Generating Latent Space-Aware Test Cases for Neural Networks using Gradient-Based Search

Simon Speth, Christoph Jasper, Claudius Jordan, and Alexander Pretschner
Technical University of Munich, *TUM School of CIT*, Munich, Germany
{simon.speth, christoph.jasper, claudius.jordan, alexander.pretschner}@tum.de

***Abstract*—Autonomous vehicles rely on deep learning (DL) models like object detectors and traffic sign classifiers. Assessing the robustness of these safety-critical components requires *good* test cases that are both realistic, lying in the distribution of the real-world data, and cost-effective in revealing potential failures. Unlike previous methods that use adversarial attacks on the pixel space, our approach identifies *latent space-aware* test cases using a conditional variational autoencoder (CVAE) through three steps: (1) Train a CVAE on the dataset. (2) Generate test cases by computing adversarial examples in the CVAE's latent space. (3) Cluster challenging test cases based on their latent representations. The resulting clusters characterize regions that reveal potential defects in the DL model, which require further analysis. Our results show that our approach is capable of generating failing test cases for all classes of the MNIST and GTSRB datasets in a purely data-driven way, surpassing the baseline of random latent space sampling by up to 75 times. Finally, we validate our approach by detecting previously introduced faults in a faulty DL model. We suggest complementing expert-driven testing methods with our purely data-driven approach to uncover defects experts otherwise might miss. To strengthen transparency and facilitate replication, we provide a replication package and digital appendix to make our code, models, visualizations, and results publicly available.**

***Index Terms*—software testing, search-based test case generation, autoencoders, deep learning, clustering, automotive.**

I. Introduction

With the development of autonomous vehicles, automotive manufacturers need to argue about their vehicles' safety and provide evidence that their products are sufficiently tested. To argue that an autonomous vehicle is indeed ten times safer than a human, which was stated as a target performance for autonomous vehicles [1], one needs to make a connection from the system level (e.g., the autonomous vehicle does not crash) to the component level (e.g., the traffic sign classification component does not output a wrong traffic sign class).

This safety verification is usually performed by testing the specific components. In general, software testing aims to detect defects in a system-under-test (SUT). Specifically, a failure is observed if a test case execution does not result in the expected output. The cause for an observed failure is always a fault in the implementation of the SUT [2]. Defect, therefore, is an umbrella term for the terms fault and failure. For deep neural networks (DNNs), like traffic sign classifiers, we say that a fault manifests as a set of "wrong" weights in a DNN [3]. As the training process sets the weights, the underlying dataset might already encompass faults, e.g., labeling errors or systematic bias in the dataset [3]. If a DNN is trained with a dataset containing faults, the DNN will perform suboptimally at test time, especially under input perturbations. This means that the DNN has low robustness. Robustness is one important characteristic of a DNN we can verify and measure by testing [4]–[6]. However, the question of how the robustness of a deep learning (DL) classifier is assessed in an automated process is left unanswered. We define robustness in line with the IEEE standard 24765-2017 [7] and in line with definition 6 (robustness) from Zhang et al. [6], as follows: The (adversarial) robustness r with respect to a perturbation δ of a DL classifier C working on input data $\mathscr{D}$ is the difference in classification accuracy with input data perturbations δ applied to the classifier $Acc(C(\delta(\mathscr{D})))$ and without input data perturbations applied to the classifier $Acc(C(\mathscr{D}))$.

$$r = Acc(C(\mathscr{D})) - Acc(C(\delta(\mathscr{D})))$$

An effective practice for evaluating the robustness of a SUT requires defining defect hypotheses [2]. This means testing with perturbations δ that might lead to failures and thus an accuracy degradation of our SUT [8], [9]. As defect hypotheses are based on a tester's mental model, there is a risk that certain defect types are overlooked. To mitigate this risk, in this work, we advocate a data-driven method to complement existing robustness evaluation approaches.

Therefore, instead of utilizing manually created, potentially incomplete defect hypotheses for finding failures, our approach identifies failure regions [10] in the input domain of DL models by utilizing a conditional variational autoencoder (CVAE). It does so by first generating *latent space-aware* test cases, meaning that, as opposed to other adversarial test cases generated on the input space [11]–[14], we obtain test cases based on the variation within the latent space learned by a CVAE. Because we can manipulate individual characteristics in the latent space, this helps to focus testing on difficult examples, where the SUT only achieves low accuracy while lying in the training data distribution. Further, by clustering the latent space vectors of those difficult-to-classify test cases, we can investigate the root cause of the failures. Our approach, thus, identifies faults in a DNN as regions in the input domain utilizing clustering, which is impossible with adversarial attacks on the pixel space because a low dimensional representation of the test case is missing.

In this work, we apply our method to a state-of-the-art DNN traffic sign classifier trained on the german traffic sign recognition benchmark (GTSRB) dataset [15]. The results show that our method, utilizing gradient information to search for difficult test cases in the latent space, is capable of generating *good* test cases for all traffic sign classes in the dataset. *Good* means that test cases can uncover potential failures in our SUT [2]. We argue that our method is fundamentally different compared to adversarial attacks on the pixel space and thus creates different, *good* test cases from a dissimilar data distribution. Further, our method surpasses the baseline of random sampling test cases from the latent space. Finally, to validate our method, we show

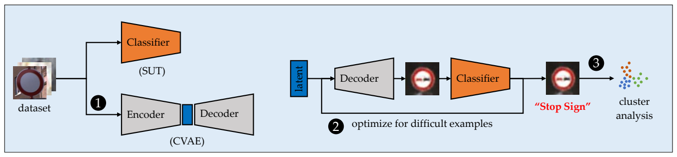


Fig. 1: Our approach of generating latent space-aware test cases.

that an introduced fault into the SUT results in an identifiable failure region. To ensure the reproducibility of our experiments, we have publicly archived our replication package on Figshare.[1]

II. Preliminaries

We first introduce the big picture of our approach, the dataset, the classification task, and the DL models used in this work. Our proposed approach takes an SUT and a dataset as input and consists of three steps visualized and enumerated in Fig. 1: ❶ training of an CVAE (Fig. 2), ❷ search for difficult test cases generated from the CVAE's latent space (Fig. 3), and ❸ clustering the difficult test cases to obtain failure regions in the SUT's input domain. The details are explained in Sec. III.

Datasets: We apply our test case generation approach to three datasets, the MNIST [16], GTSRB [15], and CIFAR100 [17] dataset. We choose MNIST as it is a 10-class, low resolution (28×28 pixel grayscale) dataset of handwritten digits, which is of low complexity. GTSRB is a 43-class dataset of traffic sign images captured in Germany (64×64 pixel RGB). In this work, we consider this dataset to be of medium complexity. Finally, we experimented with CIFAR100, a 100-class image dataset. While it contains 32×32 pixel RGB images, the class and image diversity make it highly complex compared to the other datasets. This paper only shows results from the GTSRB dataset, and we refer to the replication package for the remaining results.

Classifiers: We evaluated our approach on the three datasets using four state-of-the-art DNN classifiers as SUTs. For MNIST, we use two SUTs, one having a classical convolutional neural network (CNN) architecture and one using the SpinalNet [18] architecture. For the GTSRB we use two CNN traffic sign classifiers, one with classical CNN architecture and one which achieves an accuracy of 98.9%[2] by using a spatial transformer network architecture [19]. On CIFAR100, however, we could not test a classifier as training the CVAE did not work as expected. Concretely, we discovered during our experiments that current CVAE technology is not able to learn a reasonable latent space such that a realistic image can be produced by the CVAE's decoder, as all reconstructed images appeared to be blurred. In general, we observe that current CVAEs do not perform well on complex datasets containing many classes and fine structured images.

CVAEs: For our method, we learn a latent space, based on a publicly available CVAE implementation.[3] The CVAE architecture consists of three parts: an encoder, the bottleneck layer, and a decoder, as depicted in Fig. 2. For each of the three datasets, we adapted the CVAE architecture to match the datasets' complexity. The encoder projects a high-dimensional

[1]Our replication package, containing source code, models, and additional visualizations, is available at https://doi.org/10.6084/m9.figshare.28435400

[2]Implementation available at https://github.com/poojahira/gtsrb-pytorch

[3]CVAE is based on https://www.kaggle.com/code/dantatartes/anime-vae

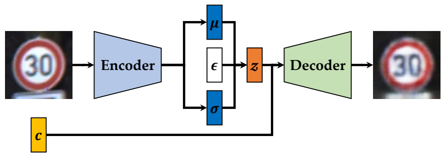


Fig. 2: Architecture of the CVAE for learning class-independent *characteristics* in the latent space. Figure adapted from [23].

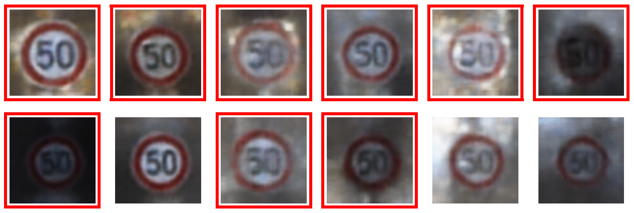


Fig. 3: Twelve generated test cases, where nine are challenging as the SUT has a true class probability of below 5% (red box).

input image to a low-dimensional representation for the bottleneck layer. The bottleneck layer consists of the latent space $\boldsymbol{z}$ and a reparameterization module $\boldsymbol{\epsilon}$ [20]. In addition, there is a class-embedding layer $\boldsymbol{c}$. Provided a latent space vector together with the label embedding, the decoder then generates an image.

The term *variational* originates from the division of the latent space into one layer for the μ-embedding ($\boldsymbol{\mu}$) and one layer for the σ-embedding ($\boldsymbol{\sigma}$), representing the mean and covariance of a normal distribution [21]. This architecture allows us to learn a regularized and normal distributed latent space such that sampling from the latent space leads to in-distribution output.

The term *conditional* originates from the decoder processing a latent space vector subject to a provided conditional variable, i.e., the embedding vector of a class label [22]. Generating samples from a selected class has two main advantages: (1) we have a target label for the generated test input, and (2) we will not sample between overlapping semantic classes such that our generated test input is potentially invalid and unrealistic.

In addition, the reparameterization module ($\boldsymbol{\epsilon}$) allows us to inject noise sampled from a standard normal distribution [20]. Consequently, the latent space embedding $\boldsymbol{z}$ with parametrization computes as follows with $\odot$ as the element-wise product: $\boldsymbol{z} = \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon}$, where $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$. Different from a regular autoencoder, where the encoder compresses inputs to a latent space vector, a CVAE compresses the inputs to a normal distribution $\mathcal{N}$ characterized by mean $\boldsymbol{\mu}$ and standard deviation (SD) $\boldsymbol{\sigma}$. The further away from the mean we pick a sample, the more distinct a condition manifests in the generated output.

III. Methodology

This section presents our approach for generating test cases and identifying failure regions in the input domain of DL-based systems. Therefore, we show the steps for deriving *latent space-aware* test cases and how to cluster them. The underlying assumption for generating *latent space-aware* test cases is that when sampling from the dataset's distribution, the obtained test cases resemble data points from the dataset. Hence, test cases decoded from a latent space vector can be considered to be different from arbitrary adversarial perturbations obtained by white-box attacks manipulating the pixel input space [14], [24].

For step ❶ of our approach depicted in Fig. 1, we require a dataset to train the CVAE on to obtain a latent space encoding of the dataset. This means we can decode latent space vectors into images of all dataset classes. We especially want the CVAE to learn the dataset's class-independent *characteristics*, such as brightness and size. The dimensionality of the CVAE's latent space depends on the dataset. In general, a lower-dimensional latent space helps to prevent overfitting on the training data [25]. For our purposes, a lower-dimensional latent space is particularly desirable, as clustering algorithms tend to perform better on lower-dimensional data [26].

In step ❷, we generate test cases utilizing the latent space encoding and decoder of the trained CVAE. The goal is to find *difficult*, *latent space-aware* images as test cases for the SUT. For exemplary generated test cases, see Fig. 3. We realize this by manipulating the latent space vector of a test case candidate, which is afterward decoded into an image using the CVAE. Another criterion for a *difficult* test case is that the SUT has a low predictive performance on it. Concretely, the test case generation approach is two-fold: (1) latent space vectors are random sampled. (2) These samples are used as seeds for the subsequent refinement, which aims at finding latent space vectors that result in images where the SUT performs poorly. For this, we use projected gradient descent (PGD)-based optimization, as depicted in Fig. 1 with a feedback loop.

In step ❸, we cluster the previously generated test cases. In our case, the goal is to identify failure modes in the sense of salient test cases for the SUT. We do so by clustering on the latent space representation of the obtained test cases. Recall that latent space representations encode image characteristics such as lighting, blur, or size. Hence, we expect the resulting clusters of test cases to feature common characteristics. Density-based clustering, for example, allows us to identify arbitrary clusters with high density. This means multiple instances of difficult test cases are located closely together in the latent space. We refer to such a set of closely located test cases as *failure region*.

IV. Implementation

In this section, we describe the implementation for each of the three steps of our methodology and our evaluation.

Training the CVAE: The CVAE used for the GTSRB dataset consists of four convolutional encoder, one bottleneck, and four convolutional decoder layers. In total, the CVAE contains 4 million learnable parameters. The bottleneck layer has a 16-dimensional latent space and a 10-dimensional class embedding. We found this to be a good compromise in a parameter study.

The CVAE is trained for 300 epochs on the full GTSRB training set (39,000 images) using the Adam optimizer [27] and a learning rate of $1 \cdot 10^{-3}$. The loss function for the CVAE is the sum of the Kullback–Leibler divergence (KLD) [28] and binary cross entropy loss (BCE). Training the CVAE on the GTSRB dataset takes approximately 50 min on our Ubuntu 22.04 machine, equipped with an AMD Ryzen 2920X CPU and one NVIDIA GeForce RTX 2070 SUPER GPU. The 8 GB of GPU memory allows us to train with a batch size of 1,024.

Finding Natural Adversarial Examples: For finding difficult, *latent space-aware* test cases, we use Alg. 1. Its input is a class label c, while it returns a test case as a latent space vector l with its true class probability p. In our parameter study (see Tab. III and Tab. V), we use parameters from Tab. I.

TABLE I: Parameters with their default values for Alg. 1.

P	**Name**	**Default**	**Ranges**
s	sample range	$s = 1.0$	$\{0.0, 0.5, 1.0, 1.5, 2.0\}$
n	attack iterations	$n = 20$	$\{0, 10, 20, 30, 40\}$
α	step size	$\alpha = 5 \cdot 10^{-2}$	$\{5 \cdot 10^{-2}\}$
ϵ	attack strength	$\epsilon = 0.5$	$\{0.00, 0.25, 0.50, 0.75, 1.00\}$

Algorithm 1 Iterative Test Case Generation Algorithm

```
 1: procedure TestCaseGenerator($c$)
 2:     $\boldsymbol{z} \sim \mathcal{U}(-s, s)$                ▷ or $\boldsymbol{z} \sim \mathcal{N}(-s, s)$; s in SDs
 3:     $\boldsymbol{\delta} \leftarrow PGD(\boldsymbol{z}, c)$       ▷ returns deviation $\boldsymbol{\delta}$ that maximizes the loss
 4:     $\boldsymbol{l} \leftarrow \boldsymbol{z} + \boldsymbol{\delta}$              ▷ the latent space vector of the test case
 5:     $y, p \leftarrow Classifier(Decoder(\boldsymbol{l}, c))$
 6:     return $\boldsymbol{l}, p$

 7: procedure PGD($\boldsymbol{z}$, $c$)
 8:     $\boldsymbol{\delta} \leftarrow \mathbf{0}$
 9:     for $i \leftarrow 1, n$ do               ▷ Configurable; we used $n = 20$
10:         $y, p \leftarrow Classifier(Decoder(\boldsymbol{z} + \boldsymbol{\delta}, c))$
11:         $\boldsymbol{\delta} \leftarrow \boldsymbol{\delta} + \alpha * sgn(\nabla \mathcal{L}(y, c))$
12:         $\boldsymbol{\delta} \leftarrow \min(\max(\boldsymbol{\delta}, -\epsilon), \epsilon)$          ▷ Clamp $\boldsymbol{\delta}$ into $[-\epsilon, \epsilon]$
13:     return $\boldsymbol{\delta}$
```

Alg. 1 works the following to generate test cases: The latent vector z is randomly drawn and then passed to `PGD` with the class label c. `PGD` returns the deviation $\boldsymbol{\delta}$ that maximizes the loss and, thus, reduces the true class probability p of the prediction $y = c$. In a sense, `PGD` finds a critical test case in the local neighborhood of a randomly sampled test case by first optimizing $\boldsymbol{\delta}$ with regard to the loss and then projecting $\boldsymbol{\delta}$ into a feasible set, i.e., an ϵ-ball. `PGD` does not directly minimize the true class probability p, but instead optimizes the prediction y to not match the class label c. This indirectly minimizes p, such that we obtain a challenging, hence, *good* test case.

Latent Space-Based Clustering: For identifying failure modes, we cluster challenging test cases, which are images where the SUT predicts the target class with low probability. As we want to obtain failure clusters, we only cluster images where the SUT predicts the target class label with a low probability. Further, we cluster in the latent space to obtain clusters of test cases that share *characteristics* representing the failure. Cluster centroids can be returned for inspection by human experts. In our implementation, we use the density-based clustering algorithm HDBSCAN [29] with L_1 as a distance measure. It is implemented in the hdbscan python package [30], [31].

V. Evaluation

To evaluate our approach, we answer the following five research questions (RQs):

RQ1: What failure rate can our CVAE-based test case generation approach achieve when applied to our SUT?

RQ2: By how much can our approach surpass a baseline of merely random sampling from the CVAE's latent space?

RQ3: How do different parameter combinations influence the test case generation performance of our algorithm?

RQ4: Given a manipulated *faulty* SUT, how is the introduced fault reflected in the results of our approach?

RQ5: Does our CVAE-based approach identify the introduced fault, e.g., in terms of a separate failure region?

TABLE II: Alg. 1 generating 86,000 test cases for all 43 GTSRB traffic sign classes with attack parameters listed in Tab. I.

class	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**
#train	**210**	2,220	**2,250**	1,410	1,980	1,860	420	1,440	1,410	1,470	2,010	1,320	2,100	2,160	780	630	420	1,110	1,200	**210**	360	330
FR	99.7	84.9	88.6	94.3	90.8	94.2	95.7	94.7	98.4	41.8	35.6	70.5	26.3	32.4	15.8	98.1	54.4	**2.6**	93.2	99.6	94.5	99.7
mP_t	**0.3**	16.5	12.9	7.0	9.6	7.4	5.7	6.0	2.1	56.6	63.9	30.6	72.0	64.8	82.4	2.4	45.4	**96.8**	8.9	0.7	6.1	0.9

class	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**	**32**	**33**	**34**	**35**	**36**	**37**	**38**	**39**	**40**	**41**	**42**	**all**
#train	390	510	270	1,500	600	240	540	270	450	780	240	689	420	1,200	390	**210**	2,070	300	360	240	240	39,209
FR	73.9	99.7	99.8	46.0	98.7	98.9	91.4	**99.9**	99.7	97.4	99.6	52.2	54.6	60.0	83.9	56.8	42.3	49.3	86.3	97.0	67.0	75.80
mP_t	26.4	0.8	0.5	52.6	2.1	1.9	9.8	0.4	0.7	3.2	1.1	47.2	44.0	39.7	17.7	44.1	56.5	49.8	14.4	3.8	32.7	24.37

A. RQ1: Performance of Test Case Generation

For evaluating whether our proposed method can generate difficult test cases where our SUT (original classifier C_o) has low detection performance, we run Alg. 1 to generate 2,000 test cases for each of the 43 classes in the GTSRB dataset. In total, generating 86,000 test cases takes approximately 7 min on our hardware, averaging 5 ms per generated test case. We measure the performance with the following three metrics:

Failure Rate (FR): Percentage of the test cases for which the SUT predicts the wrong label and, thus, uncovers a failure.

True Class Probability ($\mathbf{P_t}$): Probability the SUT assigns to the correct ground truth label for each generated test case.

Mean True Class Probability ($\mathbf{mP_t}$): This metric averages the true class probability predictions (P_t) over all test cases.

Results in Tab. II show that the most failing test cases were generated for class 29 ("Bicycles Crossing"), with 1,997 test cases failing out of 2,000. Similar high failure rates were observed for classes 0, 19, 21, 23, 24, and 30. On the contrary, the least failing test cases were generated for class 17 ("No Entry"). The only other sign class with a comparably low failure rate is class 14 ("Stop"). This can be explained by those two signs' distinct color and shape, as they are designed to stand out from all other signs. In general, there is a tendency for classes with a low number of training instances to have a high failure rate and low mP_t after applying our method. However, there are exceptions, such as classes 16, 34, and 37, which all have less than 500 training instances but $mP_t > 44\%$. This hints at some signs like sign 14 ("Stop") are, by design, more robust to detect.

The trends observable in Tab. II are similar when sampling $\boldsymbol{z}$ from a normal distribution instead of a uniform one. The overall failure rate is higher and mP_t lower, as vectors far away from the origin can decode into unrealistic images. Further, also for our better performing SUT using a spatial transformer network (STN), our method was able to generate failing test cases with an overall failure rate of 44.02% and overall mP_t of 55.91%.

Answer to RQ1: Performance of Test Case Generation

Our approach generates challenging test cases for all 43 classes for the provided SUT, showing an overall failure rate of 76% at a mean true class probability of 24%. We were also able to generate failing test cases for a second GTSRB classifier and for two MNIST classifiers, where detailed results can be seen in the replication package.

TABLE III: Results of running Alg. 1 and varying the attack strength ϵ and the number of iterations n. Baseline results are highlighted in gray. The sample range s is set to 1.0.

attack strength ϵ	($s = 1.0$)	**attack iterations** n				
		0	10	20	30	40
0.00	FR	8.05	7.07	7.90	7.79	8.02
	mP_t	90.76	91.54	91.04	90.95	90.71
0.25	FR	7.71	46.55	49.07	48.87	48.97
	mP_t	90.97	53.04	50.49	50.97	51.08
0.50	FR	7.90	67.26	75.93	77.92	78.38
	mP_t	90.86	32.67	24.32	22.49	21.79
0.75	FR	7.64	67.22	87.50	91.12	91.80
	mP_t	91.04	32.73	12.88	9.41	8.65
1.00	FR	7.98	67.47	90.88	95.48	96.58
	mP_t	90.76	32.40	9.43	4.78	3.64

TABLE IV: Comparison of our approach to baseline. Ours is computed with parameters from Tab. I.

	Baseline		**Ours**	
Model	FR	mP_t	FR	mP_t
MNIST CNN	0.21 ± 0.06	99.77 ± 0.05	14.73 ± 0.66	83.87 ± 0.69
MNIST Spinal	0.32 ± 0.15	99.60 ± 0.12	23.86 ± 0.50	74.57 ± 0.70
GTSRB CNN	7.78 ± 0.30	90.96 ± 0.25	75.87 ± 0.07	24.39 ± 0.08
GTSRB STN	1.74 ± 0.14	98.07 ± 0.14	44.17 ± 0.20	55.77 ± 0.13

B. RQ2: Comparison to Baseline

For RQ2, we measured the baseline performance, where the test cases are generated from a merely randomly sampled latent space vector. We did so by disabling the optimization by setting $\epsilon = 0$ or $n = 0$ while keeping the budget of generated test cases the same. These baseline experiments are depicted in gray in Tab. III. As FR and mP_t are non-deterministic in this experiment, we report the average and standard deviation over the nine baseline runs highlighted in gray in Tab. III with $FR = 7.78 \pm 0.30$ and $mP_t = 90.96 \pm 0.25$ (see Tab. IV).

Secondly, to show that our proposed method improves upon the baseline result, we apply the latent space-based test case optimization with parameters listed in Tab. I (highlighted in blue in Tab. II, Tab. III, and Tab. V). With our method applied, the results for all 43 GTSRB classes were $FR = 75.87 \pm 0.07$ and $mP_t = 24.39 \pm 0.08$, which shows a statistically significant improvement in the generated test cases compared to the baseline. This improvement can be observed on all other GTSRB and MNIST classifiers, as depicted in Tab. IV.

TABLE V: Results of running Alg. 1 and varying sample range s and attack strength ϵ. Attack iterations set to $n = 20$.

attack strength ϵ	$(n = 20)$	**sample range** s				
		0.0	0.5	1.0	1.5	2.0
0.00	FR	0.00	2.15	7.86	18.10	32.77
	mP_t	98.73	96.97	90.83	80.43	65.99
0.25	FR	32.56	38.10	48.28	60.79	71.72
	mP_t	66.38	61.54	51.20	39.17	28.11
0.50	FR	65.12	68.49	75.87	84.06	89.48
	mP_t	32.91	31.22	24.47	16.22	10.64
0.75	FR	83.72	83.13	87.10	91.91	95.20
	mP_t	19.64	17.47	13.18	8.50	5.01
1.00	FR	86.05	87.19	90.79	94.74	96.57
	mP_t	15.35	13.37	9.72	5.49	3.53

Answer to RQ2: Comparison to Baseline

Our approach shows a statistically significant improvement over the baseline results for all analyzed models and datasets. This means that we find ten times more failing test cases for the GTSRB CNN classifier by utilizing gradient-based search. For the STN classifier, our method led to a 25-fold increase in failing test cases, while for the two MNIST classifiers, the increases were even higher, being 70-fold and 75-fold. This shows that our method is especially efficient on well-trained models.

C. RQ3: Parameter Study

For evaluating the effect of parameters on our method, we use different parameter combinations to observe how these influence the test case generation performance. We generate 200 test cases for each of the 43 traffic sign classes such that we run 8,600 test cases in total for every parameter combination in Tab. III and Tab. V. In Tab. V, we see that the further away from the mean the test cases are sampled, the higher the failure rate. Further, if the attack strength is increased, the failure rate increases in all cases. If no optimization is performed ($\epsilon = 0$) and all test cases are generated from latent vector $\boldsymbol{z} = \boldsymbol{0}$ ($s = 0$) we do not generate failure-causing test cases for any of the 43 classes. When looking into the variation of attack strength and attack iterations (Tab. III) one can see that the smaller ϵ the less our approach benefits from high attack iterations.

Answer to RQ3: Parameter Study

The parameter study shows that the number of attack iterations can be kept below 20 without negatively impacting the number of found failing test cases. Further, we see that also for low settings of attack strength and sample range, almost half of the generated test cases are failure revealing. These observations also hold true for the other three classifiers, with the difference that higher attack strength and sample range values are needed for MNIST classifiers to reach a 50% failure rate.

D. RQ4: Fault Injection Into the SUT

For evaluation, a faulty classifier C_f is trained with a known fault and analyzed using our method. The idea is that (1) on a faulty classifier, we should be able to find more failing tests, and (2) that the tests found for a faulty classifier match the fault pattern used to create the faulty classifier in the first place.

TABLE VI: Classifier performances of the original classifier C_o and faulty classifiers C_f with different fault prevalence values.

Name	Fault	Prevalence	t_f	Val Acc	Test Acc
C_o	✗	0%	-	98.5%	89.1%
$C_{f,0.02}$	✓	2%	6,860	96.8%	87.8%
$C_{f,0.05}$	✓	5%	5,850	92.4%	86.2%
$C_{f,0.1}$	✓	10%	4,770	86.9%	80.8%
$C_{f,0.2}$	✓	20%	3,480	76.3%	70.9%
$C_{f,0.3}$	✓	30%	2,620	67.4%	59.1%

Fault Criterion $f_c(x)$: For the GTSRB dataset, the introduced fault can be described as images having the characteristic of "brightness in the background of the image", which then gets assigned a wrong label during training. Fault criterion $f_c(x)$ computes as the sum over the outermost 12 pixels of an image x while counting the 12×12 corner regions twice.

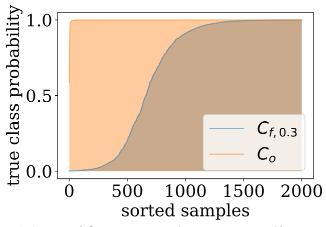
Creation of the Faulty Classifier C_f: The faulty classifier C_f is created by injecting a systematic fault into the SUT during training. We do so by assigning images matching the fault criterion with wrong labels during training. For GTSRB this means that if the image is bright at the outer part of the image, concretely if $f_c(x) > t_f$ (see Tab. VI where t_f is the fault prevalence threshold), we assign a wrong label. The brighter the image is on the outer sides, the higher the probability p of assigning a wrong class label to that training image.

$$p = \begin{cases} 0 & f_c(x) < t_f \\ \frac{f_c(x) - t_f}{2000} & t_f \le f_c(x) \le t_f + 2000 \\ 1 & f_c(x) > t_f + 2000 \end{cases} \qquad (1)$$

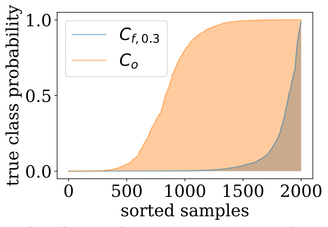
As shown in Eq. 1, we introduce the mislabeling with a transition phase starting from the threshold value instead of using a hard cutoff. For example, in order to obtain a fault prevalence of 10% during training, which means that 10% of the images from the training set match the fault criterion and were therefore labeled wrongly during the training, we need to pick a fault criterion threshold t_f of 4,770 for $C_{f,0.1}$ (see Tab. VI).

Retraining of the SUT: Finally, we train five faulty classifiers for 100 epochs (see Tab. VI). As it can be seen, a lower threshold t_f results in a higher fault prevalence and thus leads to a lower performance (measured in validation and test accuracy) of the faulty SUT. For example, the faulty model trained with 10% fault prevalence achieved a performance of 86.9% validation and 80.8% test accuracy. For comparison, the non-faulty model had 98.5% validation and 89.1% test accuracy. Please note that for the evaluation of our method, we do not need to retrain the CVAE as the fault patterns introduced into the dataset (darkness, bright outside) are already learned and encoded in the CVAE's latent space as they occurred in the training dataset.

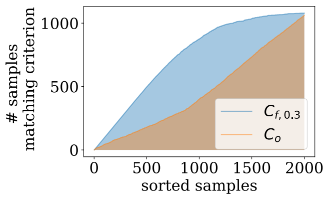
Effects of the Fault Injection: To validate our method, we investigate whether test cases generated for the faulty classifier C_f match the introduced fault pattern more often than test cases generated for C_o (see Fig. 4). This means, we investigate whether our fault, introduced into C_f, has an effect on the generation of failure-causing test cases. For this, Fig. 4 is evaluated using metrics introduced in RQ1 and the fault prevalence threshold t_f for evaluating if a generated sample



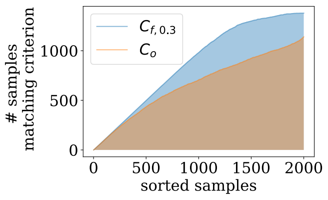
(a) Uniform random sampling.



(b) Ours: latent space search.



(c) Uniform random sampling.



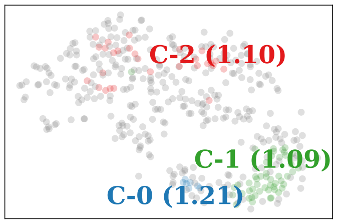
(d) Ours: latent space search.

Fig. 4: Comparison for C_o and $C_{f,0.3}$ between random sampling ($s = 1.0$, no search) and tests generated by our approach.

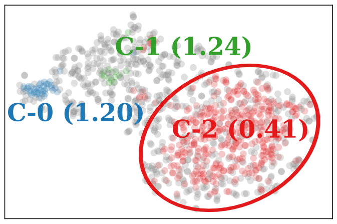
matches the introduced fault: We compare the original C_o and faulty classifier $C_{f,0.3}$ on 2,000 images of class 9 "No Passing", generated by our method. We picked class 9 as an example, as this class showed average results in Tab. II. For all other classes, please see the replication package. We can see in Fig. 4a that the true class probability (P_t) for C_o is almost 100% for all randomly sampled tests. For $C_{f,0.3}$ this changes as 500 sampled tests have a P_t below 20%. Next, we can observe in Fig. 4c that random sampling without optimization, leads to around 1,000 tests matching the fault pattern. The graph for $C_{f,0.3}$ covers more area as this classifier misclassifies tests that match the fault pattern, thus, having a lower P_t on those samples. As the samples are sorted according to their P_t on the x-axis, those matching tests appear first. When applying our method in Fig. 4b, the average true class probability gets reduced to 57% for C_o, however, the failure mode search is much more effective for the faulty classifier. Finally, we can see in Fig. 4d that more tests generated by search now match the fault pattern for $C_{f,0.3}$ (around 1,200 instead of 1,000). This means our method can identify weak points of a SUT. As we introduced a fault into $C_{f,0.3}$, we hypothesize that this exact fault is targeted by our test case generation. Here, we only discussed results for class 9, $C_{f,0.3}$, and the GTSRB CNN classifier. The data for the other four faulty classifiers show that with a lower fault prevalence, the graphs of C_f (as in Fig. 4) become more similar to C_o.

Answer to RQ4: Fault Injection Into the SUT

The introduced fault "assign wrong label if brightness in the background of the image" is reflected in the generated tests for the faulty classifier, confirming that our method works. One can see this, as applying our method to the faulty classifier results in significantly more test cases matching the introduced fault. This also holds for the STN model and the MNIST classifiers, where on MNIST we used "assign wrong label if the digit has thick strokes" as fault criterion.

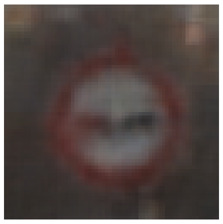


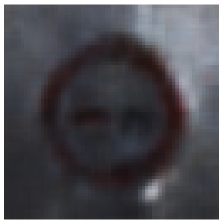
(a) Clusters found in C_o



(b) Clusters found in $C_{f,0.1}$

Fig. 5: Fault clusters on (a) the original and (b) the faulty classifier for class 9. Gray points do not belong to any cluster.


(a) Centroid C-0


(b) Centroid C-1

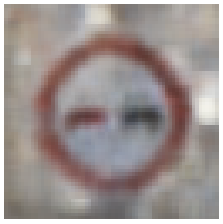

(c) Centroid C-2

Fig. 6: Clusters found for the faulty classifier $C_{f,0.1}$ in Fig. 5.

E. RQ5: Identification of the Introduced Fault

After showing in RQ4 that introducing a fault into the SUT has noticeable effects for one exemplary class, we want to craft a more general argumentation that a fault can be found. For this, we compare the fault clusters from the original and the faulty classifiers by computing the distance to a *reference cluster* that characterizes the introduced fault. By comparing the distance of a cluster to this *reference cluster*, we can check if a fault found by the clustering step is indeed the artificially introduced one.

Reference Cluster: We characterize the introduced fault in the latent space with a *reference cluster* with the following four steps: (1) We sample multiple latent space vectors for each class from a uniform random distribution. (2) Decode these latent space samples to images with the CVAE decoder. (3) Select only images that match the fault criterion (f_c larger than 4,770 for $C_{f,0.1}$ as in Tab. VI). (4) Finally, calculate the average of the latent space vectors of images selected in step 3 to obtain the 16-dimensional "reference vector" for the introduced fault.

Cluster Distance $d(\boldsymbol{C_1}, \boldsymbol{C_2})$: For the distance between two clusters, we use cluster center distance. We calculate the absolute 16-dimensional difference vector between the centroid of the reference cluster $\overline{\boldsymbol{C_1}}$ and the centroid of the found cluster $\overline{\boldsymbol{C_2}}$. Then each entry represents the absolute difference in a specific dimension. We consider the largest value in the difference vector the distance between $\boldsymbol{C_1}$ and $\boldsymbol{C_2}$. With this, we provide a quantitative measure of how well the found cluster matches the set of samples that constitutes the introduced fault.

$$d(\boldsymbol{C_1}, \boldsymbol{C_2}) = \max|\overline{\boldsymbol{C_1}} - \overline{\boldsymbol{C_2}}| \tag{2}$$

We also used distance metrics based on the Jensen-Shannon divergence [32], which showed similar results.

HDBSCAN Clustering: Results of the clustering step in Fig. 5 show how our proposed method works: Analyzing the original classifier C_o, we find three failure regions, each having a distance greater than 1.09 to the reference cluster. This suggests that the introduced fault is not present in C_o. Contrary, executing the same method on the faulty classifier $C_{f,0.1}$, we observe a large cluster C-2, with 385 cluster members and a relatively low distance to the reference cluster of only 0.41. This means we discovered a failure region in $C_{f,0.1}$, of the fault "assign wrong label if brightness is in the background of the image". Finally, we visually inspect the cluster centroids in Fig. 6 where we observe that the centroid of the cluster C-2 decodes into an image with a bright outer surrounding.

Answer to RQ5: Identification of the Introduced Fault

Comparing the clusters of fault-free classifier C_o and fault-containing classifier $C_{f,0.1}$, shows that one fault cluster for $C_{f,0.1}$ was significantly closer to the reference cluster, with a cluster distance of 1.09 for C_o compared to 0.41 for $C_{f,0.1}$. On the MNIST dataset, where low cluster distances corresponded to thick digits, we observed similar results with clusters being even better separated from each other. This demonstrates that our method can detect failure regions, showing that we can identify systematic problems in a SUT.

VI. Threats to Validity

Internal Validity: A threat in the evaluation of our method is the selection of the introduced faults. We selected these faults because brightness and thickness clusters were observable in the original classifiers and because these characteristics were also encoded in the CVAEs. If a defect characteristic is not encoded in the CVAEs, our method cannot generate test cases containing that fault. We also carefully select parameters such as in Alg. 1 or the fault prevalence parameter in our classifiers.

External Validity: By applying our method to different datasets such as MNIST, GTSRB, and CIFAR100, as well as different DL models, we minimize this threat. We showed that our method works on MNIST and GTSRB with four different classifiers. On CIFAR100, we found that current CVAEs can not learn a reasonably small latent space while having good reconstruction quality. We argue that this is a limitation of current CVAE technology, and our method would potentially also work on more complex datasets if future CVAEs or in general latent space-based reconstruction methodologies improve.

Construct Validity: We acknowledge that in our approach, the CVAE can only learn failures that are present in the training dataset, meaning that we cannot discover failure regions not present in the original training data. Furthermore, our method can only uncover failures that can be encoded in the CVAE; if a fault cannot be encoded in the latent space, the decoder cannot generate test cases containing this fault. We thus suggest using our method in addition to other established testing techniques.

VII. Related Work

Optimization-Based Test Case Generation: Byun et al. [33] use a two-stage CVAE to search for test cases in the latent space. Unlike our gradient-based search approach, they apply particle swarm optimization with a fitness function that guides test cases towards (1) high prediction uncertainty and (2) a latent space vector close to the zero vector (the center of the latent space). They found that measuring uncertainty using test-time dropout and Monte-Carlo simulation incurs a high computational cost, requiring nearly an hour to generate 10,000 MNIST test cases. In contrast, our gradient-based search method generates the same number of test cases in under 30 seconds.

Zhao et al. [34] use a Wasserstein generative adversarial network (GAN) to generate natural adversarial examples. They used iterative stochastic search and hybrid shrinking search, while the latter was found to be more efficient. A gradient-based search, as in our work, was not applicable, as black-box classifiers were used and domain applications were discrete.

Shukla et al. [35] train a GAN by perturbing latent space features such that it outputs realistic images while belonging to a different class. They created targeted and untargeted attacks by optimizing the generator and discriminator with different loss functions. Their method achieved high attack success rates and was as effective as PGD applied on the pixel space.

Kang et al. [36] propose SINVAD, a methodology that uses a VAE to generate test cases in the space of plausible images that resemble the true training distribution. Contrary to our work, they focus on generating test cases in between classes, whereas we find *conditions* that make test cases from each class fail.

Guo et al. [37] create a method utilizing white-box gradient-based search on the input space called DLFuzz. This differs from us as DLFuzz does not change the semantics, whereas we can generate test cases with any semantics present in the dataset.

Generative Models: Dunn et al. [38] use a GAN to perturb meaningful features in contrast to perturbing pixel values like in adversarial attacks. They argue that this type of perturbation detects different faults than ℓ_p-norm restricted adversarial attacks. The distinction to our work is that GANs are more prone to two issues: (1) mode dropping such that certain parts of the latent space cannot be used to generate output images and (2) mode collapse [39], [40], which significantly reduces the GANs output variety. To prevent issue (1), they keep images close to a seed image by only optimizing over the first layers of the generator. In our proposed method, we aim for an architecture that maximizes the control and variety of the generated output.

VIII. Conclusion

In this work, we propose a domain-independent, data-driven methodology for finding *good* test cases. By using our method, a tester does not need to manually identify input transformations for their specific domain. Instead, we train an autoencoder and compute test cases based on the autoencoder's latent space using gradient-based search. These test cases are *latent space aware*, as the decoder will only generate test cases that lie within its learned data distribution. We cluster the generated test cases by their latent space representation and call such clusters *failure regions*. To the best of our knowledge, this is the only work that uses a CVAE to detect failures automatically in a purely data-driven way and validate the proposed approach.

Results show that our method works with multiple DL classifiers and on multiple datasets, such as MNIST and GTSRB. We are able to generate failing test cases for all classes of the two datasets, while on both datasets surpassing the baseline of random sampling test cases from the latent space by a factor of 25 for GTSRB and 75 for MNIST. To validate our approach, we created a fault-containing derivative of the original SUT.

We could visually as well as quantitatively confirm that our method targeted the previously introduced fault.

We suggest using this approach together with other testing methodologies that require expert knowledge about one's concrete domain, such as metamorphic testing or defect-based testing. Our method thus complements such approaches by finding problems in DL models and shortcomings in the data that testers otherwise might overlook.

Acknowledgment

We thank Jasmine Baral for refactoring the codebase, as well as for implementing and conducting additional experiments.

References

[1] F. Oboril *et al.*, "MTBF model for AVs – From perception errors to vehicle-level failures," in *Intelligent Vehicles Symposium, IV*, Aachen, Germany: IEEE, 2022, pp. 1591–1598. DOI: 10.1109/IV51971.2022.9827006.

[2] A. Pretschner, "Defect-based testing," *Dependable Software Systems Engineering*, vol. 84, pp. 141–163, 2015. DOI: 10.3233/978-1-61499-810-5-141.

[3] N. Humbatova *et al.*, "Taxonomy of real faults in deep learning systems," in *International Conference on Software Engineering, ICSE*, Virtual, South Korea: ACM, 2020, pp. 1110–1121. DOI: 10.1145/3377811.3380395.

[4] E. W. Ayers *et al.*, "PaRoT: A practical framework for robust deep neural network training," in *NASA Formal Methods Symposium, NFM*, Moffett Field, CA, USA: Springer, 2020, pp. 63–84. DOI: 10.1007/978-3-030-55754-6_4.

[5] Y. Feng *et al.*, "DeepGini: Prioritizing massive tests to enhance the robustness of deep neural networks," in *International Symposium on Software Testing and Analysis, ISSTA*, Virtual, USA: ACM, 2020, pp. 177–188. DOI: 10.1145/3395363.3397357.

[6] J. M. Zhang *et al.*, "Machine learning testing: Survey, landscapes and horizons," *IEEE Transactions on Software Engineering*, vol. 48, no. 1, pp. 1–36, 2022. DOI: 10.1109/tse.2019.2962027.

[7] ISO Central Secretary, "Iso/iec/ieee international standard - systems and software engineering–vocabulary," International Organization for Standardization, Tech. Rep., 2017, p. 541. DOI: 10.1109/IEEESTD.2017.8016712.

[8] C. Berghoff *et al.*, "Robustness testing of AI systems: A case study for traffic sign recognition," in *Artificial Intelligence Applications and Innovations, AIAI*, Hersonissos, Crete, Greece: Springer, 2021, pp. 256–267. DOI: 10.1007/978-3-030-79150-6_21.

[9] D. Hendrycks and T. Dietterich, "Benchmarking neural network robustness to common corruptions and perturbations," in *International Conference on Learning Representations, ICLR*, New Orleans, LA, USA, 2019, pp. 1–16. DOI: 10.48550/arXiv.1903.12261.

[10] I. Koren and C. M. Krishna, *Fault-tolerant systems*. San Francisco, CA, USA: Elsevier, 2007, p. 400. DOI: 10.1016/B978-0-12-088525-1.X5000-7.

[11] N. Carlini and D. Wagner, "Towards evaluating the robustness of neural networks," in *Symposium on Security and Privacy, SP*, San Jose, CA, USA: IEEE, 2017, pp. 39–57. DOI: 10.1109/SP.2017.49.

[12] I. J. Goodfellow *et al.*, "Explaining and harnessing adversarial examples," in *International Conference on Learning Representations, ICLR*, San Diego, CA, USA, 2015, pp. 1–11. DOI: 10.48550/arXiv.1412.6572.

[13] C. Szegedy *et al.*, "Intriguing properties of neural networks," in *International Conference on Learning Representations, ICLR*, Banff, AB, Canada, 2014, pp. 1–10. DOI: 10.48550/arXiv.1312.6199.

[14] X. Zeng *et al.*, "Adversarial attacks beyond the image space," in *Conference on Computer Vision and Pattern Recognition, CVPR*, Long Beach, CA, USA: IEEE, 2019, pp. 4297–4306. DOI: 10.1109/CVPR.2019.00443.

[15] J. Stallkamp *et al.*, "Man vs. computer: Benchmarking machine learning algorithms for traffic sign recognition," *Neural Networks*, vol. 32, pp. 323–332, 2012. DOI: 10.1016/j.neunet.2012.02.016.

[16] Y. LeCun *et al.*, *MNIST handwritten digit database*, 2010.

[17] A. Krizhevsky, "Learning multiple layers of features from tiny images," University of Toronto, Tech. Rep., 2009.

[18] H. M. Dipu Kabir *et al.*, "SpinalNet: Deep neural network with gradual input," *IEEE Transactions on Artificial Intelligence*, vol. 4, no. 5, pp. 1165–1177, 2023. DOI: 10.1109/TAI.2022.3185179.

[19] Á. Arcos-García *et al.*, "Deep neural network for traffic sign recognition systems: An analysis of spatial transformers and stochastic optimisation methods," *Neural Networks*, vol. 99, pp. 158–165, 2018. DOI: 10.1016/j.neunet.2018.01.005.

[20] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," in *International Conference on Learning Representations, ICLR*, Banff, AB, Canada, 2014, pp. 1–14. DOI: 10.48550/arXiv.1312.6114.

[21] C. Doersch, "Tutorial on variational autoencoders," 2016. DOI: 10.48550/arXiv.1606.05908.

[22] K. Sohn *et al.*, "Learning structured output representation using deep conditional generative models," in *Neural Information Processing Systems, NIPS*, Montreal, Quebec, Canada: MIT Press, 2015, pp. 3483–3491.

[23] "Variational autoencoder structure (reparameterized)." [Online; accessed 18-February-2025], Wikimedia Commons. (2021), [Online]. Available: https://commons.wikimedia.org/wiki/File:Reparameterized_Variational_Autoencoder.png.

[24] H.-T. D. Liu *et al.*, "Beyond pixel norm-balls: Parametric adversaries using an analytically differentiable renderer," in *International Conference on Learning Representations, ICLR*, New Orleans, LA, USA, 2019, pp. 1–21. DOI: 10.48550/arXiv.1808.02651.

[25] I. Goodfellow *et al.*, *Deep learning*. online: MIT press, 2016, p. 785.

[26] I. Assent, "Clustering high dimensional data," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 2, no. 4, pp. 340–350, 2012. DOI: 10.1002/widm.1062.

[27] D. P. Kingma and J. L. Ba, "Adam: A method for stochastic optimization," in *International Conference on Learning Representations, ICLR*, San Diego, CA, USA, 2015, pp. 1–15. DOI: 10.48550/arXiv.1412.6980.

[28] Solomon Kullback and R. A. Leibler, "On information and sufficiency," *The Annals of Mathematical Statistics*, vol. 22, no. 1, pp. 79–86, 1951.

[29] R. J. Campello *et al.*, "Density-based clustering based on hierarchical density estimates," in *Pacific-Asia Conference on Knowledge Discovery and Data Mining, PAKDD*, Gold Coast, Australia: Springer, 2013, pp. 160–172. DOI: 10.1007/978-3-642-37456-2_14.

[30] L. McInnes and J. Healy, "Accelerated hierarchical density based clustering," in *International Conference on Data Mining Workshops, ICDMW*, New Orleans, LA, USA: IEEE, 2017, pp. 33–42. DOI: 10.1109/ICDMW.2017.12.

[31] L. McInnes *et al.*, "hdbscan: Hierarchical density based clustering," *The Journal of Open Source Software*, vol. 2, no. 11, p. 205, 2017. DOI: 10.21105/joss.00205.

[32] J. Lin, "Divergence measures based on the shannon entropy," *IEEE Transactions on Information Theory*, vol. 37, no. 1, pp. 145–151, 1991. DOI: 10.1109/18.61115.

[33] T. Byun *et al.*, "Manifold-based test generation for image classifiers," in *International Conference on Software Engineering Workshops, ICSEW*, 2020, p. 8. DOI: 10.1145/3387940.3391460.

[34] Z. Zhao *et al.*, "Generating natural adversarial examples," in *International Conference on Learning Representations, ICLR*, 2018, pp. 1–15. DOI: 10.48550/arXiv.1710.11342.

[35] N. Shukla and S. Banerjee, "Generating adversarial attacks in the latent space," in *Conference on Computer Vision and Pattern Recognition Workshops, CVPRW*, 2023, pp. 730–739. DOI: 10.1109/CVPRW59228.2023.00080.

[36] S. Kang *et al.*, "SINVAD: Search-based image space navigation for DNN image classifier test input generation," in *International Conference on Software Engineering Workshops, ICSEW*, Seoul, Republic of Korea: ACM, 2020, pp. 521–528. DOI: 10.1145/3387940.3391456.

[37] J. Guo *et al.*, "DLFuzz: Differential fuzzing testing of deep learning systems," in *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE*, Lake Buena Vista, FL, USA: ACM, 2018, pp. 739–743. DOI: 10.1145/3236024.3264835.

[38] I. Dunn *et al.*, "Exposing previously undetectable faults in deep neural networks," in *International Symposium on Software Testing and Analysis, ISSTA*, Virtual, Denmark: ACM, 2021, pp. 56–66. DOI: 10.1145/3460319.3464801.

[39] A. Srivastava *et al.*, "VEEGAN: Reducing mode collapse in GANs using implicit variational learning," in *International Conference on Neural Information Processing Systems, NIPS*, Long Beach, CA, USA: Curran Associates, Inc., 2017, pp. 3310–3320. DOI: 10.48550/arXiv.1705.07761.

[40] H. Thanh-Tung and T. Tran, "Catastrophic forgetting and mode collapse in GANs," in *International Joint Conference on Neural Networks, IJCNN*, Glasgow, UK: IEEE, 2020, pp. 1–10. DOI: 10.48550/arXiv.1807.04015.