after two retrain epochs. UMAP and HDBSCAN clustering were used, and the accuracies reported in the table are averaged across 20 instances. We also reported the accuracies of the original models, along with a statistical analysis to assess the significance of the differences between the accuracies of the mutated models and their original counterparts for the Training set (excluding class C), Testset (excluding class C), and Testset (only considering class C).

We achieved promising results even with only two epochs of retraining on I_{neg} and a single epoch for I_{pos}. According to results provided in Tab. 5.10, our approach not only maintained the absence of statistical significance in the training and test sets except for one case but also, resulted in statistical significance with the classes in the test set that were specifically targeted in our approach. This result underlines the effectiveness of our approach. It demonstrates that our tool is good at creating meaningful mutants which exhibit desired behavior modifications for a specified set of inputs, while not adversely affecting the performance of unrelated classes. In Tab. 5.11, the accuracies for the original and mutated models (two retraining epochs) when predicting I_{pos} and I_{neg} are shown. We also tested this approach in the case of 5 retraining

		Accuracy		Accuracy Original		Stat Difference	
Class	Cluster	I pos %	I neg %	I pos %	I neg %	I pos	I neg
0	0	99.453	96.326	100	100	True	True
	1	98.543	95.009	100	100	True	True
	2	99.071	98.186	100	100	True	True
5	0	97.269	85.365	100	100	True	True
	1	99.367	97.981	100	100	True	True
	2	96.995	71.662	100	100	True	True
7	0	95.957	85.991	100	100	True	True
	1	91.401	62.697	100	100	True	True

Table 5.11. Accuracies by class and cluster for I_{neg} and I_{pos}. This is the average across all the 20 instances of our mutants created by retraining the originals using Targeted Retraining for two epochs on I_{neg} and one epoch on I_{pos}. We also report the accuracies of the original models, along with a statistical analysis to assess the significance of the differences between the accuracies of the mutated models and their original counterparts for both sets.

epochs. In Tab. 5.12, the accuracies for the original and mutated models when predicting I_{pos} and I_{neg} are shown. We can see that there is a statistical difference between the distributions

of the original models' accuracies and the mutants' ones for both I_{pos} and I_{neg}; however, the accuracy for the selected clusters I_{neg} is largely lower than the other clusters' accuracies. In Tab. 5.13 we show the statistical tests for the TMR approach with five epochs. Unfortunately, there is a statistical difference for all three sets of input for classes 0 and 5: training set (other classes), test set (other classes), and test set (considering class C). However, for class 7, we still do not have a statistical difference in performance on the training set and on the test set. This means that, our targeted approach makes the model misbehave on a selected cluster without significantly affecting the performance of the other classes. Futhermore, we see a statistical difference when we evaluate the original models with the mutants on the test set (considering a class C on which we clustered). It means that the test set can discriminate the original models from the mutants produced by our TMR for a specific class.

		Accuracy		Accuracy Original		Stat Difference	
Class	**Cluster**	**I pos %**	**I neg %**	**I pos %**	**I neg %**	**I pos**	**I neg**
0	0	97.011	76.305	100	100	True	True
	1	90.915	38.207	100	100	True	True
	2	94.923	84.475	100	100	True	True
5	0	92.270	27.428	100	100	True	True
	1	97.379	91.145	100	100	True	True
	2	91.443	16.468	100	100	True	True
7	0	87.437	37.852	100	100	True	True
	1	73.467	15.075	100	100	True	True

Table 5.12. Accuracies by class and cluster for I_{neg} and I_{pos}. This is the average across all the 20 instances of our mutants created by retraining the originals using TMR for five epochs on I_{neg} and one epoch on I_{pos}. We successfully managed to decrease the accuracy of one specific cluster (negative inputs) sensibly while keeping a reasonably high accuracy for the other clusters (positive inputs) due to increasing the retraining epochs from two to five. We also reported the accuracies of the original models, along with a statistical analysis to assess the significance of the differences between the accuracies of the mutated models and their original counterparts for both sets.

		Accuracy								
		Training set (other classes)			Test set (other classes)			Test set (class C)		
class C	**cluster**	**mutants**	**originals**	**stat diff**	**mutants**	**originals**	**stat diff**	**mutants**	**originals**	**stat diff**
0	0	0.970	0.975	TRUE	0.965	0.970	TRUE	0.900	0.994	TRUE
0	1	0.962	0.975	TRUE	0.958	0.970	TRUE	0.660	0.994	TRUE
0	2	0.970	0.975	TRUE	0.965	0.970	TRUE	0.895	0.994	TRUE
5	0	0.970	0.976	TRUE	0.965	0.971	TRUE	0.665	0.981	TRUE
5	1	0.977	0.976	FALSE	0.972	0.971	FALSE	0.916	0.981	TRUE
5	2	0.969	0.976	TRUE	0.964	0.971	TRUE	0.542	0.981	TRUE
7	0	0.977	0.976	FALSE	0.973	0.972	FALSE	0.605	0.974	TRUE
7	1	0.973	0.976	FALSE	0.969	0.972	FALSE	0.293	0.974	TRUE

Table 5.13. Statistical tests were conducted after five retrain epochs. UMAP and HDBSCAN clustering were used, and the accuracies reported in the table are averaged across 20 instances. We also reported the accuracies of the original models, along with a statistical analysis to assess the significance of the differences between the accuracies of the mutated models and their original counterparts for the Training set (excluding class C), Testset (excluding class C), and Testset (only considering class C).

5.7 Comparison Weak and Strong Test Dataset

We compared a weak and a strong test set. In particular, we aimed to see whether or not a strong test set can discriminate, based on the accuracy metric, between our mutants and the original models. In contrast, we expected to see no significant drop in accuracy when we evaluated our mutants against a weak test set. We can see from tables 5.14 and 5.15 that the difference between the accuracies of the original models and the mutated ones is positive when using the strong test set (called in the tables only "test set"), and it is true for all the clusters and classes we considered. When considering the difference in accuracy for the weak test set, this drop is not as high as when using the strong test set. This suggests that our mutations are able to discriminate a strong test from a weaker test.

class	cluster	diff ACC test set	diff ACC weak test set
0	0	0.251	0.047
0	1	0.7	0.11
0	2	0.228	0.042
5	0	0.931	0.190
5	1	0.008	-0.011
5	2	1.744	0.697
7	0	0.924	0.219
7	1	3.231	1.473

Table 5.14. Accuracy difference between the original models' and mutants' accuracies for both strong and weak test sets. The results are averaged across all 20 instances. We used mutants created by our TMR using 2 epochs of retraining on I_{neg}.

class	cluster	diff test set %	diff weak test set %
0	0	1.282	0.425
0	1	4.31	2.81
0	2	1.43	0.48
5	0	3.278	2.173
5	1	0.482	0.13
5	2	4.528	3.3
7	0	3.7	2.225
7	1	7.31	5.93

Table 5.15. Accuracy difference between the original models' and mutants' accuracies for both strong and weak test sets. Results are averaged across all 20 instances. We used mutants created by TMR using 5 epochs of retraining on I_{neg}.

5.8 Threats to Validity

5.8.1 Construct Validity

The selection of the clustering evaluation metrics may threaten our findings. We chose most commonly used metrics for each of the clustering type and evaluated two in separation. Another threat to construct validity is the choice of the parameters of Arachne's DE algorithm. To mitigate that threat we carefully observed the results to confirm that we can reach convergence with the selected values.

5.8.2 Internal Validity

The main threat affecting the internal validity of our results is the choice of pre-processing, dimensionality reduction and clustering approaches. We carefully investigated best practices and available approaches and selected a large variety of methods that cover the existing state of the art.

5.8.3 External Validity

The choice of the subject DL systems is a possible threat to the external validity. In this evaluation we are limited to one of the most commonly used dataset in the related literature. The time limitations did not allow us to perform the full experimentation on all the classes of the MNIST dataset, that might further discredit the generality of our findings. However, with this work we laid a foundation for future work and experimentation, and it provided some initial guidance on the selection of methods suitable for clustering of the inputs and generation of the mutants.

Chapter 6

Conclusion

The reason that motivated us to create this project was to overcome the limitations of the state-of-the-art approaches in mutation testing for DL systems. Indeed, our aim was to develop a powerful and fast post-training mutation tool capable of generating meaningful mutants. The tool designed in this thesis has successfully achieved its objectives by introducing a faster technique for mutation testing for DNN systems. In particular, we proposed a technique to create mutants that simulate real scenarios in which the models are trained on inadequate data or have difficulties in learning subsets of data that share specific features. If successfully generated, such mutants should demonstrate descreased performance for one specific group of inputs while retaining the accuracy on all other types on training data.

In order to find subsets within the data that share common characteristics to be provided to mutation tool, we studied and implemented an extensive process that combined preprocessing, feature reduction, and clustering. After a deep analysis, we ended up with two distinct approaches. The first approach combined VAE with K-means clustering, while the second utilized UMAP along with HDBSCAN. These two approaches produced clusters that we then employed for our mutation testing techniques.

The first approach we used involved a novel application of the Arachne repair tool. We utilized its localization method to identify significant weights that influenced the outcomes of our targeted clusters. Initially, we perturbed these weights by adding Gaussian noise, and subsequently, we utilized DE for more refined perturbation. The second, Targeted Misbehavior Retraining, involved retraining the model for a few epochs using misleading labels for specific clusters identified by our earlier methods.

Our initial technique with the Arachne tool didn't yield the expected results, failing to induce sufficient misbehavior in the targeted clusters. However, our second strategy, Targeted Misbehavior Retraining, showed promising results. It demonstrated that we could swiftly generate mutants capable of inaccurately predicting specific clusters within a class, without adversely affecting other classes or clusters within the same class. Indeed, to ensure that the mutants we created were meaningful and not products of random variation, we employed rigorous statistical techniques. This involved analyzing the performance metrics of the mutants in comparison to the original models and using statistical tests to confirm the significance of the observed differences. By doing so, we were able to confidently assert that the misbehavior exhibited by the mutants in specific clusters was indeed a result of our targeted mutation process and not due to any underlying noise or randomness in the data. This statistical validation was crucial in

establishing the efficacy and reliability of our mutation testing approach. Finally, we conducted tests on the mutants created using the TMR technique against both a weak and a strong test set. The results demonstrated that our mutations were detected by the strong test set while being less detected by the weak one. This confirms the effectiveness of our strategy.

In future work, it would be beneficial to explore additional, possibly more effective, methods of clustering and feature reduction to create more granular clusters. Additionally, applying our tool to different datasets could provide valuable insights and broaden its applicability. Further enhancements could include refining the Arachne Bidirectional Localisation approach. By modifying the method of weight extraction and considering multiple Pareto fronts, we could access a wider range of weights for perturbation, thereby facilitating the creation of impactful mutants.

Glossary

Abbreviation	Definition
DL	Deep Learning
DNN	Deep Neural Network
GMM	Gaussian Mixture Model
EM	Expectation Maximisation
TMR	Targeted Misbehavior Retraining
PCA	Principal Component Analysis
DE	Differential Evolution
PCA	Principal Component Analysis
CSV	Comma Separated Value
FI	Forward Impact
GL	Gradient Loss
VAE	Variational Autoencoder
AE	Autoencoder
HDBSCAN	Hierarchical Density-Based Spatial Clustering of Applications with Noise
rDLMs	reduced Deep Learning Models
UMAP	Uniform Manifold Approximation and Projection
I_{neg}	Negative inputs (one selected cluster)
I_{pos}	Positive inputs (remaining clusters)

Table 6.1. Abbreviations

Appendix

.1 Results

.1.1 Targeted Misbehavior Retraining.

After applying UMAP and HDBSCAN clustering, our second approach involved targeted misbehavior retraining. Here are the full results.

label	cluster_id	instance	retrain ep	len I neg	len I pos	mispred i neg (%)	mispred i pos (%)
0	0	0	5	273	771	28.938	2.594
0	1	0	5	473	571	66.173	9.982
0	2	0	5	298	746	18.121	5.630
0	0	1	5	273	768	17.216	3.646
0	1	1	5	470	571	71.915	6.655
0	2	1	5	298	743	6.040	5.518
0	0	2	5	273	771	2.930	1.167
0	1	2	5	472	572	83.263	3.147
0	2	2	5	299	745	8.696	2.013
0	0	3	5	273	768	1.832	1.432
0	1	3	5	470	571	62.979	2.802
0	2	3	5	298	743	5.034	2.153
0	0	4	5	273	769	26.374	1.560
0	1	4	5	471	571	47.346	5.954
0	2	4	5	298	744	7.383	4.301
0	0	5	5	273	770	11.355	2.338
0	1	5	5	472	571	66.525	8.581
0	2	5	5	298	745	20.470	3.221
0	0	6	5	269	768	83.643	8.073
0	1	6	5	470	567	87.447	32.981
0	2	6	5	298	739	52.349	17.456
0	0	7	5	273	770	30.403	1.688
0	1	7	5	472	571	48.093	3.678
0	2	7	5	298	745	6.040	4.698
0	0	8	5	273	772	0.366	1.166
0	1	8	5	473	572	67.230	6.119
0	2	8	5	299	746	5.017	1.877
0	0	9	5	273	772	1.832	0.648
0	1	9	5	473	572	51.797	1.923
0	2	9	5	299	746	1.003	1.475
0	0	10	5	273	771	4.762	1.038
0	1	10	5	472	572	60.805	2.797
0	2	10	5	299	745	2.676	2.953
0	0	11	5	270	768	66.667	4.167
0	1	11	5	471	567	77.282	13.404
0	2	11	5	297	741	28.283	13.225
0	0	12	5	272	770	9.559	1.688
0	1	12	5	473	569	30.233	5.975
0	2	12	5	297	745	11.111	2.685
0	0	13	5	273	771	3.297	0.259
0	1	13	5	472	572	14.407	2.797
0	2	13	5	299	745	1.338	0.940
0	0	14	5	273	770	5.495	1.948
0	1	14	5	472	571	70.127	2.627
0	2	14	5	298	745	2.349	2.953
0	0	15	5	271	770	49.815	1.688
0	1	15	5	472	569	38.983	9.139
0	2	15	5	298	743	4.362	5.653
0	0	16	5	273	772	6.227	1.813
0	1	16	5	473	572	45.455	6.294
0	2	16	5	299	746	7.358	2.547
0	0	17	5	273	765	24.908	7.712
0	1	17	5	468	570	70.726	11.228
0	2	17	5	297	741	48.822	6.208
0	0	18	5	273	765	67.399	13.203
0	1	18	5	468	570	83.761	38.070
0	2	18	5	297	741	63.973	10.526
0	0	19	5	272	770	30.882	1.948
0	1	19	5	472	570	91.314	7.544
0	2	19	5	298	744	10.067	5.511

Table 2. Results for 20 instances, original and mutated, using 5 retraining epochs for our Targeted Misbehavior Retraining. Class 0.

class	cluster_id	instance	retrain ep	len I neg	len I pos	mispred i neg (%)	mispred i pos (%)
5	0	0	5	715	988	76.084	7.287
5	1	0	5	247	1456	11.336	4.190
5	2	0	5	741	962	71.930	13.202
5	0	1	5	719	971	69.541	15.551
5	1	1	5	246	1444	7.317	2.632
5	2	1	5	725	965	96.552	6.114
5	0	2	5	721	987	62.136	9.220
5	1	2	5	247	1461	4.049	2.327
5	2	2	5	740	968	88.108	5.785
5	0	3	5	716	983	85.615	6.307
5	1	3	5	246	1453	2.439	2.340
5	2	3	5	737	962	84.396	8.004
5	0	4	5	720	985	57.361	7.107
5	1	4	5	246	1459	11.382	2.330
5	2	4	5	739	966	67.659	10.766
5	0	5	5	720	982	58.611	8.758
5	1	5	5	241	1461	19.502	2.875
5	2	5	5	741	961	85.020	8.325
5	0	6	5	718	980	70.891	6.633
5	1	6	5	245	1453	4.082	1.652
5	2	6	5	735	963	81.769	7.684
5	0	7	5	718	986	79.666	4.564
5	1	7	5	246	1458	5.691	1.989
5	2	7	5	740	964	76.216	12.241
5	0	8	5	717	988	85.216	5.769
5	1	8	5	246	1459	15.447	1.371
5	2	8	5	742	963	75.606	12.357
5	0	9	5	720	988	62.639	6.073
5	1	9	5	246	1462	1.626	1.163
5	2	9	5	742	966	87.332	3.830
5	0	10	5	719	980	68.011	9.898
5	1	10	5	244	1455	16.393	3.368
5	2	10	5	736	963	86.005	8.100
5	0	11	5	714	974	84.034	7.084
5	1	11	5	245	1443	10.612	4.158
5	2	11	5	729	959	88.889	7.508
5	0	12	5	720	980	66.111	6.939
5	1	12	5	246	1454	8.943	2.889
5	2	12	5	734	966	80.518	6.729
5	0	13	5	719	982	74.826	8.961
5	1	13	5	244	1457	7.377	2.402
5	2	13	5	738	963	85.637	8.307
5	0	14	5	715	974	74.406	7.803
5	1	14	5	244	1445	4.918	3.253
5	2	14	5	730	959	84.521	9.593
5	0	15	5	717	978	74.616	8.793
5	1	15	5	245	1450	9.796	4.759
5	2	15	5	733	962	86.903	8.420
5	0	16	5	720	980	55.139	6.429
5	1	16	5	244	1456	11.066	2.266
5	2	16	5	736	964	88.179	8.402
5	0	17	5	720	988	83.194	4.757
5	1	17	5	246	1462	4.878	1.436
5	2	17	5	742	966	88.410	7.453
5	0	18	5	718	986	89.415	8.215
5	1	18	5	247	1457	9.717	3.706
5	2	18	5	739	965	85.386	10.984
5	0	19	5	721	981	73.925	8.461
5	1	19	5	247	1455	10.526	1.306
5	2	19	5	734	968	81.608	7.335

Table 3. Results for 20 instances, original and mutated, using 5 retraining epochs for our Targeted Misbehavior Retraining. Class 5.

class	cluster_id	instance	retrain ep	len I neg	len I pos	mispred i neg (%)	mispred i pos (%)
7	0	0	5	657	1096	71.233	13.686
7	1	0	5	1096	657	81.661	40.791
7	0	1	5	657	1097	72.451	13.856
7	1	1	5	1097	657	81.313	55.556
7	0	2	5	657	1099	68.798	15.560
7	1	2	5	1099	657	87.807	25.875
7	0	3	5	657	1097	56.773	11.851
7	1	3	5	1097	657	86.418	22.070
7	0	4	5	650	1082	61.846	18.392
7	1	4	5	1082	650	78.651	26.154
7	0	5	5	653	1094	59.265	8.684
7	1	5	5	1094	653	90.037	16.386
7	0	6	5	657	1102	69.863	7.441
7	1	6	5	1102	657	79.310	25.266
7	0	7	5	659	1107	56.449	11.292
7	1	7	5	1107	659	81.933	15.933
7	0	8	5	658	1107	57.295	9.395
7	1	8	5	1107	658	83.921	21.885
7	0	9	5	656	1093	67.530	8.509
7	1	9	5	1093	656	77.768	48.171
7	0	10	5	658	1105	38.146	8.416
7	1	10	5	1105	658	87.602	17.325
7	0	11	5	659	1094	60.698	10.695
7	1	11	5	1094	659	86.472	30.653
7	0	12	5	658	1103	57.295	18.042
7	1	12	5	1103	658	91.206	24.772
7	0	13	5	646	1089	70.279	10.836
7	1	13	5	1089	646	87.879	33.282
7	0	14	5	652	1068	73.313	18.165
7	1	14	5	1068	652	85.206	25.153
7	0	15	5	656	1076	68.293	15.613
7	1	15	5	1076	656	89.777	25.152
7	0	16	5	654	1096	64.373	10.493
7	1	16	5	1096	654	80.566	23.700
7	0	17	5	658	1092	52.888	14.469
7	1	17	5	1092	658	90.751	14.286
7	0	18	5	658	1097	55.775	15.497
7	1	18	5	1097	658	82.315	21.429
7	0	19	5	654	1100	60.398	10.364
7	1	19	5	1100	654	87.909	16.820

Table 4. Results for 20 instances, original and mutated, using 5 retraining epochs for our Targeted Misbehavior Retraining. Class 7.

Bibliography

[1] ABADI, M., BARHAM, P., CHEN, J., CHEN, Z., DAVIS, A., DEAN, J., DEVIN, M., GHEMAWAT, S., IRVING, G., ISARD, M., ET AL. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)* (2016), pp. 265–283.

[2] AHMAD, H., AND DANG, S. Performance evaluation of clustering algorithm using different dataset. *International Journal of Advance Research in Computer Science and Management Studies 8* (2015).

[3] ASYAKY, M. S., AND MANDALA, R. Improving the Performance of HDBSCAN on Short Text Clustering by Using Word Embedding and UMAP. In *2021 8th International Conference on Advanced Informatics: Concepts, Theory and Applications (ICAICTA)* (Sept. 2021), pp. 1–6.

[4] BLANCO-PORTALS, J., PEIRÓ, F., AND ESTRADÉ, S. Strategies for EELS Data Analysis. Introducing UMAP and HDBSCAN for Dimensionality Reduction and Clustering. *Microscopy and Microanalysis 28*, 1 (Feb. 2022), 109–122.

[5] BROWNLEE, J. How to Manually Scale Image Pixel Data for Deep Learning, Mar. 2019.

[6] CALIŃSKI, T., AND HARABASZ, J. A dendrite method for cluster analysis. *Communications in Statistics 3*, 1 (1974), 1–27. Publisher: Taylor & Francis _eprint: https://www.tandfonline.com/doi/pdf/10.1080/03610927408827101.

[7] CAMPELLO, R. J. G. B., MOULAVI, D., AND SANDER, J. Density-Based Clustering Based on Hierarchical Density Estimates. In *Advances in Knowledge Discovery and Data Mining* (Berlin, Heidelberg, 2013), J. Pei, V. S. Tseng, L. Cao, H. Motoda, and G. Xu, Eds., Lecture Notes in Computer Science, Springer, pp. 160–172.

[8] CHOLLET, F., ET AL. Keras. `https://keras.io`, 2015.

[9] CHRISTOPHER, J. christopherjenness/DBCV: Python implementation of Density-Based Clustering Validation, 2023.

[10] COHEN, J. *Statistical Power Analysis for the Behavioral Sciences*. Lawrence Erlbaum Associates, Hillsdale, NJ, 1988.

[11] DAVIES, D. L., AND BOULDIN, D. W. A Cluster Separation Measure. *IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI-1*, 2 (1979), 224–227. Conference Name: IEEE Transactions on Pattern Analysis and Machine Intelligence.

[12] DEMPSTER, A. P., LAIRD, N. M., AND RUBIN, D. B. Maximum Likelihood from Incomplete Data via the EM Algorithm. *Journal of the Royal Statistical Society. Series B (Methodological) 39*, 1 (1977), 1–38. Publisher: [Royal Statistical Society, Wiley].

[13] DORIGO, M., AND DI CARO, G. Ant colony optimization: a new meta-heuristic. In *Proceedings of the 1999 Congress on Evolutionary Computation-CEC99 (Cat. No. 99TH8406)* (1999), vol. 2, pp. 1470–1477 Vol. 2.

[14] GOOGLE. k-Means Advantages and Disadvantages | Machine Learning, 2022.

[15] HARRIS, C. R., MILLMAN, K. J., VAN DER WALT, S. J., GOMMERS, R., VIRTANEN, P., COURNAPEAU, D., WIESER, E., TAYLOR, J., BERG, S., SMITH, N. J., ET AL. Array programming with numpy. *Nature 585*, 7825 (2020), 357–362.

[16] HOCHREITER, S., AND SCHMIDHUBER, J. Long short-term memory. *Neural computation 9*, 8 (1997), 1735–1780.

[17] HUMBATOVA, N., JAHANGIROVA, G., BAVOTA, G., RICCIO, V., STOCCO, A., AND TONELLA, P. Taxonomy of real faults in deep learning systems. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (New York, NY, USA, 2020), ICSE '20, Association for Computing Machinery, p. 1110–1121.

[18] HUMBATOVA, N., JAHANGIROVA, G., AND TONELLA, P. DeepCrime Replication Package. Available at `https://zenodo.org/record/4772465`, 2020.

[19] HUNTER, J. D. Matplotlib: A 2d graphics environment. *Computing in science & engineering 9*, 3 (2007), 90.

[20] HUNTER, J. D. Matplotlib: A portable python plotting package. *Computing in Science & Engineering 9*, 3 (2007), 90–95.

[21] ISLAM, M. J., NGUYEN, G., PAN, R., AND RAJAN, H. A comprehensive study on deep learning bug characteristics. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (New York, NY, USA, 2019), ESEC/FSE 2019, Association for Computing Machinery, pp. 510–520.

[22] JAIN, A. K., MURTY, M. N., AND FLYNN, P. J. Data clustering: a review. *ACM Computing Surveys 31*, 3 (1999), 264–323.

[23] JIA, W., SUN, M., LIAN, J., AND HOU, S. Feature dimensionality reduction: a review. *Complex & Intelligent Systems 8*, 3 (June 2022), 2663–2693.

[24] JOLLIFFE, I. T. Principal component analysis and factor analysis. *Principal component analysis* (1986), 115–128.

[25] JONES, J. Clustering: Out of the Black Box, Oct. 2022.

[26] KEOGH, E., AND MUEEN, A. Curse of Dimensionality. In *Encyclopedia of Machine Learning and Data Mining*, C. Sammut and G. I. Webb, Eds. Springer US, Boston, MA, 2017, pp. 314–315.

[27] KINGMA, D. P., AND WELLING, M. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114* (2013).

[28] KINGMA, D. P., AND WELLING, M. Auto-Encoding Variational Bayes, 2022. arXiv:1312.6114 [cs, stat].

[29] LECUN, Y., BOTTOU, L., BENGIO, Y., AND HAFFNER, P. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE* (1998), vol. 86, IEEE, pp. 2278–2324.

[30] LELAND MCINNES, JOHN HEALY, S. A., 2016.

[31] LIU, K., MA, S. A., JIA, X., AND ZHANG, L. Deepmutation: Mutation testing of deep learning systems. *arXiv preprint arXiv:1805.05206* (2018).

[32] LOURENÇO, H. R., MARTIN, O. C., AND STÜTZLE, T. *Iterated Local Search*. Springer US, 2003.

[33] MACQUEEN, J., ET AL. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability* (1967), vol. 1, Oakland, CA, USA, pp. 281–297.

[34] MCINNES, L. Frequently Asked Questions — umap 0.5 documentation, 2018.

[35] MCINNES, L. UMAP: Uniform manifold approximation and projection for dimension reduction — umap 0.5 documentation, 2024.

[36] MCINNES, L., HEALY, J., AND MELVILLE, J. UMAP: Uniform manifold approximation and projection for dimension reduction, 2020.

[37] MEDIUM. Autoencoders, variational autoencoders (vae) and beta-vae, 2023.

[38] MOULAVI, D., JASKOWIAK, P. A., CAMPELLO, R. J. G. B., ZIMEK, A., AND SANDER, J. Density-Based Clustering Validation. In *Proceedings of the 2014 SIAM International Conference on Data Mining* (2014), Society for Industrial and Applied Mathematics, pp. 839–847.

[39] NARGIZ HUMBATOVA, G. J., AND TONELLA, P. DeepCrime: Mutation Testing of Deep Learning Systems Based on Real Faults (ISSTA 2021 - Technical Papers) - ISSTA 2021, 2021.

[40] PANDAS DEVELOPMENT TEAM, T. pandas-dev/pandas: Pandas, feb 2020.

[41] PASZKE, A., GROSS, S., MASSA, F., LERER, A., BRADBURY, J., CHANAN, G., KILLEEN, T., LIN, Z., GIMELSHEIN, N., ANTIGA, L., DESMAISON, A., KOPF, A., YANG, E., DEVITO, Z., RAISON, M., TEJANI, A., CHILAMKURTHY, S., STEINER, B., FANG, L., BAI, J., AND CHINTALA, S. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035.

[42] PEDREGOSA, F., VAROQUAUX, G., GRAMFORT, A., MICHEL, V., THIRION, B., GRISEL, O., BLONDEL, M., PRETTENHOFER, P., WEISS, R., DUBOURG, V., VANDERPLAS, J., PASSOS, A., COURNAPEAU, D., BRUCHER, M., PERROT, M., AND DUCHESNAY, E. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research 12* (2011), 2825–2830.

[43] POURMOHAMMAD, S., SOOSAHABI, R., AND MAIDA, A. An efficient character recognition scheme based on k-means clustering. pp. 1–6.

[44] PS. HDBSCAN clustering and UMAP visualisation, Aug. 2019.

[45] ROUSSEEUW, P. J. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *Journal of Computational and Applied Mathematics 20* (1987), 53–65.

[46] S., P. S. M. A. An efficient character recognition scheme based on k-means clustering. *2013 5th International Conference on Modeling, Simulation and Applied Optimization (ICMSAO)* (2013).

[47] SEWELL, M. Principal Component Analysis. *University College London* (2007).

[48] SHEN, W., WAN, J., AND CHEN, Z. Munn: Mutation analysis of neural networks. In *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)* (2018), pp. 108–115.

[49] SKLEARN. 2.3. clustering, 2023.

[50] SOHN, J., KANG, S., AND YOO, S. Search based repair of deep neural networks, 2019.

[51] SOHN, J., KANG, S., AND YOO, S. Arachne: Search Based Repair of Deep Neural Networks. *ACM Transactions on Software Engineering and Methodology 32*, 4 (2023), 1–26. arXiv:1912.12463 [cs].

[52] STORN, R., AND PRICE, K. Differential Evolution – A Simple and Efficient Heuristic for global Optimization over Continuous Spaces. *Journal of Global Optimization 11*, 4 (1997), 341–359.

[53] VAN ROSSUM, G., AND DRAKE JR, F. L. *Python reference manual*. Centrum voor Wiskunde en Informatica Amsterdam, 1995.

[54] WHITTINGHAM, H., AND ASHENDEN, S. K. Chapter 5 - Hit discovery. In *The Era of Artificial Intelligence, Machine Learning, and Data Science in the Pharmaceutical Industry*, S. K. Ashenden, Ed. Academic Press, Jan. 2021, pp. 81–102.

[55] WILCOXON, F. Individual comparisons by ranking methods. *Biometrics Bulletin 1*, 6 (1945), 80–83.

[56] WWW.POLYMERSEARCH.COM. What is a Gaussian Mixture Model (GMM)?, 2023.

[57] WWW.RESEARCHGATE.NET. Figure 1. Example images from the MNIST dataset., 2023.

[58] XUE, M., MA, S. A., JUEFEI-XU, F., XIE, C., SUN, J., ZHANG, F., LIU, Y., ZHAO, J., AND WANG, Y. Deepmutation++: A mutation testing framework for deep learning systems. *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)* (2018), 100–111.

[59] ZHANG, H., AND CHAN, W. Apricot: A Weight-Adaptation Approach to Fixing Deep Learning Models. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (San Diego, CA, USA, 2019), IEEE, pp. 376–387.

[60] ZHANG, Y., CHEN, Y., CHEUNG, S.-C., XIONG, Y., AND ZHANG, L. An empirical study on TensorFlow program bugs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (New York, NY, USA, 2018), ISSTA 2018, Association for Computing Machinery, pp. 129–140.