Repairing Failure-inducing Inputs with Input Reflection

Yan Xiao
Southeast University
National University of Singapore
China/Singapore
dcsxan@nus.edu.sg

Yun Lin*
Shanghai Jiao Tong University
National University of Singapore
China/Singapore
dcsliny@nus.edu.sg

Ivan Beschastnikh
University of British Columbia
Vancouver, BC, Canada
bestchai@cs.ubc.ca

Changsheng Sun
National University of Singapore
Singapore
cssun@u.nus.edu

David S. Rosenblum
George Mason University
Fairfax, VA, USA
dsr@gmu.edu

Jin Song Dong
National University of Singapore
Singapore
dcsdjs@nus.edu.sg

ABSTRACT

Trained with a sufficiently large training and testing dataset, Deep Neural Networks (DNNs) are expected to generalize. However, inputs may deviate from the training dataset distribution in real deployments. This is a fundamental issue with using a finite dataset, which may lead deployed DNNs to mis-predict in production.

Inspired by input-debugging techniques for traditional software systems, we propose a runtime approach to identify and fix failure-inducing inputs in deep learning systems. Specifically, our approach targets DNN mis-predictions caused by unexpected (deviating and out-of-distribution) runtime inputs. Our approach has two steps. First, it recognizes and distinguishes deviating ("unseen" semantically-preserving) and out-of-distribution inputs from in-distribution inputs. Second, our approach fixes the failure-inducing inputs by transforming them into inputs from the training set that have similar semantics. We call this process *input reflection* and formulate it as a search problem over the embedding space on the training set.

We implemented a tool called InputReflector based on the above two-step approach and evaluated it with experiments on three DNN models trained on CIFAR-10, MNIST, and FMNIST image datasets. The results show that InputReflector can effectively distinguish deviating inputs that retain semantics of the distribution (e.g., zoomed images) and out-of-distribution inputs from in-distribution inputs. InputReflector repairs deviating inputs and achieves 30.78% accuracy improvement over original models. We also illustrate how InputReflector can be used to evaluate tests generated by deep learning testing tools.

KEYWORDS

Deep Neural Networks, Out-of-Distribution Data, Input Repair, Model Testing.

*Corresponding author

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ASE 2022, October 10 – 14, 2022, Michigan, United States
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9475-8/22/10.
https://doi.org/10.1145/3551349.3556932

1 INTRODUCTION

Complex Deep Neural Network (DNN) models are now being integrated into modern software stacks (e.g., mobile, web, cloud) for tasks as diverse as object detection [47], image classification [21], medical diagnostics [7], and semantic segmentation [37]. As more software relies on DNNs, existing software engineering (SE) techniques and practices must be updated since, for example, incorrect software executions may now be caused by the model.

DNNs require that inputs in deployment come from the same distribution as the training dataset [61]. However, real world inputs that are semantically similar to a human observer may look different to the model. For example, the distribution of data observed by an onboard camera of a self-driving car may change due to environmental factors; the images may be brighter or more blurry than those in the training dataset. Even worse, altogether new and unexpected inputs may be presented to a deployed model. When presented with such unexpected inputs, the model may manifest unexpected behaviors.

To address the issue caused by the failure-inducing inputs, SE and AI communities have proposed a variety of techniques, including delta debugging [29, 39, 63], data augmentation [9, 14, 48, 66], adversarial training [13, 40, 50, 51], and adversarial sample defense techniques [25, 48, 58, 59, 64].

SE researchers have proposed delta debugging techniques to locate and recover from inputs that cause failure. For example, Kirschner et al. [29] proposed an input-debugging technique to deal with failure-inducing inputs in data files like HTML. Nevertheless, techniques on sequential data cannot be easily adapted to image inputs of a conventional deep learning model. By contrast, AI/ML researchers have proposed offline retraining techniques that increase the range of the training dataset so that the trained model can better generalize. However, in the meantime, this generalization is constrained by the model architecture and the finite training dataset. And, real-world inputs in deployment can have unexpected variations that are difficult to capture through data augmentation during training.

In this work, inspired by delta debugging, we investigate whether failure-inducing image inputs in deep learning models can be identified and fixed *without* retraining the model. Driven by this question, we propose an input-reflection technique, InputReflector, which (1) identifies the failure-inducing input, and (2) fixes the input so that the model makes a correct prediction.

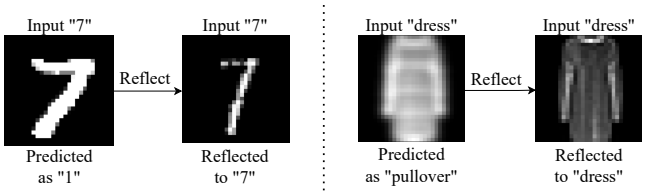

Figure 1: Input Reflection examples.

Figure 1 presents two examples where the input reflection process corrects the classification of a brightened digit "7" and a blurry "dress". The digit "7" presents a different handwriting style (e.g., with thicker strokes than those in the training data), causing the model to mis-predict it as a "1". Our reflection approach, without learning any handwriting style, can reflect this input into a digit "7" from the training dataset to mitigate the mis-prediction. Similarly, the right side of Figure 1 shows a blurry "dress" input. The DNN model incorrectly classifies this input as a "pullover". By contrast, the reflected version is correctly classified as a "dress". Both examples illustrate how input reflection can help DNNs deal with unexpected deviating inputs in production.

We use *representative* and *differentiable* input transformation types (e.g., blur), to learn *auxiliary models*. These models tell us how much an input deviates from normal training samples. Representativeness ensures that these auxiliary models generalize to unseen inputs, while differentiability ensures that the auxiliary models properly measure the distance between an unseen input and a known training sample. In contrast to data augmentation, our use of auxiliary models separates out the concern of model fitting from model generalization, which improves model maintainability.

In this work we make the first attempt to reflect inputs by overcoming the above challenges in the context of computer vision models. We build one auxiliary model as a Siamese model [2] to learn a measurement to discriminate the in-distribution, deviating, and out-of-distribution inputs. In-distribution inputs are fed to the subject model for prediction as usual. Out-of-distribution inputs raise a warning for follow-up human intervention. As for the deviating inputs, we train a Quadruplet network [5] for the auxiliary model and use it to map a deviating input to its similar known training input. Our results show that InputReflector can effectively distinguish deviating and out-of-distribution inputs from in-distribution ones and fix failure-inducing (i.e., deviating) inputs without sacrificing accuracy on in-distribution inputs.

We believe that the approach in InputReflector is a broadly useful building block in SE for ML. To demonstrate this, we use InputReflector to evaluate the quality of tests generated by deep learning test-case generation tools (RobOT [57], ADAPT [34], DeepXplore [44], and DLFuzz [18]).

In summary, we make the following contributions:

- We propose a novel technique to improve DNN generalization by focusing on DNN inputs and treating the DNN as a black-box in production. Our technique contributes (1) a general discriminative measurement to distinguish in-distribution, deviating, and out-of-distribution inputs; and (2) a search method to reflect an unexpected deviating input into one that has similar semantics and is present in the training set.

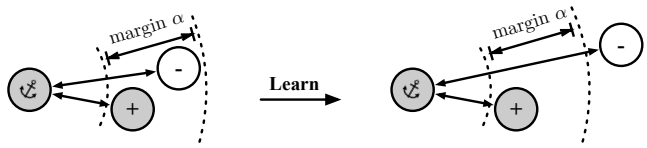

Figure 2: An example of using triplet loss to discriminate the positive and negative samples via an anchor.

- We implemented a tool called InputReflector based on the above approach, and evaluate it on three DNN models (ConvNet, VGG-16, and ResNet-20) trained on the CIFAR-10, MNIST, and FMNIST datasets. Our experimental results show that our tool InputReflector recognizes 75.53% of the unexpected deviating inputs and achieves 30.78% higher accuracy after "reflection". Our implementation is publicly accessible[1].
- We empirically demonstrate that InputReflector can effectively evaluate test cases generated by four deep learning testing tools.

2 BACKGROUND

In this section we review Siamese networks and triplet loss, which are key to our solution. Siamese networks are used to learn representative embeddings of the input, based on which the triplet loss is used to discriminate positive and negative examples.

2.1 Siamese Network

Bromley et al.[2] introduced the idea of a Siamese network to quantify the similarity between two images. Their Siamese network quantifies the similarity between handwriting signatures. Typically, a Siamese network transforms the inputs $\mathbf{x}$ into a feature space $\mathbf{z} = f(\mathbf{x})$, where similar inputs have a shorter distance and dissimilar inputs have larger distance. Given a pair of inputs $\mathbf{x}_1$ and $\mathbf{x}_2$, we use the Euclidean distance to compare $f(\mathbf{x}_1)$ and $f(\mathbf{x}_2)$. Siamese networks [6] have been applied to a wide-range of applications, like object tracking [35, 65], face recognition [53], and image recognition [30]. In this work, we use Siamese networks to evaluate the semantic similarity between an input in deployment and samples in the training dataset.

2.2 Triplet Loss

Triplet loss was first proposed in FaceNet [49]. Given a training dataset where samples are labeled with classes, the triplet loss is designed to project the samples into a feature space where samples under the same class have shorter distance than those under different classes.

Figure 2 shows an example of triplet loss. Given a sample as *Anchor* of class c_1, taking a positive sample (annotated as "+") under c_1 and a negative sample (annotated as "-") under c_2, minimizing the triplet loss is to pull the positive sample closer while pushing the negative sample further from the anchor. Technically, the definition of triplet loss is as follows:

$$\mathcal{L}_{\text{triplet}} = max(Dis(Anchor, Pos) - Dis(Anchor, Neg) + \alpha, 0) \quad (1)$$

[1] https://github.com/yanxiao6/InputReflector

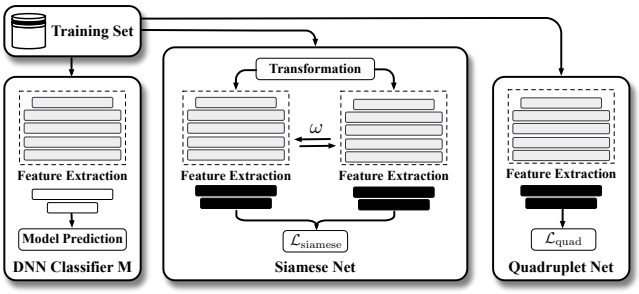

Figure 3: The design of InputReflector in training. $\mathcal{L}_{\text{siamese}}$ is used in Figure 4.(1) to detect deviating data. And, $\mathcal{L}_{\text{quad}}$ is used in Figure 4.(2) to find a reflected sample.

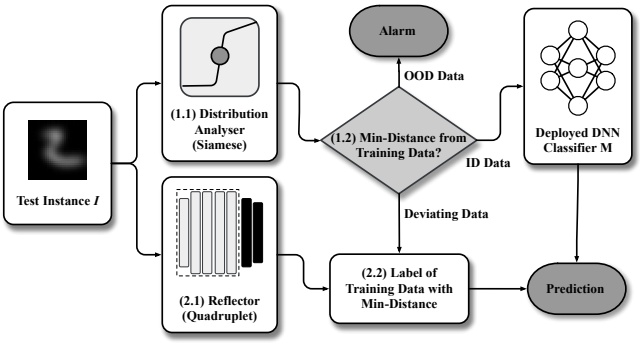

Figure 4: The design of InputReflector in deployment. OOD stands for out-of-distribution data and ID stands for in-distribution data.

In the above, α is the margin, introduced to keep a distance gap between positive and negative samples. The loss can be optimized if the Siamese network can push $Dis(Anchor, Pos)$ to 0 and $Dis(Anchor, Neg)$ to be larger than $Dis(Anchor, Pos) + \alpha$.

Next, we describe our approach, which uses Siamese networks and triplet loss to realize input reflection.

3 DESIGN OF INPUTREFLECTOR

Figures 3 and 4 review the training-time and runtime design of InputReflector. InputReflector uses information during model training to inform its runtime strategy. We review how InputReflector acquires the required information during training later in this section.

At *runtime* (Figure 4), given an input I and a deployed model M, InputReflector follows a two-step approach:

(1) Generate the embeddings of I based on the distribution analyser (1.1) and check how well I conforms to the distribution of the training dataset by using distance as a proxy for semantics (1.2). In this step, InputReflector distinguishes in-distribution, out-of-distribution, and deviating data.
(2) If I is semantically similar to the training samples, then generate the embeddings of I based on the reflector (2.1) and determine which training sample is best used instead of I for the prediction (2.2).

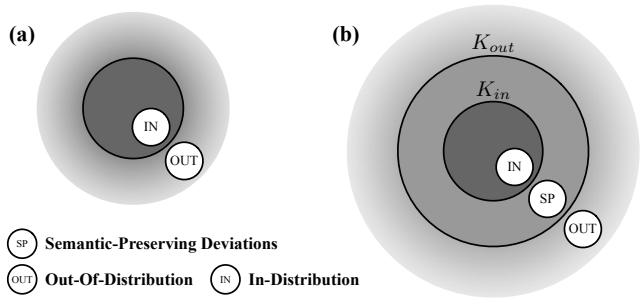

Figure 5: Comparison of problem formulation and thresholds used by (a) previous work, and (b) this paper.

The key to the above steps is correct characterization of the semantics of an input. For this, we rely on sample distribution and design a **Distribution Analyzer** (Section 3.1). This piece of our approach is inspired by prior research [28, 46] that indicates that the distribution of embedding vectors can be used to group semantically similar data (e.g., images of the same class). InputReflector uses the embedding distribution to check the semantic similarity between the test instance and the training data. For inputs that are classified by the Distribution Analyzer as deviating samples, we design the **Reflector** (Section 3.2) to search for a close training sample to replace the runtime test instance. Building on previous work [5, 28, 46], our InputReflector design assumes that samples with similar embedding vectors are semantically similar.

We designed the Distribution Analyzer and Reflector as auxiliary DNNs, for their inherent expressiveness. The Distribution Analyzer, implemented as a Siamese network, captures the general landscape of in-distribution, deviating, and out-of-distribution samples. This is different from previous studies that only distinguish between in-distribution and out-of-distribution samples (Figure 5(a)).

The Reflector, implemented as a Quadruplet network, calculates the detailed distance measurement between the samples. The goal of Siamese network is to push the semantic-preserving deviating input away from both the in-distribution data and out-of-distribution data as shown in Figure 5(b) while the quadruplet network's goal is to pull instances with the same label closer and push away instances with different labels (see Figure 6).

Next, we detail the Distribution Analyzer.

3.1 Distribution Analyzer Design

A challenge in building an effective Distribution Analyzer is *designing a smooth measure for samples that are semantically the same, similar, and dissimilar.* Specifically, we need to design the auxiliary model so that samples can be projected into a space where there is such smoothness between the three categories.

To address this challenge, we require the auxiliary model to learn the smooth measurement from in-distribution, deviating, and out-of-distribution samples as shown in Figure 5(b). For this, let an in-distribution sample be x, a sample in the training data ($\mathcal{X}^t$) be x^t, K_{in} be the distance between x and its most different semantic-preserving (deviating) sample, and K_{out} be the distance between x and its most similar semantic-different (out-of-distribution) sample.

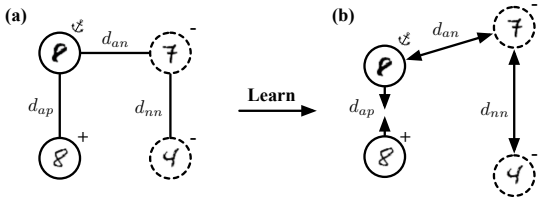

Figure 6: Illustration of learning with a quadruplet network: (a) before learning, and (b) after learning. After loss update, the distance between the anchor and a positive sample d_{ap} decreases, while the distances between the anchor and a negative sample d_{an} and between two different negative samples d_{nn} increase. d_{nn} is a component in the new constraint (Section 3.2.1) over triplet loss, which makes the quadruplet loss more effective.

Any deviating sample x' should conform to:

$$K_{in} < min_{x^t \in X^t}(f(x', x^t)) < K_{out} \quad (2)$$

In this equation, f is a similarity function to calculate the distance between x' and x^t. Since it is hard to evaluate their similarity in high dimension, we design an auxiliary model to learn their feature embeddings with low dimension. We then define f as the distance between the embeddings (ebd) of the two instances, formally, $f(x', x^t) = \left\| ebd(x') - ebd(x^t) \right\|_2$.

To train a quality auxiliary model we prepare *representative* transformed samples (detailed in subsection 4.1) that preserve the semantics of the training data. Specifically, our auxiliary models are trained on three types of data: (1) in-distribution samples (i.e., original training data), (2) deviating samples (i.e., representative transformed data), and (3) one kind of out-of-distribution samples (i.e., extremely transformed data, discussed in Section 4.1).

Next we discuss how we learning feature embeddings and how we set K_{in} and K_{out}.

3.1.1 Siamese Network Training. Existing work on detecting out-of-distribution data uses one threshold as in Figure 5(a). By contrast, the challenge in our work is to find two thresholds, K_{in} and K_{out} in Figure 5(b), so that we can discriminate three types of data. Inspired by the Siamese network with triplet loss, which are used to identify people across cameras [5, 49], we build a Siamese network to learn two thresholds (K_{in} and K_{out}) so that we can discriminate the three kinds of data.

To split the three kinds of data, both training data and their *available* transformed versions are needed. For example, if we consider the blur transformation as an example, then the three kinds of data would be the normal training data x, blurry training data x', and very blurry training data x'' (Figure 7).

Our goal is to learn feature embeddings from the training dataset that push the semantic-preserving deviating input away from both the in-distribution data and out-of-distribution data. Specifically, the Siamese network in the distribution analyzer is designed to make the distance between the very blurry training data and the normal data larger than the distance between the blurry training data and the normal data. Formally, the aim of the Siamese network is to learn f so that $f(x_{c_i}, x_{c_j}) + K_{in} < f(x_{c_i}, x_c^{'}) < f(x_{c_i}, x_{c_j}) + K_{out} < f(x_{c_i}, x_c^{''})$. The loss function of this network is designed to minimize the objective:

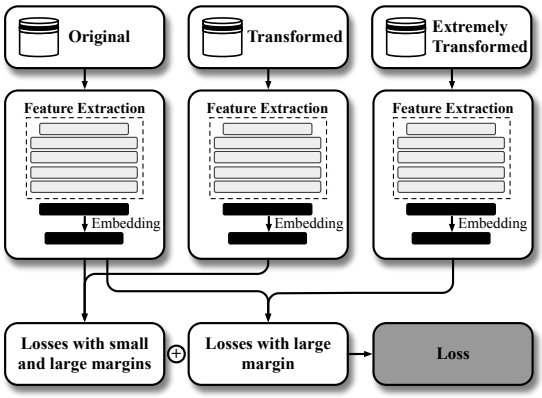

Figure 7: The architecture of the Siamese network used during the training phase.

$$\begin{aligned} L = & max(f(x_{c_i}, x_{c_j}) - f(x_{c_i}, x_c^{'}) + m_1, 0) + \\ & max(f(x_{c_i}, x_c^{'}) - f(x_{c_i}, x_{c_j}) - m_2, 0) + \\ & max(f(x_{c_i}, x_{c_j}) - f(x_{c_i}, x_c^{''}) + m_2, 0) \end{aligned} \quad (3)$$

where m_1 and m_2 are the values of margins and $m_1 < m_2$ whose default values are 0.5 and 1.0 respectively, x_{c_i}, x_{c_j}, and x_c denote instances with the same label c, but $i \neq j$ which means that x_{c_i} and x_{c_j} are different instances.

This loss function consists of three components. Minimizing the first and the last max components pushes both transformed versions of the training data away from the in-distribution data but keep the extremely transformed version further away using a larger margin m_2. After training, the goal of $f(x_{c_i}, x_{c_j}) + K_{in} < f(x_{c_i}, x_c^{'})$ and $f(x_{c_i}, x_{c_j}) + K_{out} < f(x_{c_i}, x_c^{''})$ can be reached. The middle max component is designed to distinguish transformed (x') and extremely transformed data (x'').

Note that the distance between in-distribution data with a large margin m_2, $f(x_{c_i}, x_{c_j}) + m_2$, can serve as a *smaller* upper bound of the distance between transformed data and in-distribution data. Hence, we use $f(x_{c_i}, x_{c_j}) + m_2$ instead of $f(x_{c_i}, x_c^{''})$ in the middle component of Eq. (3)

Figure 7 shows the architecture of our Siamese network. Three kinds ofdata (original training data, transformed training data, and extremely transformed training data from available transformations) are given to the three models that share weights with each other. During training, the DNN classifier M learns to extract meaningful and complex features in feature extraction layers before these are fed to the classification layers. To benefit from these rich representations of feature extraction layers, the Siamese network builds on M to pass these features (feature extraction in Figure 7) through a succession of dense layers. The outputs of the final dense layer are embeddings that we use to minimize the loss function in Eq. (3).

3.1.2 Distribution Discrimination. We designed the distribution analyzer to distinguish between the three kinds of data. In the training module, the embeddings of the original training data and

validation data are generated from the trained Siamese network. For each validation sample, its embedding is used to search for the closest element from the training data by the distance measure f. K_{in} and K_{out} are selected from the distances of embeddings between validation and training data. In deployment, given a test instance x, the minimum distance between its embedding and those of the training data will be a discriminator to determine its category:

$$\begin{cases} \text{In-distribution data} & min_{x_t \in \mathcal{X}^t}(f(x, x^t)) < K_{in} \\ \text{Out-of-distribution data} & min_{x_t \in \mathcal{X}^t}(f(x, x^t)) > K_{out} \\ \text{Deviating data} & otherwise \end{cases} \quad (4)$$

Note that unlike other enhanced classifier approaches [55, 61], InputReflector understands potential model prediction risks and can be made to discard runtime test instances that are unsuitable for model prediction (the **Out-of-Distribution Alarm** in Figure 4).

3.2 Reflector Design

When the distribution analyzer recognizes an input instance as a deviation from the in-distribution data, the *reflector* will map this input to the closest sample in the training data. Reflector's design relies on separation of concern [45]: we let the subject model focus on fitting while tasking the auxiliary models with generalizability learning.

3.2.1 Quadruplet Network Training. Inspired by Chen et al. [5] who built a deep quadruplet network for person re-identification, we construct a quadruplet network using parts of the subject DNN classifier M. As illustrated in Figure 3 and Figure 6, the goal of the Quadruplet network is to learn the feature embeddings so that instances of intra-class will be clustered together but those of inter-class will be further away. As a result, when an unexpected deviating instance is presented, its embedding can be used to search for the closest samples from the training data, and these can be used as a proxy for identifying the input class.

Although the Quadruplet network construction is similar to the Siamese network, they have different goals. The quadruplet network's aim is to pull instances with the same label closer and push away instances with different labels (Figure 6). The Siamese network's goal is to push the deviating input away from both the in-distribution data and out-of-distribution data.

Algorithm 1 lists the details of our Quadruplet network construction and how we use it in deployment. The Quadruplet network consists of the feature extraction layers of the target DNN classifier M (line 2) and a succession of dense layers (lines 2-4), which are trained by the quadruplet loss.

It introduces a new constraint on top of the triplet loss, which pushes away negative pairs from positive pairs w.r.t different anchor images [23].

We use $(loss_{an}+loss_{nn})$ as the Quadruplet network loss. The first part, $loss_{an}$, is the traditional triplet loss that is the main constraint. After convergence, the maximum intra-class distance (e.g., d_{ap} in Figure 6) is required to be smaller than the minimum inter-class distance (e.g., d_{an}) with respect to the same anchor.

The second part, $loss_{nn}$, is auxiliary to the first loss and conforms to the structure of traditional triplet loss but has different triplets. Its aim is to make the maximum intra-class distance (e.g., d_{ap}) smaller than the minimum inter-class distance regardless of whether pairs contain the same anchor (e.g., d_{nn}). We use two different margins ($m_1 > m_2$) to balance the two constraints.

Algorithm 1: Quadruplet Network Training and Inference

Input: Target DNN classifier: M;
Input instances in D_{train}: $\mathcal{X}^t$, true labels in D_{train}: $\mathcal{Y}^t$;
Deviating test instances: $\mathcal{X}^d$
Output: [Training] Trained Quadruplet network;
Output: [Inference] Alternative predictions of $\mathcal{X}^d$

```
1  # Training
2  base_model = M.get_layer("cnn").output;
3  Add several dense layers after base_model as embeddings embed;
4  model = Model(base_model.input, embed);
5  model.compile(optimizer, Quadruplet_loss);
6  model.fit(X^t, Y^t);
7  # Inference
8  embed^t = model.predict(X^t);
9  for x in X^d do
10     embed(x) = model.predict(x);
11     idx^T = argmin(dist(embed(x), embed^t));
12     prediction_new = Y^T[idx^T];
13 end
```

The primary challenge in constructing the Quadruplet network is how to sample the quadruplet from the training data. Due to space constraints, the reader can find the technical details in [**?**]. The sample mining process of the Siamese network in Section 3.1 also uses the same process.

As discussed in [5], $(loss_{an} + loss_{nn})$ leads to a larger inter-class variation and a smaller intra-class variation as compared to the triplet loss. And, as we will show in the evaluation (Section 4), this loss combination improves generalization. The benefit of this arrangement is that the Quadruplet network can be used to help generalize the subject classifier M to unexpected deviations from in-distribution data.

3.2.2 Input Reflector for unexpected deviating instances. Lines 8-13 in Algorithm 1 list the procedure for the reflector using the trained Quadruplet network in deployment. Given an instance x that is classified as deviating by the distribution analyzer, the reflector first obtains its feature embedding learned by the Quadruplet network (line 10).

Next, the distance between $embed(x)$ and those embeddings of the training data generated in the training module (line 8) are calculated so that the minimum distance can be found.

Finally, the deviating test instance x is reflected to the specific training instance with the minimum distance. The label of this training instance becomes the alternative prediction for x (lines 11-12).

Now that we have detailed InputReflector's design, we evaluate its performance on different datasets and DL models.

4 EVALUATION

Our evaluation aims to answer the following research questions:

RQ1. How effective is the distribution analyzer in distinguishing three types of data? To evaluate the distribution analyzer, we compare its area under the receiver operating characteristic curve

(AUROC) with two related techniques: Generalized ODIN (G-ODIN) [24] and SelfChecker [61]. Specifically, we evaluate the accuracy of the distribution analyzer on detecting out-of-distribution and deviating data. Since the authors of G-ODIN did not release the code nor hyperparameter values, we implemented G-ODIN ourselves and used grid search to find the best combination of hyperparameters on the validation dataset, from which we obtained close results on the dataset in the G-ODIN paper. As there are two parts to SelfChecker, alarm and advice, to answer this RQ, we compare the distribution analyzer with SelfChecker's alarm accuracy.

RQ2. What is the accuracy of the reflection process on unseen deviating data? We answer this question by comparing the accuracy of reflector (Section 3.2) in InputReflector against the accuracy of the subject model M and advice accuracy of SelfChecker on unseen deviating inputs.

RQ3. What is the performance of InputReflector on the in-distribution and deviating input in deployment? Since the distribution analyzer is designed to learn K_{in} and K_{out} from available transformed versions of training data (blur as an available transformation and other transformations as unseen in this work since blur is the most informative and representative), we compare the accuracy of InputReflector against the accuracy of original subject model (M), subject model with data augmentation ($M + Aug$), and SelfChecker on both in-distribution inputs and the deviating inputs.

RQ4. What is InputReflector's performance when training data includes adversarial examples? Adversarial examples in training data decrease the accuracy of M. We evaluate how well InputReflector performs when training data contains adversarial examples.

RQ5. What InputReflector's overhead in training and in deployment? To evaluate the efficiency of InputReflector, we calculated its overall time overhead and compared it against the ones of $M + Aug$ and SelfChecker. In the training phase, $M + Aug$ needs to conduct data generation and extra training. But, we regard the generation part as having zero cost and only consider the extra training time. Augmentation also does not have extra time cost in deployment. For SelfChecker and InputReflector, we measure their time overheads during training and deployment.

RQ6. How well does InputReflector generalize to unseen transformations? We first train InputReflector with blur+zoom inputs. Then, we evaluate InputReflector's generalizability by using it to detect and fix inputs transformed with contrast/bright/rotate/shear.

4.1 Experimental Setup

We evaluate InputReflector on three popular image datasets (MNIST [32], FMNIST [60], and CIFAR-10 [31]) using three DL models (ConvNet [27], VGG-16 [52], and ResNet-20 [21]). To reduce variance due to randomness, we ran each experiment five times and report the average.

Datasets. We prepare three types of datasets, i.e., in-distribution, deviating, and out-of-distribution datasets.

(1) In-distribution datasets. In-distribution data conform to the distribution of the training data. As in prior studies [24, 33, 36], we regard the testing data of each dataset (i.e., MNIST, FMNIST, and CIFAR-10) as the in-distribution data.

(2) Deviations from in-distribution datasets. We selected six kinds of transformations used by the computer vision community [10, 14] to construct the transformed data of in-distribution testing datasets: blur, bright, contrast, zoom, rotation, and shear. These operations transform an image x as follows:

- $zoom(x, d)$: zoom in x with a zoom degree d in range [1,5).
- $blur(x, d)$: blur x using Gaussian kernel with degree d in [0,5).
- $bright(x, d)$: uniformly add a value for each pixel of x with a degree d in range [0, 255) and then clip x within [0, 255].
- $contrast(x, d)$: scale the RGB value of each pixel of x by a degree d in range (0, 1] and then clip x within [0, 255].
- $rotation(x, d)$: rotate x by a degree d in range [0, 360).
- $shear(x, d)$: horizontally shear x with shear factor d in [0, 1)

We search for crash transformation degrees (d_{crash}) for each training and testing image on which the original classifier begins to mis-predict. The instances with such degrees then serve as the deviations of in-distribution data. We manually confirmed that each instance is visually recognizable.

(3) Out-of-distribution datasets. There are two kind of out-of-distribution data: the first one is the same as the existing work — another dataset which is completely different from the training data. The second one is the extremely transformed training and testing data using extreme degrees of the six transformations. Taking the maximum degree of a transformation (d_{crash}), d_{max}, we construct the extremely transformed sample from a normal sample by applying the transformation degree as $d_{max} + d_{crash}$.

Specifically, we use FMNIST, MNIST, SVHN [41] as the first totally different dataset for MNIST, FMNIST, CIFAR-10 respectively. The extremely transformed data cannot be recognized by both DL models and humans. We present the performance of the distribution analyzer on both datasets: another dataset and extremely transformed testing data.

Subject DL models. We use three DL models as our subject models: ConvNet [27], VGG-16 [52], and ResNet-20 [21]. These are commonly-used models that range from small to large, with the number of layers ranging from 9 to 20 [61].

Evaluation metrics. We adopt two measures to evaluate the distribution analyzer. AUROC plots the true positive rate (TPR) against the false positive rate (FPR) by varying a threshold, which can be regarded as an averaged score that can be interpreted as the model's ability to discriminate between positive and negative inputs. TNR@TPR95 is the true negative rate at 95% true positive rate, which simulates an application requirement that the recall should be 95%. Having a high TNR under a high TPR is much more challenging than having a high AUROC score. We use classical model accuracy on testing data to evaluate the performance of both sample selector and InputReflector. We also adopt the Wilcoxon rank sum test (p-value <.05) to evaluate whether the performance between InputReflector and baselines are significantly different. We quantify the difference in performance using effect size (i.e., Cliff's δ). Following [26], we classify the effect size as negligible (0 < Cliff's δ < 0.147), small (0.147 < Cliff's δ < 0.33), medium (0.33 < Cliff's δ < 0.474) or large (Cliff's δ > 0.474).

Table 1: Performance of two methods on detecting out-of-distribution data.

	AUROC			TNR@TPR95		
	ConvNet	VGG-16	ResNet-20	ConvNet	VGG-16	ResNet-20
	G-ODIN/Siamese			G-ODIN/Siamese		
			blur			
MT	69.50/69.29	60.88/85.15	78.00/78.38	13.67/50.11	02.00/56.23	45.37/48.25
FT	73.99/69.85	71.51/57.58	74.86/75.50	37.16/50.05	38.26/28.40	27.17/44.98
CF	77.09/79.97	45.28/85.28	73.70/76.51	52.69/55.70	04.44/36.95	40.96/39.94
			bright			
MT	58.77/73.70	60.23/81.83	89.00/88.11	21.66/49.96	38.30/46.43	57.01/67.57
FT	78.62/84.70	74.14/73.45	73.79/76.82	40.18/59.94	49.54/33.03	33.02/41.62
CF	77.04/95.39	44.77/86.90	77.00/82.71	55.20/79.80	04.34/37.69	46.65/60.06
			contrast			
MT	58.71/71.73	73.74/83.43	88.38/84.39	20.37/49.90	31.87/48.71	58.90/69.68
FT	74.82/81.98	71.77/70.30	79.33/75.79	38.70/57.32	41.12/31.23	30.39/41.46
CF	72.73/89.78	41.73/82.51	69.22/79.95	50.02/70.45	03.44/32.64	39.19/53.08
			zoom			
MT	67.43/85.15	66.85/77.19	75.24/82.75	03.97/54.38	03.82/43.96	32.92/56.41
FT	81.28/83.77	70.67/70.10	81.56/83.05	44.66/57.68	33.30/31.66	37.57/51.03
CF	88.25/94.35	47.60/90.11	88.92/91.11	64.49/74.78	05.87/40.93	53.71/56.82
			rotation			
MT	55.31/74.18	59.15/63.03	71.61/72.35	4.75/19.04	4.99/18.67	15.12/22.18
FT	36.13/76.53	55.62/58.36	79.44/76.77	3.45/44.18	12.04/11.01	8.87/34.09
CF	59.13/73.93	56.38/64.82	53.14/75.73	3.00/16.22	8.36/11.39	10.01/20.08
			shear			
MT	53.75/90.17	62.33/78.37	57.91/74.29	7.90/68.14	6.70/23.86	19.64/46.70
FT	25.14/86.44	60.77/70.23	71.25/74.27	4.17/51.14	9.89/20.54	11.34/60.84
CF	57.64/67.77	60.89/77.64	61.33/84.75	4.59/26.87	3.69/29.88	7.26/64.12
			Mean			
MT	60.58/**77.37**	63.86/**78.17**	76.69/**80.05**	12.05/**48.59**	14.61/**39.64**	38.16/**51.80**
FT	61.66/**80.55**	**67.41**/66.67	76.71/**77.03**	28.05/**53.39**	**30.69**/25.98	24.73/**45.67**
CF	71.98/**83.53**	49.44/**81.21**	70.55/**81.79**	38.33/**53.97**	5.02/**31.58**	32.96/**49.02**

MT, FT, and CF stand for MNIST, FMNIST, and CIFAR-10, respectively.

Runtime Configuration. We conducted all experiments on a Linux server with Intel i9-10900X (10-core) CPU @ 3.70GHz, one RTX 2070 SUPER GPU, and 64GB RAM, running Ubuntu 18.04.

4.2 Results and Analyses

We now present results that answer our four research questions.

RQ1. Distribution Analyzer.

We discuss the results on detecting out-of-distribution and deviating-distribution data respectively.

Detection of Out-of-distribution Data. Table 1 presents the AUROC and TNR@TPR95 of the distribution analyzer (Siamese) and Generalized ODIN (G-ODIN) to detect out-of-distribution data from in-distribution and deviating data. The out-of-distribution data here is totally differently data (as is done in existing work [24]) and extremely transformed data (blur, bright, contrast, zoom, rotation, and shear). This is why the results in Table 1 are different from previous work (which focuses on detecting data that is totally differently from in-distribution data).

Table 1 shows that, Siamese outperforms G-ODIN in most of the cases except for FMNIST on VGG-16. We can see that both techniques have similar AUROC on FMNIST. The main reason is that the Siamese model cannot distinguish well between the deviating data and out-of-distribution data (MNIST). When FMNIST data is contrasted, e.g., trouser, it will be similar to digit "1". For other settings, Siamese achieves good performance on both different datasets and extremely transformed data, significantly outperforming G-ODIN. Overall, Siamese achieves 78.49% AUROC against 66.54% for G-ODIN. Its performance over G-ODIN has a statistical significance: p-value < 10E-3, with a large effect size of 0.491.

Detection of Deviating-distribution Data. Table 2 compares the performance of G-ODIN, SelfChecker, and our distribution analyzer in detecting deviating data from in-distribution data on three DL models and three datasets with six transformations.

The distribution analyzer (Siamese) significantly outperforms G-ODIN and SelfChecker (p-value < 10E-5 and 10E-6) with large effect sizes of 0.633 and 1. Except for CIFAR-10 on ConvNet, Siamese outperforms G-ODIN. The reason it fails is that ConvNet is too simple a DNN for CIFAR-10 on which Siamese cannot learn informative embeddings. But G-ODIN decomposed the softmax score to distinguish confident inputs from unconfident ones, which is hardly affected by the model architecture [24].

In all other settings, Siamese achieves both high AUROC (averagely 87.33% against 70.58% of G-ODIN) and TNR@TPR95 (averagely 54.23% against 31.54% of G-ODIN) that is harder to reach. That means that Siamese can recognize more in-distribution data with the 95% recall of deviating data, which is important for the downstream reflection process. Given by the threshold search from validation dataset, Siamese can correctly detect on average 75.53% of deviating data. SelfChecker has bad performance since it is constrained by the assumption that the testing instance conforms to the distribution of training dataset. But the deviating testing data here share similar semantics with the training data but are far away distribution-wise.

Figure 8 visualizes the separation between three types of data on MNIST with two labels ("2" and "7"). This is generated from the learned embeddings from Siamese network and then mapped into hyperspace by t-SNE [54], which is a dimension-reduction technique to project high-dimensional data into visible two-dimensional data.

We can see that the three types of data are distant from each other and that the distance between in-distribution and deviating data is *shorter* than the distance between in-distribution and out-of-distribution data.

> ***Result 1:*** *the distribution analyzer effectively detects out-of-distribution and deviating data with 78.49% and 87.33% AUROC. G-ODIN is second-best with 66.54% and 70.58% AUROC.*

RQ2. Reflector Accuracy.

Figure 9 compares the classification accuracies of the subject model M, SelfChecker, and the reflector on deviating data generated by unseen transformations. All six kinds of deviating data have not been seen by these models.

We find that the reflector achieves higher accuracy than M and SelfChecker. The poor performance of SelfChecker is predictable since the deviating data does not strictly conform to the distribution of the training data. The reflector uses a quadruplet network that learns an embedding to pull instances with the same label closer together and push instances with different labels further apart.

Table 2: Performance of three methods on detecting deviating data.

Deviating	AUROC			TNR@TPR95		
	ConvNet	VGG-16	ResNet-20	ConvNet	VGG-16	ResNet-20
	G-ODIN/SelfChecker/Siamese			G-ODIN/Siamese		
blur						
MNIST	74.62/64.47/99.96	59.14/60.19/99.73	71.01/70.10/99.81	19.21/99.99	06.61/55.37	24.33/93.92
FMNIST	93.61/62.62/99.39	85.90/63.20/96.16	78.72/61.68/94.56	66.43/99.97	47.43/59.52	29.90/57.89
CIFAR-10	90.28/64.44/90.61	52.47/64.91/86.16	82.20/64.86/88.35	47.37/32.41	06.12/21.97	21.27/37.26
bright						
MNIST	99.95/61.03/99.99	50.14/57.40/99.61	58.08/66.37/97.48	100.0/99.96	03.47/66.31	06.29/86.96
FMNIST	89.13/67.31/99.84	83.24/67.31/91.94	72.65/71.95/97.58	48.97/88.84	48.30/19.94	22.10/88.12
CIFAR-10	88.86/60.78/75.23	57.69/51.24/73.00	73.95/66.28/82.13	45.95/18.08	06.55/13.23	15.02/26.65
contrast						
MNIST	99.93/60.12/99.99	66.79/69.29/99.62	55.13/64.17/97.12	100.0/100.0	04.58/61.18	05.39/84.30
FMNIST	90.82/54.90/99.90	76.06/45.69/95.51	71.76/63.97/97.77	55.98/90.46	26.93/29.09	24.01/88.41
CIFAR-10	93.37/58.14/75.87	63.95/56.07/87.35	85.32/64.15/89.39	64.07/16.24	11.33/29.13	24.64/35.79
zoom						
MNIST	83.33/45.87/99.67	68.41/44.76/98.47	88.42/64.15/97.42	30.39/98.91	04.00/71.80	51.83/86.69
FMNIST	80.63/59.24/95.45	65.76/67.02/86.04	63.94/59.88/88.54	32.10/68.60	07.42/33.36	14.55/49.09
CIFAR-10	80.34/49.88/73.46	42.54/45.82/76.82	62.34/60.67/76.42	31.56/15.05	03.79/17.27	13.55/17.20
rotation						
MNIST	76.12/51.29/96.62	71.77/53.36/70.16	70.58/61.38/85.99	9.98/83.19	3.31/30.05	56.66/49.80
FMNIST	29.94/40.83/90.96	74.23/47.31/72.21	41.09/50.52/71.93	39.07/55.43	19.54/20.31	18.50/23.09
CIFAR-10	55.65/49.35/55.10	54.48/50.27/60.19	78.88/57.04/72.46	53.41/13.22	44.84/38.88	56.20/54.42
shear						
MNIST	65.22/57.46/98.97	66.47/49.84/77.04	73.50/61.59/98.88	14.99/96.19	5.51/38.62	54.50/94.79
FMNIST	26.15/51.28/96.27	75.88/50.17/74.04	42.39/57.31/71.87	25.37/80.87	20.47/23.80	18.88/18.45
CIFAR-10	61.75/45.17/60.73	58.94/46.82/66.69	87.66/60.31/89.21	60.49/22.05	51.06/51.80	78.79/74.60
Mean						
MNIST	83.20/56.71/**99.20**	63.79/55.81/**90.77**	69.45/64.63/**96.12**	45.76/**96.37**	4.58/**53.89**	33.17/**82.74**
FMNIST	68.38/56.03/**96.97**	76.85/56.78/**85.98**	61.76/60.89/**87.04**	44.65/**80.70**	28.35/**31.00**	21.32/**54.18**
CIFAR-10	**78.38**/54.63/71.83	55.01/52.52/**75.04**	78.39/62.22/**82.99**	**50.48**/19.51	20.62/**28.71**	34.91/**40.99**

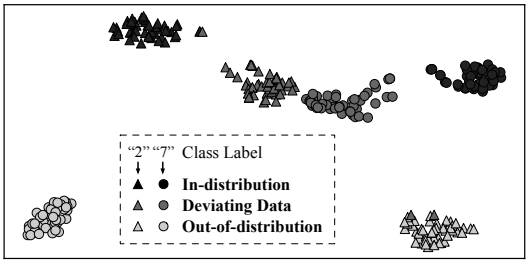

Figure 8: Visualization of the feature embedding learned by the Siamese network.

Therefore, unseen deviating data is embedded close to training samples with similar semantics. Using the labels of these nearby training samples therefore improves performance.

> ***Result 2:*** *the reflector can improve the accuracy of the subject models by 47.34% on unseen deviating data.*

RQ3. InputReflector Performance.

To show the effectiveness of InputReflector, Table 3 presents the accuracies of the subject model M, M with data augmentation ($M + Aug$), SelfChecker, and InputReflector on the in-distribution and deviating testing data from both seen (blur) and unseen transformations (bright, contrast, zoom, rotation, and shear). Both $M + Aug$ and InputReflector use the "blur" version of the training data.

The size of augmented data is the same as that of the original training dataset. $M + Aug$ enhances the original training data with the blurry version of the data to retrain M. InputReflector uses blurry training data to train the auxiliary models, i.e., Siamese and Quadruplet network, instead of retraining M.

InputReflector significantly outperforms M and SelfChecker (both p-value < 10E-6) with large effect sizes of 1, and is comparable to $M + Aug$. As we can see, SelfChecker cannot significantly improve M's accuracy since it has poor performance providing alternative predictions for deviating data. Both data augmentation (average 79.71%) and InputReflector (average 80.09%) improve M accuracy. The difference in their accuracies is not significant, indicating that InputReflector has similar generalization ability as data augmentation.

> ***Result 3:*** *InputReflector and data augmentation have similar generalization ability with accuracies of 80.09% and 79.71%, respectively. Both significantly outperform M and SelfChecker. InputReflector can additionally detect out-of-distribution data.*

RQ4. InputReflector Performance with Adversarial Data Augmentation.

Not all data used in data augmentation can enhance M. For example, the accuracy of M will degrade when using adversarial examples for data augmentation.

To show the sensitivity of $M + Aug + Adv$ and InputReflector $+Adv$ on adversarial/poisonous data, we include both adversarial

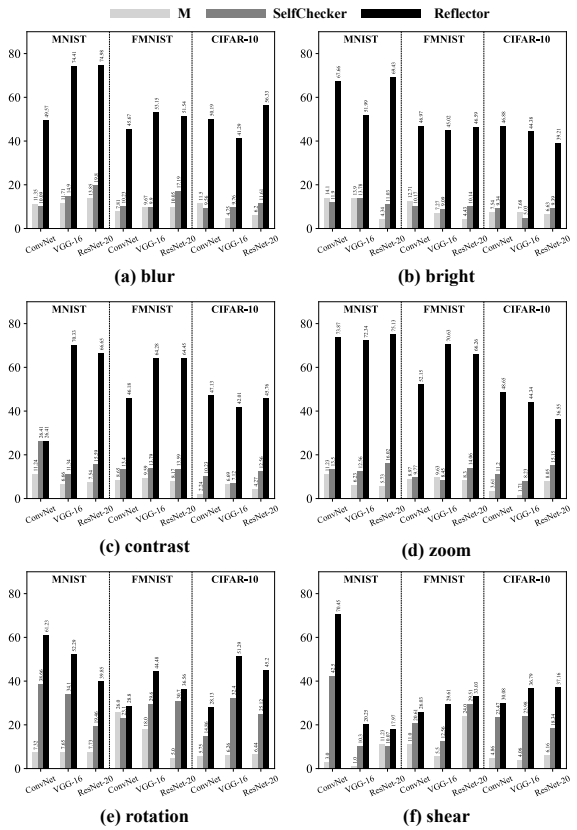

Figure 9: Accuracy of three methods (original model M, SelfChecker, and Reflector) on unseen deviating input.

and blurry training data for both methods. We use the Fast Gradient Sign Method (FGSM) [16] to generate adversarial examples, whose average attack success rate on our subject models is 78.6% (i.e., 78.6% of the adversarial samples can fool the subject classifier). Table 4 shows the mean accuracies of $M + Aug + Adv$ and InputReflector $+Adv$ on the in-distribution data and the in-distribution plus deviating data with the six transformations.

As shown in Table 4, the accuracy of InputReflector $+Adv$ is significantly higher than M and $M + Aug + Adv$ (both p-value < 10E-3) with effect sizes of 0.401 and 0.194 respectively. The low accuracies of $M + Aug + Adv$ on in-distribution data indicate that data augmentation with adversarial examples will harm model performance even on the normal testing data. Retraining the subject model with adversarial samples is known to compromise the model generalizability [3, 4, 20, 50, 56]. The retrained model with data augmentation usually overfits those adversarial samples and suffers from low accuracy on testing dataset.

By contrast, InputReflector's accuracy on in-distribution data is robust to adversarial examples. However, the generalization ability of both methods is reduced, as indicated by the lower accuracies on in-distribution and deviating data (as compared to using blurry training data as augmentation). InputReflector's robustness is aided by sample mining, which is used in both Siamese and Quadruplet networks, and which filters out many adversarial examples. FGSM was proposed in 2014, and there are more powerful adversarial sample generation techniques (i.e., with higher attack success rates)

Table 3: Accuracy of four methods on the in-distribution and deviating testing data.

	ConvNet	VGG-16	ResNet-20
	M/M+Aug/SelfChecker/InputReflector		
	blur		
MT	55.16/99.38/57.19/99.36	55.65/99.55/60.45/99.02	56.65/99.44/55.39/99.40
FT	50.04/93.67/54.30/92.48	51.94/94.41/55.14/91.44	51.40/91.60/52.17/91.57
CF	45.84/78.43/47.29/76.08	46.85/88.59/44.39/88.42	43.91/79.50/50.78/79.20
	bright		
MT	55.10/93.14/56.09/92.11	52.91/96.49/59.81/95.22	52.59/91.64/60.37/94.43
FT	50.62/75.13/49.82/81.18	51.92/75.32/56.12/77.40	50.53/80.17/60.25/84.65
CF	41.89/64.88/47.94/65.23	45.33/70.99/48.45/77.43	44.84/65.73/46.86/65.14
	contrast		
MT	55.11/92.76/57.83/90.77	53.12/95.74/58.56/93.83	53.49/92.67/60.06/94.62
FT	50.46/75.73/59.26/80.09	51.90/76.30/58.30/77.10	50.46/80.69/57.95/86.66
CF	41.21/74.87/42.64/75.92	47.82/79.96/49.18/83.14	42.95/72.88/45.83/72.76
	zoom		
MT	56.54/90.52/57.23/85.92	56.74/84.90/56.38/84.87	51.89/83.32/59.43/90.37
FT	52.49/79.56/51.65/78.49	50.74/84.90/50.28/81.66	48.59/80.59/49.54/81.54
CF	43.86/76.41/49.25/77.06	48.32/86.39/50.03/84.09	44.13/78.38/53.56/78.26
	rotation		
MT	53.25/80.07/55.78/78.50	53.62/80.95/62.45/80.36	53.59/78.35/55.18/78.25
FT	46.27/66.95/50.91/69.95	47.19/71.07/52.59/69.62	46.63/70.03/48.30/68.94
CF	43.45/56.80/42.16/56.50	47.96/71.65/58.15/71.85	41.51/63.34/52.18/63.50
	shear		
MT	49.50/83.81/56.32/85.03	49.80/81.24/59.55/80.93	55.34/79.49/60.37/79.68
FT	46.19/67.95/49.03/66.99	47.38/74.72/53.56/76.45	46.50/68.15/45.32/66.80
CF	43.00/59.70/45.87/61.55	46.86/65.74/51.20/65.76	41.37/59.82/46.33/57.07
	Mean		
MT	54.11/**89.95**/56.74/88.62	53.64/**89.81**/59.53/89.04	53.93/87.49/58.47/**89.46**
FT	49.35/76.50/52.50/**78.20**	50.18/**79.45**/54.33/78.95	49.02/78.54/52.26/**80.03**
CF	43.21/68.51/45.86/**68.72**	47.19/77.22/50.23/**78.45**	43.12/**69.94**/49.26/69.32

MT, FT, and CF stand for MNIST, FMNIST, and CIFAR-10, respectively.

[8, 62]. With more advanced adversarial examples, the advantage of InputReflector over data augmentation will *increase*, since data augmentation overfitting will get worse.

Result 4: *InputReflector is more tolerant than data augmentation when considering adversarial examples. InputReflector +Adv significantly outperforms M and M + Aug + Adv (both p-value < 10E-3) with effect sizes of 0.401 and 0.194, respectively.*

RQ5. Overhead.

We measured the time consumption of $M + Aug$, SelfChecker, and InputReflector during training and deployment. Table 5 shows that $M + Aug$ and InputReflector have similar training times while SelfChecker has higher overhead since it needs to estimate the distribution of each layer. For each test instance in deployment, InputReflector has more time overhead than $M + Aug$ because of the distance calculation and the search for closest training data. But, this process is faster than SelfChecker.

Given a more complex dataset with inputs of larger size, the training time for all techniques will increase. However, the deployment-time reflection process in InputReflector is not sensitive to the input size since the time is spent on estimation and calculation of embedding vectors that have a fixed size.

Table 4: Accuracy using adversarial examples as data augmentation.

Accuracy	In-distribution			In-distribution+Deviating		
	ConvNet	VGG-16	ResNet-20	ConvNet	VGG-16	ResNet-20
	M/M+Aug+Adv/InputReflector+Adv			M/M+Aug+Adv/InputReflector+Adv		
MNIST	98.97/97.46/**99.57**	99.58/97.54/**99.65**	99.44/98.31/**99.48**	53.40/84.60/**85.61**	53.15/81.44/**83.06**	54.06/80.85/**80.88**
FMNIST	**92.27**/92.08/**92.27**	94.20/92.20/**94.57**	92.75/91.34/**93.58**	48.56/71.69/**72.57**	49.45/74.42/**78.73**	48.40/74.61/**75.55**
CIFAR-10	80.17/76.80/**80.71**	88.95/87.59/**89.31**	81.62/75.38/**82.96**	43.21/**65.68**/64.80	47.24/71.00/**73.00**	42.70/62.18/**64.81**

Table 5: Time overhead.

Time	M+Aug	Selfchecker	InputReflector
Training	41.97m	>1h	39.5m
Deployment	0.98s	34.33s	2.86s

Result 5: *InputReflector has similar training time as data augmentation, but its inference time per test instance is higher (2.86s vs 0.98s).*

RQ6. Generalizability.

InputReflector is designed to handle unseen deviated samples. A natural question to ask is how well InputReflector generalizes from transformations used during training (e.g., blur) to handle unknown transformations in deployment (e.g., brightening). We briefly investigate this question by training InputReflector with blur+zoom inputs to detect and fix contrast+bright+rotate+shear inputs. Table 6 shows that InputReflector achieves 2.48% higher accuracy on average than training with blur inputs alone.

These preliminary results indicate that composition of basic transformations can further improve InputReflector's transformation generalizability. From an engineering perspective, we believe that a production version of InputReflector would need to use a representative set of transformations so that InputReflector can handle more types of unseen deviated samples.

5 EVALUATING MODEL TESTING TOOLS WITH DISTRIBUTION ANALYZER

So far we have shown that InputReflector achieves good performance on detecting the failure-inducing inputs and mitigating their side effects. We now demonstrate that InputReflector is also useful to evaluate frameworks that generate tests for DL models, such as RobOT [57], ADAPT [34], DeepXplore [44], and DLFuzz [18]. Specifically, we use InputReflector to detect how the samples generated with those tools conform to the distribution of the training dataset. The more different the samples are from the training dataset, the more diverse they are, thus the higher their quality.

Table 7 shows the performance of Distribution Analyzer in distinguishing the generated test cases from in-distribution data for tests generated by the aforementioned DL testing tools. DeepXplore and DLFuzz achieve much lower AUROC values (56.22% on average) than RobOT and ADAPT, indicating that DeepXplore and DLFuzz do not generate diverse tests. This aligns with the finding in [57]. On average, RobOT achieves higher AUROC than ADAPT, but RobOT performs worse on MNIST. An in-depth look reveals that ADAPT tends to generate more test cases around a seed than RobOT, while ADAPT generates more diverse test cases for simple datasets like MNIST.

Table 6: Generalization capacity of InputReflector. Accuracy on in-distribution and deviating testing data with inputs under contrast/bright/rotate/shear transformations. InputReflector was trained on blur+zoom data.

	ConvNet	VGG-16	ResNet-20
	M/M+Aug/InputReflector		
	blur		
MNIST	55.16/99.44/99.37	55.65/99.60/99.51	56.65/99.50/99.41
FMNIST	50.04/93.32/92.17	51.94/94.28/92.38	51.40/91.81/91.18
CIFAR-10	45.84/78.92/79.37	46.85/89.12/88.75	43.91/77.78/75.34
	bright		
MNIST	55.10/87.55/92.51	52.91/91.47/92.60	52.59/88.82/92.45
FMNIST	50.62/74.13/74.63	51.92/70.95/76.47	50.53/76.10/77.50
CIFAR-10	41.89/64.72/66.14	45.33/74.08/74.31	44.84/64.44/66.70
	contrast		
MNIST	55.11/88.38/89.23	53.12/87.77/92.60	53.49/92.85/92.61
FMNIST	50.46/72.58/74.53	51.90/69.39/78.44	50.46/77.43/80.61
CIFAR-10	41.21/75.04/76.14	47.82/80.82/84.25	42.95/72.89/73.24
	zoom		
MNIST	56.54/99.03/99.18	56.74/89.56/89.53	51.89/98.49/98.33
FMNIST	52.49/93.07/91.24	50.74/92.45/91.17	48.59/91.35/90.55
CIFAR-10	43.86/77.09/75.09	48.32/88.71/86.85	44.13/76.52/73.94
	rotate		
MNIST	53.25/80.67/81.83	53.62/80.99/80.26	53.59/75.89/76.95
FMNIST	46.27/67.46/62.66	47.19/72.17/72.87	46.63/67.74/70.96
CIFAR-10	43.45/58.04/60.64	47.96/71.26/71.13	41.51/63.64/63.60
	shear		
MNIST	49.50/86.74/87.78	49.80/81.54/81.25	55.34/79.53/79.40
FMNIST	46.19/71.87/71.03	47.38/74.96/75.98	46.50/60.98/58.16
CIFAR-10	43.00/60.66/61.84	46.86/65.44/66.10	41.37/60.45/60.17
	Mean		
MNIST	53.60/90.30/**91.65**	53.51/88.49/**89.29**	53.77/89.18/**89.86**
FMNIST	49.08/**78.74**/77.71	49.71/79.03/**81.22**	48.50/77.57/**78.16**
CIFAR-10	43.03/69.08/**69.87**	47.48/78.24/**78.56**	42.78/**69.29**/68.83

Table 8 reports the accuracies of generated test images for the four testing tools on the subject model M and InputReflector (IR). InputReflector achieves the lowest accuracy on RobOT and the

Table 7: Performance of Distribution Analyzer.

AUROC	RobOT	ADAPT	DeepXplore	DLFuzz
MNIST	75.34	**80.03**	70.56	54.18
FMNIST	**77.06**	72.53	61.18	54.54
CIFAR-10	**85.34**	50.02	48.64	48.19
Average	**79.25**	67.53	60.13	52.30

Table 8: Accuracy of generated test cases. IR stands for InputReflector.

Accuracy	RobOT		ADAPT		DeepXplore		DLFuzz	
	M	*IR*	*M*	*IR*	*M*	*IR*	*M*	*IR*
MNIST	2.40	36.15	0	46.65	0	54.00	0	75.85
FMNIST	4.75	17.80	0	43.08	0	54.20	0	53.90
CIFAR-10	0.10	6.30	0	10.70	0	35.05	0	44.50
Average	2.42	20.08	0	33.48	0	47.75	0	58.08

highest accuracy on DLFuzz, indicating that the Reflector in InputReflector has a harder time mapping RobOT-generated test cases to the correct nearby samples in the training data. Thus, the tests generated by RobOT are further from the training data than tests generated by the other tools.

Result 6: *InputReflector holds promise as a technique to evaluate model testing tools. Our results reproduce the findings from previous work [57]: RobOT and ADAPT generate more diverse test instances than DeepXplore and DLFuzz.*

6 RELATED WORK

Runtime trustworthiness checking. Several SE studies consider checking a DNN's trustworthiness in deployment. DISSECTOR by Wang et al. [55] detects inputs that deviate from normal inputs. It trains several sub-models on top of the pre-trained model to validate samples fed into the model. Xiao et al. [61] proposed SelfChecker to monitor DNN outputs using internal layer features. SelfChecker triggers an alarm if the internal layer features of the model are inconsistent with the final prediction and also provides an alternative prediction. SelfChecker achieves better performance than DISSECTOR [61]. Both DISSECTOR and SelfChecker assume that the training and validation datasets come from a distribution similar to that of the inputs that the model will face in deployment. For example, SelfChecker cannot detect out-of-distribution data and provides an alarm constrained by this assumption.

Detection of out-of-distribution data. Several methods [24, 33, 36] have been proposed to detect out-of-distribution data that is completely different from the training data. ODIN [36] and Generalized ODIN [24] are based on trained neural network classifiers that must be powerful enough for the specific dataset. Hsu et al. [24] introduced a decomposed confidence function to avoid using out-of-distribution data to train the classifier, as was done by Liang et al. [36]. The internal representation of DNN models given an input has also been used to detect out-of-distribution data. Mahalanobis [33] takes hidden layers of the DNN model as representation spaces. This work used the distance calculation and input pre-processing to compute the Mahalanobis distance to measure the extent to which an input belongs to the in-distribution in these spaces. But there is a hyperparameter in input pre-processing that must be tuned for each out-of-distribution dataset.

None of these techniques detect deviating data that is between in-distribution and out-of-distribution data, which is the key feature of InputReflector. Hsu et al. [24] showed that Generalized ODIN is empirically superior to previous approaches [22, 33, 36]; we therefore use Generalized ODIN as our baseline to evaluate the performance of the distribution analyzer in InputReflector.

Data augmentation. To improve DNN model generalization, data augmentation techniques have been proposed [9, 14, 48, 66]. In a recent work in this space, Guo et al. [14] augment training data using mutation-based fuzzing. The authors specifically focus on improving the robust generalization of DNNs. Our results indicate that DNN model accuracy on normal testing data will degrade if the training data is augmented with adversarial examples. Moreover, even an augmented training dataset is ultimately finite, limiting model generalization.

Evaluating test case generators. There is a wide range of techniques to automatically generate test cases [11, 15, 42, 43], e.g., EvoSuite and Randoop. To evaluate these techniques, several studies have been conducted to compare the quality of automatically generated and manually written test suites along dimensions such as code coverage [12], mutation score [1], and test smells [17].

In contrast to program test generators, DNN test generators [18, 34, 44, 57] have limited measures to evaluate their quality. Generally, neuron coverage has been shown to be inadequate [19]. To the best of our knowledge, the neuron-based mutation score is one of the few methods designed for this goal [38]. InputReflector, from the perspective of distribution analysis, provides a facility to evaluate DNN test generator. We expect our work in this direction to contribute to the quality assessment of model testing tools.

7 CONCLUSION

Deployed DNNs must contend with inputs that may contain noise or distribution shifts. Even the best-performing DNN model in training may make wrong predictions on such inputs. We presented an input reflection approach to deal with this issue. Input reflection identifies the failure-inducing input and fixes it by reflecting it towards a nearby sample in the training dataset.

We implemented input reflection as part of the InputReflector tool and evaluated it on several datasets and model variants. On three popular image datasets with six transformations InputReflector distinguishes unseen problematic inputs with an average accuracy of 75.53%. Furthermore, by combining InputReflector with the original DNN, we can increase the average model prediction accuracy by 30.78% on the in-distribution and deviating testing dataset. InputReflector can also detect out-of-distribution data that the original DNN cannot handle. We also demonstrated that InputReflector can effectively evaluate the quality of generated test cases by state-of-the-art model testing tools. We hope that our work inspires further research into robust handling of deployment-time inputs to DNNs.

8 ACKNOWLEDGMENTS

This research is supported in part by the Minister of Education, Singapore (T1-251RES1901, T2EP20120-0019, MOET32020-0004), the National Research Foundation, Singapore, and Cyber Security Agency of Singapore under its National Cybersecurity Research and Development Programme (Award No. NRF-NCR_TAU_2021-0002) and A*STAR, CISCO Systems (USA) Pte. Ltd and National University of Singapore under its Cisco-NUS Accelerated Digital Economy Corporate Laboratory (Award I21001E0002). Ivan Beschastnikh also acknowledges the support of the NSERC Discovery Grant RGPIN-2020-05203.

REFERENCES

[1] Alberto Bacchelli, Paolo Ciancarini, and Davide Rossi. 2008. On the effectiveness of manual and automatic unit test generation. In *2008 The Third International Conference on Software Engineering Advances*. IEEE, 252–257.

[2] Jane Bromley, James W Bentz, Léon Bottou, Isabelle Guyon, Yann LeCun, Cliff Moore, Eduard Säckinger, and Roopak Shah. 1993. Signature verification using a "siamese" time delay neural network. *International Journal of Pattern Recognition and Artificial Intelligence* 7, 04 (1993), 669–688.

[3] Hao-Yun Chen, Jhao-Hong Liang, Shih-Chieh Chang, Jia-Yu Pan, Yu-Ting Chen, Wei Wei, and Da-Cheng Juan. 2019. Improving adversarial robustness via guided complement entropy. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 4881–4889.

[4] Lin Chen, Yifei Min, Mingrui Zhang, and Amin Karbasi. 2020. More data can expand the generalization gap between adversarially robust and standard models. In *International Conference on Machine Learning*. PMLR, 1670–1680.

[5] Weihua Chen, Xiaotang Chen, Jianguo Zhang, and Kaiqi Huang. 2017. Beyond triplet loss: a deep quadruplet network for person re-identification. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 403–412.

[6] Davide Chicco. 2021. Siamese neural networks: An overview. *Artificial Neural Networks* (2021), 73–94.

[7] Dan Ciresan, Alessandro Giusti, Luca M Gambardella, and Jürgen Schmidhuber. 2012. Deep Neural Networks Segment Neuronal Membranes in Electron Microscopy Images. In *Advances in neural information processing systems*. 2843–2851.

[8] Francesco Croce and Matthias Hein. 2020. Reliable evaluation of adversarial robustness with an ensemble of diverse parameter-free attacks. In *International conference on machine learning*. PMLR, 2206–2216.

[9] Tommaso Dreossi, Shromona Ghosh, Xiangyu Yue, Kurt Keutzer, Alberto Sangiovanni-Vincentelli, and Sanjit A Seshia. 2018. Counterexample-guided data augmentation. *arXiv preprint arXiv:1805.06962* (2018).

[10] Logan Engstrom, Brandon Tran, Dimitris Tsipras, Ludwig Schmidt, and Aleksander Madry. 2019. Exploring the landscape of spatial robustness. In *International conference on machine learning*. PMLR, 1802–1811.

[11] Gordon Fraser and Andrea Arcuri. 2011. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 416–419.

[12] Gordon Fraser and Andrea Arcuri. 2015. 1600 faults in 100 projects: automatically finding faults while achieving high coverage with evosuite. *Empirical software engineering* 20, 3 (2015), 611–639.

[13] Yaroslav Ganin, Evgeniya Ustinova, Hana Ajakan, Pascal Germain, Hugo Larochelle, François Laviolette, Mario Marchand, and Victor Lempitsky. 2016. Domain-adversarial training of neural networks. *The journal of machine learning research* 17, 1 (2016), 2096–2030.

[14] Xiang Gao, Ripon K Saha, Mukul R Prasad, and Abhik Roychoudhury. 2020. Fuzz testing based data augmentation to improve robustness of deep neural networks. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 1147–1158.

[15] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. 213–223.

[16] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. 2014. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572* (2014).

[17] Giovanni Grano, Fabio Palomba, Dario Di Nucci, Andrea De Lucia, and Harald C Gall. 2019. Scented since the beginning: On the diffuseness of test smells in automatically generated test code. *Journal of Systems and Software* 156 (2019), 312–327.

[18] Jianmin Guo, Yu Jiang, Yue Zhao, Quan Chen, and Jiaguang Sun. 2018. Dlfuzz: Differential fuzzing testing of deep learning systems. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 739–743.

[19] Fabrice Harel-Canada, Lingxiao Wang, Muhammad Ali Gulzar, Quanquan Gu, and Miryung Kim. 2020. Is neuron coverage a meaningful measure for testing deep neural networks?. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 851–862.

[20] Fengxiang He, Shaopeng Fu, Bohan Wang, and Dacheng Tao. 2020. Robustness, Privacy, and Generalization of Adversarial Training. *arXiv preprint arXiv:2012.13573* (2020).

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

[22] Dan Hendrycks and Kevin Gimpel. 2016. A baseline for detecting misclassified and out-of-distribution examples in neural networks. *arXiv preprint arXiv:1610.02136* (2016).

[23] Alexander Hermans, Lucas Beyer, and Bastian Leibe. 2017. In defense of the triplet loss for person re-identification. *arXiv preprint arXiv:1703.07737* (2017).

[24] Yen-Chang Hsu, Yilin Shen, Hongxia Jin, and Zsolt Kira. 2020. Generalized odin: Detecting out-of-distribution image without learning from out-of-distribution data. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 10951–10960.

[25] Andrew Ilyas, Shibani Santurkar, Logan Engstrom, Brandon Tran, and Aleksander Madry. 2019. Adversarial Examples Are Not Bugs, They Are Features. *Advances in neural information processing systems* 32 (2019).

[26] Vigdis By Kampenes, Tore Dybå, Jo E Hannay, and Dag IK Sjøberg. 2007. A systematic review of effect size in software engineering experiments. *Information and Software Technology* 49, 11-12 (2007), 1073–1086.

[27] Jinhan Kim, Robert Feldt, and Shin Yoo. 2019. Guiding Deep Learning System Testing Using Surprise Adequacy. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 1039–1049.

[28] Sungyeon Kim, Dongwon Kim, Minsu Cho, and Suha Kwak. 2020. Proxy anchor loss for deep metric learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 3238–3247.

[29] Lukas Kirschner, Ezekiel Soremekun, and Andreas Zeller. 2020. Debugging inputs. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 75–86.

[30] Gregory Koch, Richard Zemel, Ruslan Salakhutdinov, et al. 2015. Siamese neural networks for one-shot image recognition. In *ICML deep learning workshop*, Vol. 2. Lille.

[31] Alex Krizhevsky, Geoffrey Hinton, et al. 2009. Learning Multiple Layers of Features from Tiny Images. (2009). https://www.cs.toronto.edu/~kriz/cifar.html

[32] Yann LeCun, Corinna Cortes, and CJ Burges. 2010. MNIST Handwritten Digit Database. (2010). http://yann.lecun.com/exdb/mnist/

[33] KIMIN LEE, Kibok Lee, Honglak Lee, and Jinwoo Shin. 2018. A Simple Unified Framework for Detecting Out-of-Distribution Samples and Adversarial Attacks. In *32nd Conference on Neural Information Processing Systems (NIPS)*. Neural Information Processing Systems Foundation.

[34] Seokhyun Lee, Sooyoung Cha, Dain Lee, and Hakjoo Oh. 2020. Effective white-box testing of deep neural networks with adaptive neuron-selection strategy. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 165–176.

[35] Bo Li, Wei Wu, Qiang Wang, Fangyi Zhang, Junliang Xing, and Junjie Yan. 2019. Siamrpn++: Evolution of siamese visual tracking with very deep networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 4282–4291.

[36] Shiyu Liang, Yixuan Li, and R Srikant. 2018. Enhancing The Reliability of Out-of-distribution Image Detection in Neural Networks. In *International Conference on Learning Representations*.

[37] Jonathan Long, Evan Shelhamer, and Trevor Darrell. 2015. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 3431–3440.

[38] Lei Ma, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Felix Juefei-Xu, Chao Xie, Li Li, Yang Liu, Jianjun Zhao, et al. 2018. Deepmutation: Mutation testing of deep learning systems. In *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 100–111.

[39] Ghassan Misherghi and Zhendong Su. 2006. HDD: Hierarchical delta debugging. In *Proceedings of the 28th international conference on Software engineering*. 142–151.

[40] Takeru Miyato, Andrew M Dai, and Ian Goodfellow. 2016. Adversarial training methods for semi-supervised text classification. *arXiv preprint arXiv:1605.07725* (2016).

[41] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y Ng. 2011. Reading digits in natural images with unsupervised feature learning. (2011).

[42] Carlos Pacheco and Michael D Ernst. 2007. Randoop: feedback-directed random testing for Java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. 815–816.

[43] Carlos Pacheco, Shuvendu K Lahiri, Michael D Ernst, and Thomas Ball. 2007. Feedback-directed random test generation. In *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 75–84.

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

[45] Huy Nguyen Anh Pham and Evangelos Triantaphyllou. 2008. Prediction of diabetes by employing a new data mining approach which balances fitting and generalization. In *Computer and Information Science*. Springer, 11–26.

[46] Qi Qian, Lei Shang, Baigui Sun, Juhua Hu, Hao Li, and Rong Jin. 2019. Softtriple loss: Deep metric learning without triplet sampling. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 6450–6458.

[47] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. 2016. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 779–788.

[48] Pouya Samangouei, Maya Kabkab, and Rama Chellappa. 2018. Defense-gan: Protecting classifiers against adversarial attacks using generative models. *arXiv preprint arXiv:1805.06605* (2018).

[49] Florian Schroff, Dmitry Kalenichenko, and James Philbin. 2015. Facenet: A unified embedding for face recognition and clustering. In *Proceedings of the IEEE*

conference on computer vision and pattern recognition. 815–823.

[50] Ali Shafahi, Mahyar Najibi, Amin Ghiasi, Zheng Xu, John Dickerson, Christoph Studer, Larry S Davis, Gavin Taylor, and Tom Goldstein. 2019. Adversarial training for free!. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. 3358–3369.

[51] Ashish Shrivastava, Tomas Pfister, Oncel Tuzel, Joshua Susskind, Wenda Wang, and Russell Webb. 2017. Learning from simulated and unsupervised images through adversarial training. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2107–2116.

[52] Karen Simonyan and Andrew Zisserman. 2014. Very Deep Convolutional Networks for Large-scale Image Recognition. *arXiv:1409.1556* (2014).

[53] Yaniv Taigman, Ming Yang, Marc'Aurelio Ranzato, and Lior Wolf. 2014. Deepface: Closing the gap to human-level performance in face verification. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 1701–1708.

[54] Laurens Van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-SNE. *Journal of machine learning research* 9, 11 (2008).

[55] Huiyan Wang, Jingwei Xu, Chang Xu, Xiaoxing Ma, and Jian Lu. 2020. Dissector: Input validation for deep learning applications by crossing-layer dissection. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 727–738.

[56] Haotao N Wang, Tianlong Chen, Shupeng Gui, TingKuei Hu, Ji Liu, and Zhangyang Wang. 2020. Once-for-All Adversarial Training: In-Situ Tradeoff between Robustness and Accuracy for Free. *Advances in Neural Information Processing Systems* 33 (2020), 7449–7461.

[57] Jingyi Wang, Jialuo Chen, Youcheng Sun, Xingjun Ma, Dongxia Wang, Jun Sun, and Peng Cheng. 2021. RobOT: Robustness-oriented testing for deep learning systems. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 300–311.

[58] Jingyi Wang, Guoliang Dong, Jun Sun, Xinyu Wang, and Peixin Zhang. 2019. Adversarial sample detection for deep neural network through model mutation testing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 1245–1256.

[59] Run Wang, Felix Juefei-Xu, Lei Ma, Xiaofei Xie, Yihao Huang, Jian Wang, and Yang Liu. 2019. Fakespotter: A simple yet robust baseline for spotting ai-synthesized fake faces. *arXiv preprint arXiv:1909.06122* (2019).

[60] Han Xiao, Kashif Rasul, and Roland Vollgraf. 2017. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747* (2017).

[61] Yan Xiao, Ivan Beschastnikh, David S Rosenblum, Changsheng Sun, Sebastian Elbaum, Yun Lin, and Jin Song Dong. 2021. Self-Checking Deep Neural Networks in Deployment. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 372–384.

[62] Cihang Xie, Mingxing Tan, Boqing Gong, Jiang Wang, Alan L Yuille, and Quoc V Le. 2020. Adversarial examples improve image recognition. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 819–828.

[63] Andreas Zeller. 1999. Yesterday, my program worked. Today, it does not. Why? *ACM SIGSOFT Software engineering notes* 24, 6 (1999), 253–267.

[64] Xiyue Zhang, Xiaofei Xie, Lei Ma, Xiaoning Du, Qiang Hu, Yang Liu, Jianjun Zhao, and Meng Sun. 2020. Towards characterizing adversarial defects of deep learning software from the lens of uncertainty. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 739–751.

[65] Zhipeng Zhang and Houwen Peng. 2019. Deeper and wider siamese networks for real-time visual tracking. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 4591–4600.

[66] Zhun Zhong, Liang Zheng, Guoliang Kang, Shaozi Li, and Yi Yang. 2020. Random erasing data augmentation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 13001–13008.