Predictive Mutation Analysis of Test Case Prioritization for Deep Neural Networks

Zhengyuan Wei, Haipeng Wang, Imran Ashraf, and W.K. Chan*
Department of Computer Science, City University of Hong Kong, Hong Kong, China
zywei4-c@my.cityu.edu.hk, haipewang5-c@my.cityu.edu.hk, iashraf3-c@my.cityu.edu.hk, wkchan@cityu.edu.hk
*corresponding author

***Abstract*—Testing deep neural networks requires high-quality test cases, but using new test cases would incur the labor-intensive test case labeling issue in the test oracle problem. Test case prioritization for failure-revealing test cases alleviates the problem. Existing metric-based techniques analyze vector-based prediction outputs. They cannot handle regression models. Existing mutation-based techniques either remain ineffective or incur high computational costs. In this paper, we propose EFFIMAP, an effective and efficient test case prioritization technique with predictive mutation analysis. In the test phase, without performing a comprehensive mutation analysis, EFFIMAP predicts whether model mutants are killed by a test case by the information extracted from the execution trace of the test case. Our experiment shows that EFFIMAP significantly outperforms the previous state-of-the-art technique in both effectiveness and efficiency in the test phase of handling test cases of both classification and regression models. This paper is the first work to show the feasibility of predictive mutation analysis to rank test cases with a higher probability of exposing model prediction failures in the domain of deep neural network testing.**

***Keywords*—Test case prioritization; mutation analysis; testing**

I. INTRODUCTION

Deep neural network (DNN) models find their applications in many application domains, e.g., machine translation [1], image segmentation [2], medical diagnosis [3], and autonomous driving [4]. Previous studies [5]–[9] show DNN models easy to produce unexpected outputs by samples slightly perturbed from the training ones. Infamous examples include casualty incidents and racial discrimination [10]–[12]. A major bottleneck in developing DNN models is the evaluation of their quality, such as correctness, robustness, and fairness. Many testing techniques [5], [7], [8], [13]–[19] have been proposed to address different technical challenges in this bottleneck.

A common problem faced by many testing techniques is the lacking of the oracles (groundtruth labels) of the test cases. Although there are methods (such as metamorphic testing or generic oracle) to identify test cases exposing model failures, they focus on specific kinds of errors (e.g., semantic errors [20] or numerical bugs [21]). Besides, retraining with more labeled samples is the most popular to improve DNN models. In general, producing groundtruth labels of novel samples is human-intensive. Tu et al. [22] reported that manually labeling 22,500+ commits required 175 person-hours. Thus, the labeling-cost problem remains a key challenge in DNN testing.

‡ This research is supported in part by the CityU MF EXT (project no. 9678180).

The problem of test case prioritization (TCP) is to label an affordable number of test cases to reveal the misbehavior of a DNN model (denoted as model $\mathcal{S}$) earliest possible. There are two general categories of effective TCP techniques: metric-based [16], [23], [24] and mutation-based [5], [9], [25].

The former category of techniques [16], [23], [24] measures the likelihood of the misbehavior of $\mathcal{S}$ on each test case t in a test pool. The core idea is to formulate an effective metric to measure such a likelihood. Many metrics applicable to measure classification models have been proposed. DeepGini [16] measures the Gini impurity [26] on the probability vector produced by a model. Dissector [23] proposes a PVscore metric based on multiple probability values in each output vector of the model under test and its submodels. However, the assumption of having multiple values in the output vector of a model makes Dissector inapplicable to handle regression model [27], which only produces a single value as output. LSA/DSA [19] profile the feature maps of $\mathcal{S}$ on processing t and compare them with those from the training samples to compute a metric that measures the degree of surprise. They can handle both classification and regression models, but their comparison process is time-consuming because each test case t needs to compare with the whole training dataset.

The latter category [5], [9], [25] uses mutation analysis, generating mutants of $\mathcal{S}$ and deciding whether a sample kills a mutant. They rank test cases according to their metrics or ranker models in the test phase. The high computational cost of mutation testing is a barrier to the wider adoption of mutation testing in the industry [28]–[30], and mutation testing for DNN models intensifies the problem as the execution of a DNN model is also computationally expensive. The most relevant technique to address this issue in the traditional program testing domain is PMT [31]–[33]. PMT defines a set of program-specific (type- or operator-based) features. It extracts them from the program's peer versions or peer projects for learning and prediction as well as from each mutant of such a peer version/project for deciding whether this mutant is killed or alive for learning. But, it fails to learn if the internal states of the program across versions vary significantly, and fails to predict if the features between the program and its mutant always vary or if all DNN operators and functions are always executed in every inference, which is common across DNN models. In the DNN testing domain, Prima [5] has to execute mutants to generate its model-relevant and mutation-specific

2693-9177/22/$31.00 ©2022 IEEE
DOI 10.1109/QRS57517.2022.00074

features when building its ranker model and prioritizing the test cases in the test phase. Its high computational cost problem in the test phase remains unsolved.

In this paper, we propose EFFIMAP (**Effi**cient **M**utation **A**nalysis for **P**rioritization), a novel, effective and efficient technique to prioritize test cases in the DNN testing domain. EFFIMAP includes three components: *Generator*, *Tracer*, and *Estimator*. We formulate *Generator* with novel strategies to construct both model mutants and sample mutants. It incrementally builds a set of high-quality model mutants with increasingly higher failure-revealing effects, which collectively makes the mutants achieve higher killing coverage on incorrect test cases in the validation dataset. It also formulates a novel autoencoder strategy to generate sample mutants to address the inadequate data issue and guides the autoencoder model's training process to kill increasingly more mutants with diverse sample mutants. The execution of $\mathcal{S}$ with each sample (or sample mutant) as input is profiled by *Tracer* to generate a novel distribution-oriented execution trace, capturing the distribution-relevant log data on how the sample (or the sample mutant) is progressively transformed across layers into the model output, making the trace sensitive to the distribution shifts between the model under test and its model mutants. We then build a machine-learning *Estimator* with a single execution trace as an input, which learns the differences in the outputs between model mutants and $\mathcal{S}$ on the corresponding sample for predictive mutation analysis for ranking. In the test phase, the execution trace of a novel sample will be input to *Estimator* to predict the extent of killing these mutants and compute an enhanced mutation score according to a novel metric. EFFIMAP then prioritizes these novel samples by the enhanced mutation score for labeling effort reduction. In this way, the expensive mutant execution process appearing in the test phase in the existing techniques is replaced by our predictive analyzer, *Estimator*, which requires *no* mutant execution in the test phase.

We evaluate EFFIMAP on four models on the challenging task of prioritizing novel samples, unlike adversarial examples with clear clues of incorrectness generated from original samples. The results show EFFIMAP outperforms Prima in the effectiveness of test case prioritization with a computational cost reduction of 95%–98% in the test phase. We also show the first work in predictive mutation-adequate test suite construction in the DNN testing domain guided by our predictive ranking analysis.

The main contribution of this paper is twofold: (1) The paper presents *the first work* of predictive mutation analysis *without* mutant execution in the test phase in the DNN testing domain. (2) It shows the novelty, feasibility, effectiveness, and efficiency of EFFIMAP for test case prioritization.

In the rest of the paper, Sections II to IV present the preliminaries and our technique with an evaluation, respectively. We will review the closely related work in Section V. Section VI concludes this paper.

II. PRELIMINARIES

A. Classification and Regression DNN Models

A DNN model $\mathcal{S}$ contains a set of layers with unique indexes (from 1 to g). The input to first layer f_1 is a sample t. The last layer f_g produces an output of $\mathcal{S}$, donated as $\mathcal{S}(t)$. The input of layer $f_{i\neq 1}$ is the output of its preceding layer. Each value in output *feature map* of layer $f_{i\neq g}$ is a *neuron*.

The kinds of model outputs vary by model type and the *prediction error* δ_t of $\mathcal{S}(t)$ is computed with its ground truth of t, donated by l_t. The *correctness* of model $\mathcal{S}$ with sample t as input is defined based on the prediction error. We also call an incorrect prediction of $\mathcal{S}$ as a *failure*.

For a classification model, $\mathcal{S}(t)$ is a probability vector where the predicted class $\mathcal{S}_c(t) = \text{argmax}_c\, \mathcal{S}(t)[c]$. If $\mathcal{S}_c(t) = l_t$, then δ_t is 0, otherwise 1, The output $\mathcal{S}(t)$ is deemed *correct* if $\delta_t = 0$, otherwise *incorrect*. For a regression model, $\mathcal{S}(t)$ is a value, and δ_t is $|\mathcal{S}(t) - l_t|$. The output $\mathcal{S}(t)$ is deemed *correct* if $\delta_t < \epsilon$, otherwise *incorrect*, where ϵ is a given instance-specific bound based on $\mathcal{S}$'s performance [34], [35].

When assessing $\mathcal{S}$ on a dataset T, we divide T into two subsets, one containing all correct samples ($\{t \in T \mid \mathcal{S}(t)$ is correct$\}$) and another containing all incorrect samples ($\{t \in T \mid \mathcal{S}(t)$ is incorrect$\}$), denoted by T^+ and $T^\times$, respectively. The *performance* of model $\mathcal{S}$ on a dataset T is measured by the *accuracy* for a classification model, defined as the ratio of correct samples (i.e., $|T^+|/|T|$), and the *mean square error (MSE)* for a regression model, i.e., $\frac{1}{|T|}\sum_{t\in T}(\delta_t)^2$.

B. Terminologies in Mutation Analysis

Mutating a model $\mathcal{S}$ and a sample t produce a *model mutant* ($\overline{\mathcal{S}}$) and a sample mutant (t'), respectively [5], [25].

We define a Boolean predicate $\Delta(\mathcal{S}, \overline{\mathcal{S}}, t)$ to give 1 (true) if (1) $\mathcal{S}_c(t) = \overline{\mathcal{S}}_c(t)$ for $\mathcal{S}$ as a classification model or (2) $|\mathcal{S}(t) - \overline{\mathcal{S}}(t)| > \epsilon$ for $\mathcal{S}$ as a regression model, otherwise 0 (false). If $\Delta(\mathcal{S}, \overline{\mathcal{S}}, t)$ is true, we say that $\overline{\mathcal{S}}$ **covers** t and, interchangeably, t **kills** $\overline{\mathcal{S}}$. The sample t is called a mutant-killing sample. Given a set of mutants $\mathbb{S}$ used in mutation analysis, the ratio of mutants killed by a sample t is called the *mutation score* (achieved by t on $\mathbb{S}$), i.e., $|\{\overline{\mathcal{S}} \in \mathbb{S} \mid \Delta(\mathcal{S}, \overline{\mathcal{S}}, t) = 1\}|/|\mathbb{S}|$. Given a dataset T, the set of samples each killing the mutant $\overline{\mathcal{S}}$ is referred to *mutant coverage* (achieved by $\overline{\mathcal{S}}$ on T), and the largest subset of $\mathbb{S}$ killed by T is called the *mutation coverage* (achieved by $\mathbb{S}$ on T). A **test case** is a sample t in the test phase, and the set of selected test cases during testing constructs a test suite. A test suite $\mathcal{X}$ (where $\mathcal{X} \subseteq T$) is *mutation-adequate* with respect to $\mathbb{S}$ if the mutation coverage of T and $\mathcal{X}$ are the same.

C. Variational Autoencoder

Variational autoencoder (VAE) [36] consists of a pair of encoder and decoder neural network components with a sampler in between. When producing a variant of t, the encoder encodes t to a feature map in a latent space; the sampler generates additional inputs from a normal distribution $N(0, 1)$ followed by adding them to the feature map; and the decoder decodes the perturbed feature map to produce a variant of t.

D. Dissector, PMT, and Prima

Dissector [23] asserts a model $\mathcal{S}$ predicts a sample t with higher confidence of its predicted class than a submodel of $\mathcal{S}$. It generates a sequence of submodels from $\mathcal{S}$. A submodel is donated as $\overline{\mathcal{S}}$ with probability vector as output. Let the highest and second highest probability values of $\overline{\mathcal{S}}(t)$ and the probability value of $\overline{\mathcal{S}}_c(t)$ be a, b, and c, respectively. Dissector computes a SVscore defined as $c/(b+c)$ if $a = c$, otherwise $c/(a+c)$. It then computes a PVscore metric by the weighted normalization of SVscores from these submodels. In the test phase, Dissector executes the test cases over the model $\mathcal{S}$ and the submodels and then ranks the test cases by the PVscore.

We cast PMT into DL model testing by treating a model as if it is a program. PMT [31] and Prima [5] each produce a set of m model mutants (denoted by $\mathbb{S}_1$) and the sets of n sample mutants for each sample t (denoted by $\mathbb{T}_t$).

In the ranker building phase (aka training phase), they execute each model in $\{\mathcal{S}\} \cup \mathbb{S}_1$ over all samples in $\{t\} \cup \mathbb{T}_t$ for each t in T_v and collect the respective model output, where T_v is the validation dataset of $\mathcal{S}$. PMT utilizes a cross-version or cross-project model of $\mathcal{S}$. It defines 15 code-based features for t and uses a predicate indicating a mutant is killed as the target for learning. However, these code-based features for each t very little in the DNN test domain, harming the effectiveness of PMT on the TCP problem. Prima computes six kinds of mutation-specific features for t, such as the number of killed mutants and the mean difference in predicted probability between each killed mutant and $\mathcal{S}$. It collects the prediction error of $\mathcal{S}(t)$ (where $t \in T_v$) as the targets for learning. Both of them apply an existing machine learning algorithm [37] to produce a ranker $\mathcal{R}$. The number of samples in the training dataset for $\mathcal{R}$ is $|T_v|$, which is much smaller than the training dataset for $\mathcal{S}$, lowering their effectiveness.

In the test phase, they produce the features of each novel sample t. Prima uses $\mathcal{R}$ to estimate the prediction error of $\mathcal{S}(t)$, which requires the executions of both model and sample mutants. PMT extracts features from $\mathcal{S}$ to predict the predicates and then compute the mutation score. They prioritize test cases in descending order of the final output. Prima must conduct a total of $n(m+1)|T_e|$ executions (for m model mutants and n sample mutants for each sample in the test pool T_e for ranking) with feature synthesis for each $t \in T_e$, which makes it slow in the test phase.

III. Our Technique

This section presents EffiMAP, a novel predictive mutation analysis technique for test case prioritization. EffiMAP has three components: *Generator*, *Tracer*, and *Estimator*. Fig. 1 depicts its main workflow.

A. Overview

Generator produces model mutants and sample mutants for predictive mutation analysis. Apart from addressing the challenge of inadequate data issues in predictive-based mutation techniques, it formulates a novel technique to generate mutants strongly correlated to the misbehavior of $\mathcal{S}$.

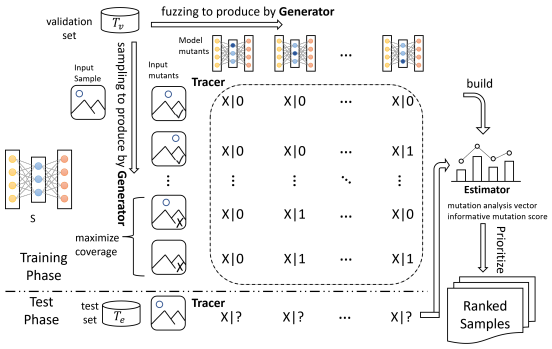


Figure 1: An overview of EffiMAP.

Tracer captures mutation-independent execution features on the inference, which is sensitive to the correctness of the model $\mathcal{S}$, thereby enabling the predictive mutation analysis on individual samples. *Tracer* profiles each layer of $\mathcal{S}$ and produces a sequence of values, called *execution trace*.

Estimator bridges the gap between the execution features and the correctness of $\mathcal{S}$. It firstly maps the execution trace to the mutation results of the set of model mutants $\mathbb{S}$ and then applies an enhanced mutation analysis to compute a novel mutation score as the metric for prioritization.

B. Target Orientated Mutant Generation

1) Model Mutation: Generator fuzzily generates a set of model mutants $\mathbb{S}$ with two objectives to increase the mutant diversity to distinguish different failing test cases. Such diversity is measured by the mutation coverage incrementally on incorrect samples. Moreover, mutants with higher failure-revealing energy are selected.

We define the failure-revealing energy (energy for short) of a model mutant $\overline{\mathcal{S}}$ on the validation dataset T as follows:

$$\begin{aligned}\text{Energy}(\mathcal{S}, \overline{\mathcal{S}}, T_v) = |\{t \in T_v^{\times} \mid \Delta(\mathcal{S}, \overline{\mathcal{S}}, t) = 1\}| \\ - |\{t \in T_v^{+} \mid \Delta(\mathcal{S}, \overline{\mathcal{S}}, t) = 1\}|\end{aligned} \quad (1)$$

where $T_v^{\times}$ and T_v^{+} contain the incorrect samples and correct samples divided by the correctness of $\mathcal{S}$ on T_v.

Similarly, to generate mutants with distinguishing abilities, we have reviewed existing works on their cause-and-effect chains to the performance of $\mathcal{S}$. Some works [25], [38] contain random mutation operators to construct a better test suite but lack fault-revealing abilities, which we do not include in the work. Some works present such abilities. He et al. [39] show different filters in convolutional layers contributing unequally to the performance and ReAct [40] shows the rectification of activation effective in detecting anomalies due to mismatched normalization.

Generator constructs three mutation operators:

- *Prune Convolution*: disable one filter in a convolutional layer by setting the corresponding output channel to a zero matrix.
- *Revert Normalization*: substitute the values of an output channel of a normalization layer back to the values of the corresponding input channel of the same layer.

Algorithm 1: Generate model mutants

Input : DNN model $\mathcal{S}$, Validation dataset T_v, Mutation operators O, Number of mutants m, Performance margin η, Fuzzing resource γ

Output: A set of model mutants $\mathbb{S}$

1 $p \leftarrow \text{Evaluate}(\mathcal{S}, T_v)$
2 $T_v^{\times} \leftarrow \text{FindIncorrect}(\mathcal{S}, T_v)$
3 $C \leftarrow \text{FindKillers}(\mathcal{S}, \mathcal{S}, T_v^{\times})$
4 $\mathbb{S} \leftarrow \emptyset, e_{\propto} \leftarrow \infty, \gamma_0 \leftarrow \text{GetTime}()$
5 **while** *GetElapseTime*$(\gamma_0) < \gamma$ **do**
6 $\quad \overline{\mathcal{S}} \leftarrow \text{GenerateMutant}(O, \mathcal{S})$
7 $\quad p' \leftarrow \text{Evaluate}(\overline{\mathcal{S}}, T_v)$
8 $\quad$ **if** $\overline{\mathcal{S}} \notin \mathbb{S}$ *and* $\frac{|p'-p|}{p} \leq \eta$ **then**
9 $\quad\quad C' \leftarrow \text{FindKillers}(\mathcal{S}, \overline{\mathcal{S}}, T_v^{\times})$
10 $\quad\quad c \leftarrow \text{NewCoverage}(C, C')$
11 $\quad\quad e \leftarrow \text{Energy}(\mathcal{S}, \overline{\mathcal{S}}, T_v)$
12 $\quad\quad$ **if** $|\mathbb{S}| < m$ **then**
13 $\quad\quad\quad \mathbb{S} \leftarrow \mathbb{S} \cup \{(\overline{\mathcal{S}}, c, e)\}$
14 $\quad\quad\quad C \leftarrow C \vee C', e_{\propto} \leftarrow \min(e_{\propto}, e)$
15 $\quad\quad\quad$ **continue**
16 $\quad\quad$ **end if**
17 $\quad\quad$ **if** $c = 1$ **then**
18 $\quad\quad\quad \mathbb{S} \leftarrow \mathbb{S} \setminus \{$ *first* $s \in \mathbb{S} \mid c_s = 0 \}$
19 $\quad\quad\quad \mathbb{S} \leftarrow \mathbb{S} \cup \{(\overline{\mathcal{S}}, c, e)\}$
20 $\quad\quad\quad C \leftarrow C \vee C'$
21 $\quad\quad$ **else if** $e > e_{\propto}$ **then**
22 $\quad\quad\quad \mathbb{S}' \leftarrow \{$ *first* $s \in \mathbb{S} \mid e_s < e \wedge c_s \leq c \}$
23 $\quad\quad\quad$ `// skip below if` $\mathbb{S}'$ `is empty`
24 $\quad\quad\quad \mathbb{S} \leftarrow \mathbb{S} \setminus \mathbb{S}' \cup \{(\overline{\mathcal{S}}, c, e)\}$
25 $\quad\quad\quad e_{\propto} \leftarrow \min(\{e \mid \forall e \in s \in \mathbb{S}\})$
26 $\quad\quad$ **end if**
27 $\quad$ **end if**
28 **end while**

- *Rectify Activation*: cap the values of activated neurons in a selected activation layer to $\beta\%$ of the highest value in the same output feature map.

Alg. 1 shows the algorithm of *Generator* to generate model mutants. It accepts a model $\mathcal{S}$, a validation set T_v, a set of mutation operators O, the number of required mutants to generate m, a parameter of performance margin η, and a fuzzing resource γ as inputs.

At line 1, it evaluates the performance p of the model $\mathcal{S}$ as the baseline on the validation dataset T_v via the function *Evaluate()*. It separates the set of incorrect samples $T_v^{\times}$ from T_v (line 2) via the function *FindIncorrect()*. The function *FindKillers*(a, b, c) returns a predicate vector containing the value of $\Delta(a, b, t)$ for each sample t in c. (In line 3, the inputs of both a and b are $\mathcal{S}$ so that the returned predicate vector C is a zero vector.) Next, it initializes the variables representing the set of model mutants $\mathbb{S}$, the minimum energy $e_{\propto}$, and the current fuzzing timestamp γ_0 to an empty set, the infinity, and the current time, respectively (line 4).

From lines 5–27, it goes into a fuzzing loop to generate model mutants iteratively until the fuzzing resource γ is exhausted. The fuzzing process is limited to a bounded execution time, and if the duration time measured via the function *GetElapseTime()* is larger than γ, the process terminates. In each iteration, it randomly applies one mutation operator in O to mutate $\mathcal{S}$, gets a mutant $\overline{\mathcal{S}}$ through the function *GenerateMutant()* at line 6, and measures the performance p' of the mutant $\overline{\mathcal{S}}$ (line 7) over T_v. It checks whether the mutant is new ($\overline{\mathcal{S}} \notin \mathbb{M}$) and whether it behaves similarly to $\mathcal{S}$ at line 8. The mutant is deemed *similar* if its performance is close to that of $\mathcal{S}$ within $\eta\%$. The algorithm then attains the predicate vector C' of the pair of $\mathcal{S}$ and $\overline{\mathcal{S}}$ on $T_v^{\times}$ via *FindKillers()*, measures whether there is new coverage of C' over C via *NewCoverage()*, and computes the energy of $\overline{\mathcal{S}}$ according to Eq. 1 at lines 9–11.

The algorithm has two phases: collect a new mutant or update an existing one. It stays in the first phase if the number of mutants in $\mathbb{S}$ is less than m (line 12). It puts the new mutant into $\mathbb{S}$ annotated with its new coverage indicator c and energy e (line 13). The predicate vector C and the minimum energy $e_{\propto}$ are maintained (line 14) by performing the element-wise logical-or operation with C and the *min()* function, respectively. The algorithm begins the second phase if $\mathbb{S}$ contains m mutants. The objective is to find "more interesting" mutants by checking (1) whether the new mutant can cover new elements in $T_v^{\times}$ (i.e., whether $c = 1$ at line 17), (2) whether the energy of the new mutant is more powerful than the minimum energy (i.e., whether $e > e_{\propto}$ at line 21). If (1) holds, the algorithm drops the first element in $\mathbb{S}$ that its new coverage indicator is 0 and then adds the annotated new mutant to $\mathbb{S}$, followed by maintenance of predicate vector C (lines 18–20). Otherwise, if (2) holds, it drops the first element in $\mathbb{S}$ that its energy is less than that of $\overline{\mathcal{S}}$ and its coverage is not better than that of $\overline{\mathcal{S}}$, followed by maintenance of the minimum energy (lines 22–25). However, if no such element is found, lines 24–25 will be skipped.

2) Input Mutation: Typically a learning-based method requires enormous data to achieve an accurate approximation, and the validation set is only a relatively small set of samples. Using the original validation dataset of one model as the training dataset for another model would easily lead the latter model to suffer from a severe overfitting problem due to the inadequate data issue. EFFIMAP formulates a novel generative strategy to enrich the validation dataset to alleviate this problem. *Generator* trains an adopted variational autoencoder $\mathcal{A}$ to minimize the reconstruction loss of $\mathcal{A}$, and, further, regularizes $\mathcal{A}$ to generate valuable sample mutants that could reveal the model failures during its training process.

Alg. 2 shows the algorithm to build a variational autoencoder for generating sample mutants. It accepts the model $\mathcal{S}$, a validation dataset T_v, an autoencoder model $\mathcal{A}$, and the maximum training epochs ρ_{max} as inputs. Like Alg. 1, it splits the set $T_v^{\times}$ with all incorrect samples from the validation set T_v (line 1) and prepares a zero vector indicating distinct correctness of $\mathcal{S}$ for each t in $T_v^{\times}$ via *FindDistinct()* (line

Algorithm 2: Build autoencoder for input mutation

Input : DNN model $\mathcal{S}$, Validation dataset T_v, Autoencoder $\mathcal{A}$, Training epochs ρ_{max}
Output: A trained autoencoder $\hat{\mathcal{A}}$

1 $T_v^{\times} \leftarrow$ FindIncorrect$(\mathcal{S}, T_v)$
2 $D \leftarrow$ FindDistinct$(\mathcal{S}, T_v^{\times}, T_v^{\times})$
3 $\hat{\mathcal{A}} \leftarrow \mathcal{A}(), \rho \leftarrow 0$
4 **while** $\rho < \rho_{max}$ **do**
5 $\quad T_{\diamond}^{\times} \leftarrow$ AugmentData$(T_v^{\times}, \neg D)$
6 $\quad \hat{\mathcal{A}}' \leftarrow$ Train$(\hat{\mathcal{A}}, T_{\diamond}^{\times})$
7 $\quad T_v^{*} \leftarrow$ Sample$(\hat{\mathcal{A}}', T_v^{\times})$
8 $\quad D' \leftarrow$ FindDistinct$(\mathcal{S}, T_v^{\times}, T_v^{*})$
9 $\quad$ **if** $D'.sum() > D.sum()$ **then**
10 $\quad\quad D \leftarrow D'$
11 $\quad\quad \hat{\mathcal{A}} \leftarrow \hat{\mathcal{A}}'$
12 $\quad$ **end if**
13 $\quad \rho \leftarrow \rho + 1$
14 **end while**

2), followed by initialization of the trained autoencoder $\hat{\mathcal{A}}$ to random state and the current trained epoch to 0 (line 3). The funciton *FindDistinct*(a, b, c) returns a vector containing prediction difference between $a(t_1)$ and $a(t_2)$ for each sample pair (t_1, t_2) in (b, c). Note that in line 3, the inputs of both b and c are $T_v^{\times}$ so that the returned vector D is a zero vector. The algorithm iteratively trains $\hat{\mathcal{A}}$ (lines 4–14) with ρ_{max} epochs.

Wang et al. [9] show anomaly samples are more likely to cross the decision boundaries if $\mathcal{S}$ is slightly mutated. Our insight is that a slight mutation on a sample to produce an anomaly sample also has a higher chance of changing the prediction of $\mathcal{S}$ so that $\mathcal{S}$ can distinguish them. Therefore, in line 5, the algorithm perturbs incorrect samples of $T_v^{\times}$ if they do not have such changing effect (indicated by $\neg D$) by data augmentation methods (e.g., Image transformation [41], mixup [42], and adversarial example generator [43]) via *AugmentData()*. The function *AugmentData()* perturbs t in $T_v^{\times}$ only if its indicator in $\neg D$ is true but does not mutate other samples, producing an enhanced $T_{\diamond}^{\times}$ for training.

The algorithm then trains the autoencoder $\mathcal{A}$ with $T_{\diamond}^{\times}$ and produces a candidate model $\hat{\mathcal{A}}'$ at line 6. It uses $\hat{\mathcal{A}}'$ to sample one mutant from each t in $T_v^{\times}$ via *Sample()* and produces T_v^{*} (line 7). It then calls *FindDistinct()* with $T_v^{\times}$ and T_v^{*} as inputs to attain the distinctness vector D' at line 8. It validates whether the generated sample mutants have higher changing effects (line 9), and if so, it keeps this optimized model state to $\hat{\mathcal{A}}$ and updates D.

Generator uses the trained $\hat{\mathcal{A}}$ to generate n sample mutants for each sample t in T_v, constructing $T_v^{\star}$ with T_v included for subsequent steps.

C. Execution Trace as Features

We are inspired by a fundamental concept in deep learning: A DNN model encodes a distribution of samples (aka. input distribution) and transforms the input distribution into an output distribution through a series of intermediate transformations. *Tracer* profiles the distribution-relevant statistics from the feature maps and the descriptive statistics from the output of $\mathcal{S}$ as log data. As mutation analysis is already computationally expensive, *Tracer* is designed only to profile the computationally-efficient log data.

Tracer records the following kinds of log data as an *execution trace* to capture the processing in the forward inference on predicting a sample.

- *Log on Distribution*: The mean and the variance of the feature map for each layer of the model.
- *Log on Heatmap*: The proportion of the neurons produced by each activation layer, where neurons have been deactivated before and after the activation layer.
- *Task Specific Log*: The value of the Shannon entropy [44], [45] on the output of the model.

For the first kind of log, our insight is that the distribution shift on the feature maps of an incorrect sample will sooner or later occur when propagating across layers for feature extraction; otherwise, the extracted features without showing any distribution difference are less likely to be predicted through the output layer to produce an incorrect output. The second kind of log intends to capture the non-linearity property of DNN, from which the performance of a DNN model heavily depends on neuron activation. The third kind of log collects descriptive statistics on the model output.

Tracer generates the execution traces of all the samples in the dataset $T_v^{\star}$ produced by *Generator* for the next step.

D. Predictive Mutation Analysis for Ranking

Estimator receives the outputs from *Generator* and *Tracer*, both of which have the design to produce a good set of outputs for the predictive mutation analysis. It collects the mutation results of samples and sample mutants between the model and model mutants and builds a predictive estimator to produce the predicates of killings.

Rather than learning from the differences between a pair of execution features on how to kill a model mutant, we aim to significantly reduce the high mutant execution cost in the subsequent test phase. Thus, the challenge is learning from the execution features to predict whether the sample kills each model mutant. Our insight is that given a fixed model mutant $\overline{\mathcal{S}}$ which mutates a specific position of $\mathcal{S}$, if a sample t can kill $\overline{\mathcal{S}}$, the execution features (from the feature maps of the mutated layer) of $\mathcal{S}$ must be altered, resulting in the predicate of killing to be true. In mutation analysis with a set of fixed model mutants $\mathbb{S}$, the execution trace captures the execution features of $\mathcal{S}$, and the mutation results are the set of predicates of killings of $\mathbb{S}$. *Estimator* adapts a learning-based algorithm to connect these two sets of distributions to encode the differences in execution features between $\mathcal{S}$ and $\overline{\mathcal{S}}$. In this paper, we demonstrate the feasibility.

To ease readers follow, we recap the artifacts from the above: *Generator* fuzzily generates a set of m model mutants $\mathbb{S}$ and a dataset $T_v^{\star}$ containing n sample mutants of each t of T_v and all samples of T_v for mutation analysis. *Tracer*

profiles the execution trace of all samples in the dataset $T_v^\star$ by inferencing the model $\mathcal{S}$ with each sample t.

We note the execution trace of $\mathcal{S}(t_j)$ as ϕ_j for each sample t_j in $T_v^\star$, and the set of all these execution traces as Φ. We assume the set of model mutants $\mathbb{S}$ is an ordered set. For each model mutant $\overline{\mathcal{S}}_i$ in $\mathbb{S}$, *Estimator* measures whether each sample t_j can kill the model mutant $\overline{\mathcal{S}}_i$, indicating by the predicate $\Delta(S, \overline{\mathcal{S}}_i, t_j)$. We note the predicate vector of t_j to be π_j, where $\pi_j[i] = \Delta(S, \overline{\mathcal{S}}_i, t_j)$, and the set of predicate vectors of all samples in $T_v^\star$ is donated by Φ.

Estimator learns to map Φ to Π. It adopts the XGBoost framework [37] to construct an estimator model, denoted as $\mathcal{E}$, and utilizes the scikit-learn tools [46] to fit $\mathcal{E}$ to the training data (i.e., from Φ to Π). We choose a decision-tree model because the number of log data in an execution trace is not large. The decision-tree algorithm is scalable and more effective under this condition [47], and the prediction result is human-explainable. The trained estimator is notated as $\hat{\mathcal{E}}$. Note that the above-mentioned procedure to construct $\hat{\mathcal{E}}$ requires only being conducted once in an offline manner.

In the test phase, for each test case t in the test pool, *Tracer* profiles the execution of $\mathcal{S}(t)$ and produces the execution trace ϕ_t. The execution trace ϕ_t is input to the estimator $\hat{\mathcal{E}}$ to attain the predicate vector π_t representing the mutation result. Since the produced π_t is also a probability vector representing the confidence with respect to whether each model mutant is killed, it can also yield the Shannon entropy [44], [45] from the underlying probability vector π_t. Our insight is that in normal cases for a model under test, its mutants should not easily be all killed, even though there may be a surprise that one/few mutant(s) is killed; thus, the higher entropy implies a higher probability of misbehavior of a model (more informative). *Estimator* computes the Shannon entropy of π_t, donated as $sh(\pi_t)$, and binarizes π_t with 0.5 as the threshold to attain the predicate vector and computes the mutation score, donated as $ms(\pi_t)$. Then *Estimator* computes the *informative mutation score* defined as:

$$\text{ims}(\pi_t) = \text{sh}(\pi_t) \cdot \text{ms}(\pi_t) \tag{2}$$

where $\text{sh}(\pi_t)$ and $\text{ms}(\pi_t)$ are defined above. The informative mutation score is the metric of *Estimator* to prioritize the test cases in the test pool.

Unlike the existing work of Prima, EFFIMAP executes no (sample or model) mutants in the test phase, significantly reducing the computational cost of applying a comprehensive mutation analysis and advancing state of the art.

IV. EVALUATION

A. Research Questions

We aim to answer the following three research questions,

- RQ1: Is EFFIMAP effective in prioritizing novel test cases compared with existing state-of-the-art techniques?
- RQ2: Is EFFIMAP effective in reducing the computational cost compared to the state-of-the-art mutation-based technique?
- RQ3: Is the test suite construction guided by EFFIMAP effective in exposing model prediction failures?

B. Experimental Setup

1) Implementation and Environment: We implement all techniques in Python v3.8, XGBoost v1.5.2 [37], Scikit-learn v0.24.2 [46], and Pytorch v1.8.1 [48]. We run all experiments on a Ubuntu 20.04 server equipped with a 48-core 3.0GHz Xeon CPU, 256GB RAM, and a 2080Ti GPU. All codes of the experiments are available at GitHub [49], which will fetch the public datasets automatically.

We compare EFFIMAP (EM) with Dissector (DS) and Prima (PM). DS and PM are the current state-of-the-art techniques in test case prioritization using the metric-based and mutation-based approaches, respectively. We download the code [50] of Dissector [23] and the code [51] of Prima [5]. These two tools do not include all the code to make them executable (i.e., no sub-models generation in [50] and no building learning-to-rank models in [51], which are stated in their code repositories' README pages). We implement the missing parts according to their original papers [5], [23], where we generate the sub-models of the model under test for DS and the learning-to-rank models for PM. The training settings for the sub-models are reused from their original models. The hyperparameters for training PM's learning-to-rank models are set according to Section IV.C of the original paper [5].

We set the number of model mutants m to 100 and the number of input mutants n to 200, which are adopted from Prima [5] for a fair comparison. The perturbation margin $\eta\%$ for Alg. 2 is set to 5%, following [35], and the fuzzing resource γ is set to 10 minutes for a reasonable time. The parameter $\beta\%$ of the Rectify mutation operator is set to 90% (best parameter in [40]). We download the autoencoder $\mathcal{A}$ from Github[1] for Alg. 2, with the training epochs $\rho_{max} = 100$. We adopt the Gaussian Noise operation [52], which is also used in existing testing and debugging works [9], [25], to implement the function *AugmentData()* with Gaussian distribution of $N(0, 1)$. For constructing the estimator $\mathcal{E}$, we set *n_estimators* = 1000 and *max_delta_step* = 5 when fitting the estimator of EM to its training data.

2) Model and Dataset: We evaluate EFFIMAP on both classification and regression tasks. We select subjects (models and datasets) that have been widely used in previous studies of DNN testing. As such, four models with their datasets are selected. They include (1) ResNet32 [53] pretrained on CIFAR100 [54], (2) ResNet18 [55] pretrained on TinyImageNet [56], (3) MLP-based model M_3 [57] built for MNIST [58], and (4) CNN-based model M_4 [57] built for SVHN [59]. The former two models are pretrained on corresponding datasets, while for the latter two datasets, we can not find any pretrained models with our best effort. To evaluate PMT, we also introduce ResNet56 and VGG13 both from [53] to serve as the cross-version and cross-project of ResNet32, respectively. We randomly split each downloaded

[1] https://github.com/AntixK/PyTorch-VAE/blob/master/models/vanilla_vae.py

TABLE I: Descriptive Statistics of Subjects

ID	Task	Dataset	Model	#Data	#Classes	Perf
1	C	CIFAR100	ResNet32	50K/10K	100	70.16
2	C	TinyImageNet	ResNet18	100K/9K	200	69.67
3	R	MNIST	MLP-M_3	50K/10K	10	0.85
4	R	SVHN	CNN-M_4	73K/23K	10	1.4

* #Data = original numbers of training/test samples; Perf = performance, accuracy for classification (C), MSE for regression (R).

test dataset on 1:9 to construct the validation dataset T_v and **the test pool** T_e. Note that test cases in T_e are all novel samples never trained by the downloaded model, and the prioritization is conducted on these novel samples.

To obtain the well-trained models for M_3 and M_4 on regression tasks, we apply the code from [55], [60] on M_3 and M_4 to produce regression value. The groundtruths of their corresponding datasets are integers from 0 to 9 for regression. We have evaluated the MSE performance of the trained models on their test datasets, and both M_3 and M_4 match the reported high performance in the tutorials [55], [57], [60]. The instance-specific bound ϵ to determine the correctness for M_3 and M_4 are set to $\eta\%$ of the square root of their MSE performance. Table I shows the descriptive statistics of each subject (model + dataset), all achieving high performance.

3) Metrics: We adopt the formula from [5] to measure the prioritization effectiveness using the metric *Ratio of Area Under Curve (**RAUC**)*: Given a test pool T_e and a model under test $\mathcal{S}$, we note L_{rank} to an ordered list containing all the test cases in T_e, which is sorted by a technique. Suppose $\mathcal{O}$ is the test oracle of $\mathcal{S}$ which can measure the correctness of $\mathcal{S}$ on each test case t in T_e. $\mathcal{O}(t) = 0$ if $\mathcal{S}(t)$ is correct, otherwise $\mathcal{O}(t) = 1$. With the assistance of the test oracle $\mathcal{O}$, we can sort all the test cases of T_e into another ordered list L_{oracle} in descending order by the correctness (i.e., fulfilling the condition: $\forall i < j, L_{oracle}[i] \geq L_{oracle}[j]$).

Let $cum(L, k)$ be a function to compute the accumulated sum of the values of the top-k elements in a list L, defined as $cum(L, k) = \sum_{i=1}^{k} L[i]$. The *area under curve (AUC)* for the top-k elements in a list L where $k \leq |L|$, denoted by $AUC(L, k)$, is defined as $AUC(L, k) = \sum_{i=1}^{k} cum(L, i)$. The RAUC for a given ranked list L_{rank} produced by a technique on T_e and measured for the top-k elements is defined as $RAUC(L_{rank}, k) = AUC(L_{rank}, k)/AUC(L_{oracle}, k)$.

Like [5], we measure the RAUCs for the top-k elements of L_{rank}, where $k \in \{100, 200, 300, 500\}$, as well as the top-$k\%$ of the list, where $k\%$ is equivalent to $\alpha|L|$ number of elements, where $\alpha \in \{10\%, 20\%, 30\%, 50\%, 100\%\}$.

We profile the **elapsed time** starting from each stage of techniques until generating the ranked lists over the whole test pool T_e, using the *time()* function provided in Python, to measure the efficiency.

Suppose $\mathbb{X}_{adq}$ is a set of mutation-adequate test suites from T_e. Let $\mathbb{X}^{\times}_{adq} = \{\mathcal{X} \in \mathbb{X}_{adq} \mid \exists t \in \mathcal{X}, t \text{ is incorrect}\}$ be its largest subset, in which each test suite contains at least one incorrect test case. We measure the ratio $|\mathbb{X}^{\times}_{adq}|/|\mathbb{X}_{adq}|$ (indicating the probability of a test suite with incorrect test cases), and refer to it as the *Ratio of Effective Test Suite (**RETS**)*. Suppose $\mathcal{X}$ is a test suite in $\mathbb{X}_{adq}$. We measure the *Ratio of Effective Test Case (**RETC**)* of $\mathcal{X}$ by the ratio of the number of incorrect test cases in $\mathcal{X}$ to the mean number of mutants killed by a test case in $\mathcal{X}$, defined as $|\{t \in \mathcal{X} \mid t \text{ is incorrect}\}|/\frac{1}{|\mathcal{X}|}\sum_{t\in\mathcal{X}}|\{\overline{\mathcal{S}} \in \mathbb{S} \mid t \textit{ kills } \overline{\mathcal{S}}\}|$, to take into account that techniques each kill its own set of model mutants. A higher RETS or RETC value indicates a more effective technique.

4) Experimental Procedures: Experiment 1a for RQ1 on Effectiveness: We execute Alg. 1 and Alg. 2 with each pre-trained model $\mathcal{S}$ with its validation dataset. They produce a set of m mutants $\mathbb{S}$ and a well-trained autoencoder $\hat{\mathcal{A}}$. We use $\hat{\mathcal{A}}$ to generate n sample mutants for each sample in T_v. Each sample and its mutants t is fed to $\mathcal{S}$ to extract its execution trace ϕ_t and is also fed to each model mutant $\overline{\mathcal{S}} \in \mathbb{S}$ to compute its predicate vector π_t. We then input the set of execution traces Φ with their corresponding predicate vectors Π as groundtruths to train the estimator $\mathcal{E}$. The test dataset is served as the test pool T_e under prioritization. In the test phase, we profile the execution trace for each test case t in T_e on $\mathcal{S}$ and input the trace to the trained estimator $\hat{\mathcal{E}}$ to obtain t's predicate vector and compute the entropy. Then, we compute the informative mutation score of t according to Eq. (2). The test cases in T_e are then sorted by the informative mutation score in descending order to produce a ranked list L. We also configure DS and PM according to their original papers to generate their ranked lists. We measure each prioritized list L of test cases output by each technique through $RAUC(L, k)$ for the top k or $k\%$ elements, and denote the results as *RAUC*-k and *RAUC*-$k\%$. For the result of *RAUC*-100%, we also note it as RAUC-ALL.

Experiment 1b for RQ2 on Efficiency: We repeat experiment 1a and measure elapsed time of each stage in each technique.

Experiment 2a for RQ3 on Mutation Analysis: We evaluate to what extent these mutation-adequate test suites can expose model prediction failures effectively. It simulates a scenario of making an assessment on test suites.

To construct a mutation-adequate test suite for denoted as $\mathcal{X}$, our procedure is as follows: We iteratively and randomly pick a test case t from the test pool T_e and add it into $\mathcal{X}$ (initialized as empty) if t is predicted to kill at least one more mutant in $\mathbb{S}$ than $\mathcal{X}$, i.e, $\mathcal{X} \cup \{t\}$ increases the mutation coverage over $\mathbb{S}$ until $\mathcal{X}$ is mutation-adequate. When $\mathcal{X}$ and T_e achieve the same mutation coverage, the mutation-adequate $\mathcal{X}$ is constructed. Note that each technique produces its own set of model mutants $\mathbb{S}$. We repeat the above procedure 1000 times for each technique on each model and measure the RETC on each constructed test suite and the RETS on the set of test suites. This experiment is conducted on PM and EM as they are mutation-based techniques.

Experiment 2b for RQ3 on Mutation Analysis: An incorrect test case at the bottom section of a prioritized list of test cases (i.e., the sublist containing the test cases ranked lower than all others) is unlikely to be picked for labeling. Finding incorrect test cases in this section by simply going through the

TABLE II: Results of RAUC of all techniques.

ID	Task	Tech	RAUC								
			100	200	300	500	10%	20%	30%	50%	ALL
1	C	DS	100.0	100.0	100.0	99.75	97.78	95.25	93.36	91.31	91.31
1	C	PM	61.73	65.61	64.92	62.51	61.44	59.67	54.15	57.11	67.60
1	C	EM	100.0	100.0	100.0	99.87	99.31	98.18	96.16	93.04	93.04
2	C	DS	99.96	99.28	99.01	98.63	97.42	95.55	93.50	91.87	91.87
2	C	PM	65.21	60.57	56.91	59.89	57.21	59.36	53.52	55.89	66.63
2	C	EM	98.03	99.01	99.34	99.54	99.63	98.54	96.48	93.48	93.48
3	R	DS	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.
3	R	PM	69.75	69.11	71.31	66.76	63.86	57.60	54.65	53.26	42.83
3	R	EM	73.57	74.11	74.01	73.24	71.54	71.81	71.76	71.83	71.83
4	R	DS	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.
4	R	PM	61.99	66.51	61.88	62.99	53.57	50.86	52.55	54.22	51.02
4	R	EM	72.08	72.37	77.43	77.67	71.18	68.28	67.29	66.39	66.39

* C = classification, R = regression, Tech. = technique, n.a. = can't handle.

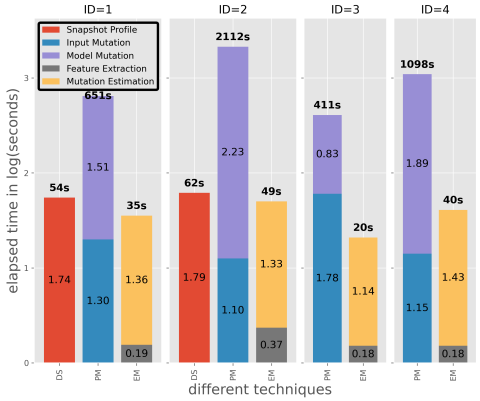


Figure 2: Efficiency of Techniques (in seconds with log_{10} scale).

ranked list will likely waste most labeling efforts, and yet, not examining this section will risk missing some critical samples, thus the task is challenging. We studied how to address this problem by constructing mutation-adequate test suites.

We examine the ranking list L generated by DS on each classification model and identify the L's bottom-$\bar{k}$ section that contains $2^a/10\%$ of all incorrect test cases in L for a = 2 to 6. For instance, if a = 3, the percentage is 0.8%. We construct, using the procedure in *Experiment 2a* by picking test cases from each such bottom-$\bar{k}$ section, to produce 1000 mutation-adequate test suites. For each regression model, we reorder the test cases in T_e in the descending order of the mean square error of an individual test case to construct a resultant list L and repeat the above procedure to construct test suites from the bottom-$\bar{k}$ section that also contains $2^a/10\%$ of incorrect test cases in L for a = 2 to 6. Then, we sort the test suites in ascending order of their test suite sizes and select the top-most 100 test suites, followed by measuring the proportions of incorrect test cases in each such test suite.

C. Results and Data Analysis

1) Answering RQ1: Table II summarizes the prioritization results on RAUC achieved by DS, PM, and EM. EM achieves the highest RAUC scores among the techniques for all k and has wider applicability than DS. We find EM outperforming PM on all four models by large extents in the number of selected incorrect test cases for every selection range.

On classification models, PM is significantly less effective than EM and DS. These models have 100–200 classes. In [5], PM was effective on models with fewer classes by one order of magnitude (e.g., 10 classes). The result shows that the effectiveness of PM does not scale up to handle more challenging models. We conduct Wilcoxon Signed-Rank Test [61] to investigate whether the difference between EM and PM is statistically meaningful using the raw data on each model. We find all p-values smaller than 0.05, indicating a significant difference at the 5% significance level.

By considering whether each sample in a ranked list is incorrect, we find EM and the test oracle produce almost same rank lists on the classification models. It shows that EM can be highly effective for classification models. As reviewed in Section II-D, DS's metric formula blocks it from prioritizing test cases for regression models, and EM can handle both regression and classification models. In many application domains (e.g., object recognition with object counting in an image), many models have multiple heads of regression and classification outputs. DS cannot handle them, whereas PM and EM are applicable. Although DS optimizes for classification models, EM can compete with it with wider applicability.

Answering RQ1 (Effective fault-based prioritization): EFFIMAP is significantly more effective than the previous state-of-the-art mutation-based approach (Prima) and has wider applicability than a prior art (Dissector).

2) Answering RQ2: Figure 2 shows the results (in seconds of log_{10} scale for elapsed time) on the efficiency of DS, PM, and EM on four subjects in the test phase. Also, DS is inapplicable to the two regression models. Each bar shows the time spent in different stages of a technique on applicable models. The number on top of a bar sums up the total elapsed time.

EM achieves the lowest time spent among all three techniques and across all four models. To complete the entire process of test prioritization in the test phase, EM spends 5.41%, 2.34%, 5.07%, and 3.68% of the time spent by PM. This difference is mainly due to the elimination of the mutation executions process, whereas, almost 100% of time spent by PM is consumed by this process. However, the main overhead of EM is in the inference time spent by the estimator model (i.e., the ranking process). *Tracer* (in gray) spends much less time than *Estimator* (in yellow) in all cases. The ratios of *Estimator* to TRACE in time spent on the four models are 7.2x, 3.6x, 6.4x, and 8.2x, respectively (where a number like 7.2x means 720%).

The learning-to-rank process and feature extraction in PM are lightweight (less than 0.1% of the total time spent and thus not shown in the figure). In DS, almost 99.9% of all the overheads are occupied by its snapshot profiling.

Answering RQ2 (Efficient fault-based prioritization): EFFIMAP incurs a significantly lower computational cost of mutation analysis conducted in the test phase by 18x to 42x compared to the state-of-the-art mutation-based approach

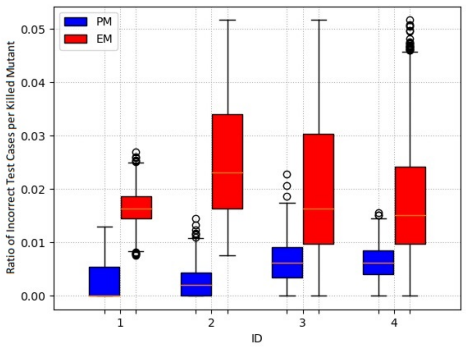


Figure 3: Ratios of Incorrect Test Cases Per Killed Mutant

(Prima). EFFIMAP is also slightly more efficient than the state-of-the-art metric-based technique (Dissector).

3) Answering RQ3: We find from *Experiment 2a* that in terms of RETS, across the four subjects, EM (versus PM) produces 1000 (vs. 417), 1000 (vs. 507), 862 (vs. 848), and 993 (vs. 957) test suites containing incorrect test cases with an average of 963.75 (vs. 682.25) test suites, respectively.

Figure 3 further summarizes the result of *Experiment 2a* in terms of RETC. In the boxplot, there are four pairs of bars where one pair of bars is for one model. Each pair of bars represents the pair of EM and PM. The y-axis is the RETC value. Across the four pairs of bars, EM achieves the averages of 0.017, 0.025, 0.018, and 0.017, respectively, in terms of RETC; and PM achieves 0.002, 0.003, 0.007, and 0.006, respectively. PM consistently achieves smaller values in this metric. It shows that the ability of PM's mutants to discover incorrect test cases is lower than that of EM. PM's correlation effect between killing a mutant to the discovery of a failure (by including an incorrect test case in a constructed test suite) at the test case level is significantly lower than that achieved by EM. The bars (excluding the outline regions) between the two techniques on each plot are completely non-overlapping, indicating that the differences are significant (confirmed by the hypothesis testing using HSD [62] for multiple comparisons with the 5% significance level and Bonferroni correction where the corresponding p-values ≤ 0.001 in all cases).

Figure 4 summarizes the result of *Experiment 2b*. Each plot shows the proportion of incorrect test cases in a test suite in the scenarios of different concentrations of incorrect test cases in the test pool (which is the bottom section of L) for selection (indicated by the value of $2^a/10\%$ in the experimental setup). The y-axis is the ratio of incorrect test cases in a constructed test suite. Across all models, EM achieves the best results, followed by DS to examine the bottom-ranked test cases (applicable to classification models only), and finally PM. Note that the variances of DS and PM are too small to be visualized as boxes in the figure.

On the two classification models, the curves for DS and PM are close to each other, and the curve for EM is located much above them. At each incorrect test case concentration level, EM is more effective than PM and DS by 4 folds or more. On the regression model with ID = 4, the difference between the pair of curves for EM and PM is relatively small. We find from the data that some test cases can kill a large proportion of all model mutants of EM, which leads to a reduction in the performance gap between EM and PM. On the regression model with ID = 3, we observe a trend in the gap between the two techniques, similar to the case of ID = 1. We have conducted hypothesis testing using HSD on the effective test case ratios to compare techniques. Across all four models, EM is significantly different from PM and DS at the 5% significance level with Bonferroni correction (p-values ≤ 0.001 in all cases).

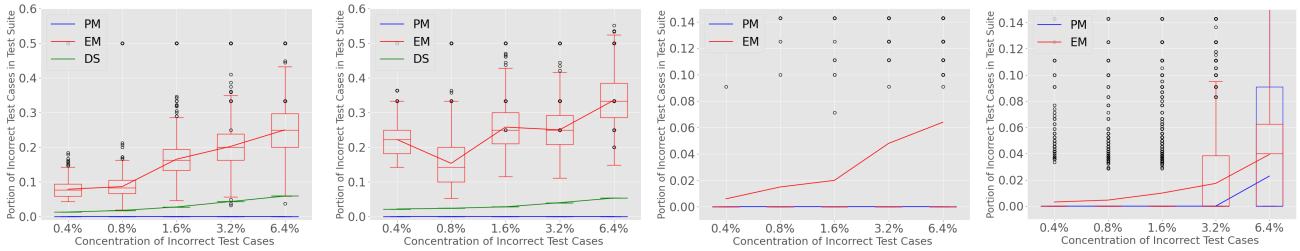
Answering RQ3 (Effective test suite prioritization): EFFIMAP significantly outperforms the previous state-of-the-art technique to produce mutation-adequate test suites with higher effective test suite ratios and effective test case ratios. When the concentrations of incorrect test cases in a test pool are low, EFFIMAP is also more effective than the latter technique.

D. Threats to Validity

We have evaluated the feasibility of applying PMT for test case prioritization in the DNN testing domain. For ease of presentation, let PMT be $PMT(\mathcal{S}, \mathsf{F})$, where PMT accepts a model $\mathcal{S}$ under mutation analysis for model learning and uses F as feature definitions. We configure PMT to generate m model mutants from the cross-version model $\mathcal{S}_{cv}$ (ResNet56 with same model architecture family) and cross-project model $\mathcal{S}_{cp}$ (VGG13 with different architecture) and n sample mutants (where m and n are same as Experiment 1). We configure PMT with two sets of features for learning, one set is the set of original code-based features of PMT (donated as F_c) and another set is the features of Prima (donated as F_m). We compare their resultant ranking lists on the model under test (ResNet32) to the random prioritization.

Figure 5 shows the ranking results of all four PMT variants compared with the random prioritization, where random prioritization merely reorders samples randomly, and ideal ordering is produced by the test oracle. All lines (except ideal) in the plot almost collapse together. The ranking effectiveness of PMT variants is just as effective as random prioritization. Among the four variants, in the zoom-in plot, applying a cross-project strategy or using code-based features even slightly performs worse than random prioritization, while using model-based features performs better. It reveals that PMT (no matter using cross-version or cross-project strategies for learning to rank) still requires significant changes to make it outperform random prioritization, which is challenging (in designing a set of model-specific features for learning within the same model architecture, echoing the discussion in Section II.) Thus, we exclude PMT as a candidate peer technique in our experiments.

Our models and datasets are widely used in previous testing work and have been well-trained. On each model, we measure the effectiveness and efficiency of all techniques whenever applicable. Using more datasets and models, metrics and hyperparameters can strengthen the generalization of the



(a) ID = 1 (b) ID = 2 (c) ID = 3 (d) ID = 4

Figure 4: Failure Detection of Mutation-Adequate Test Suites on Challenging Test Pools.

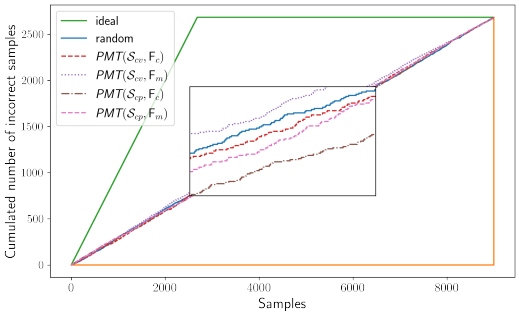


Figure 5: Ranking results of PMT for cross-version/cross-project + 2 feature sets. PMT cannot outperform Random.

experiment. We have tried to set m, n, η, and γ to different values and find that the relative order of the three techniques in the three RQs remains consistent with the results reported in this paper. The implementation may contain bugs. To make the experiment reliable, we reused the available code from [5], [23] and tested and compared it with their results. Almost 95% of the time spent by each EM and PM is consumed by the mutation process, mostly in the offline pre-test phase. Besides labeling cost reduction, two potential use cases of EFFIMAP are input validation and guided retraining as reported in related works [16], [23]. We have empirically evaluated on the model with ID = 1 that EM can achieve the ROC-AUC value [63] of 0.8185 measured by the informative mutation score to reject out-of-distribution samples. Moreover, in this model, EM outperforms the random selection of adversarial samples (of gaussian blur type) by 3.16% in robust accuracy after retraining the model under test with the selected samples. EM selects stronger adversarial samples. Although we evaluate EFFIMAP on the image classification and regression tasks only, the idea of EFFIMAP formulating predictive mutation analysis for DNN models could be generalized to other tasks (e.g., object detection) by defining the predicate of mutation.

V. Related Work

Our work is in the area of test case prioritization [9], [16], [17], [24]. DeepGini [16] computed the Gini impurity on the probability vector of the classification models to prioritize test cases. Dissector [23] computed a metric based on the trend of the relative performance of the model under test and its submodels. Unlike EFFIMAP, they [16], [23] cannot handle regression models. Byun et al. [64] measured the confidence, uncertainty, and surprise for test cases for prioritizing test cases, and found sentiment measures effective in flagging problematic inputs. LSA/DSA [19] were built atop the degree of surprise of test cases with respect to the training samples for prioritization. However, the surprise metric requires a large computational resource as the computation happens between a test case to each training sample. EFFIMAP only needs to process its TRACER and ESTIMATOR, which is lightweight. We have extensively reviewed Prima [5] in previous sections as a mutation-based technique. The experiment of Prima showed that Prima outperformed LSA and DSA [19] and DeepGini [16]. EFFIMAP boosts effectiveness and efficiency compared with Prima. A related area is test case selection. PEACEPACT [65] perturbed test samples to assess how easy to generate adversarial examples from them by altering neuron activation. Zhang et al. [24] computed the activation patterns of each class as constraints and checked whether a test case might violate them. EFFIMAP also uses the information about neuron activation for estimation. DeepReduce [17] reduced a test suite by prioritizing the test cases within their distribution of the test suite. EFFIMAP builds a ranking model for test case prioritization.

Mutation testing [28]–[30] on DNN models is emerging. DeepMutation [25] was the first mutation testing framework with model mutants and sample mutants. Wang et al. [9] used mutation analysis to distinguish adversarial examples (from clean ones). EFFIMAP also constructs a set of model mutants with clear cause and effect from the literature. PMT [31] proposed predictive mutation testing with cross-version/project strategies for testing traditional programs, but as we discussed, it was ineffective in the DNN testing domain. Prima [5] integrated a comprehensive mutation analysis into TCP. EFFIMAP effectively bridges the gap between predictive mutation analysis and test case prioritization.

Diverse methods, such as random-based generation [66], search-based generation [15], [67], and metamorphic relation-based generation [68]–[70] have been studied for test case generation. SENSEI [66] fuzzes to generate data for robustness improvement. DeepHyperion [67] used an illumination search

to find misbehaving samples by analysis of selective neurons. MODE [15] computed differential heat maps and searched a given dataset to find the samples with a larger dot product value of its heat map to the former map. Novel metamorphic relations were formulated in [68]–[70] to generate test pairs and detect violations for machine translation models. EFFIMAP formulates an autoencoder strategy to generate in-distribution sample mutants to alleviate the lack of data issue, which is different from the above techniques.

VI. CONCLUSION

We have presented EFFIMAP, a novel predictive mutation analysis approach to prioritizing test cases while significantly reducing the computational cost in mutation analysis. EFFIMAP comes with a novel target-oriented mutant generation method to generate a set of diverse mutants. It formulates a novel distribution-oriented execution trace as features and a novel and effective mutation analysis strategy to substitute the traditional mutation analysis performed in the test phase. The experimental results in this paper demonstrate that EFFIMAP is effective and efficient in reducing the computational cost of mutation analysis, showing the feasibility of predictive mutation analysis in DNN model testing.

REFERENCES

[1] G. Klein, Y. Kim, Y. Deng, J. Senellart, and A. Rush, "OpenNMT: Open-source toolkit for neural machine translation," in *Proceedings of ACL 2017, System Demonstrations*. Vancouver, Canada: Association for Computational Linguistics, Jul. 2017, pp. 67–72.

[2] S. Minaee, Y. Y. Boykov, F. Porikli, A. J. Plaza, N. Kehtarnavaz, and D. Terzopoulos, "Image segmentation using deep learning: A survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pp. 1–1, 2021.

[3] M. Bakator and D. Radosav, "Deep learning and medical diagnosis: A review of literature," *Multimodal Technologies and Interaction*, vol. 2, no. 3, 2018.

[4] C. Chen, A. Seff, A. Kornhauser, and J. Xiao, "Deepdriving: Learning affordance for direct perception in autonomous driving," in *IEEE International Conference on Computer Vision (ICCV)*, 2015, pp. 2722–2730.

[5] Z. Wang, H. You, J. Chen, Y. Zhang, X. Dong, and W. Zhang, "Prioritizing test inputs for deep neural networks via mutation analysis," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 397–409.

[6] K. Pei, Y. Cao, J. Yang, and S. Jana, "Deepxplore: Automated whitebox testing of deep learning systems," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 1–18.

[7] M. Zhang, Y. Zhang, L. Zhang, C. Liu, and S. Khurshid, "Deeproad: Gan-based metamorphic testing and input validation framework for autonomous driving systems," in *33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2018, pp. 132–142.

[8] I. Dunn, H. Pouget, D. Kroening, and T. Melham, "Exposing previously undetectable faults in deep neural networks," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 56–66.

[9] J. Wang, G. Dong, J. Sun, X. Wang, and P. Zhang, "Adversarial sample detection for deep neural network through model mutation testing," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 1245–1256.

[10] C. Olsson, "(2018-03-01) incident number 4 in *McGregor, S.* (ed.) *Artificial Intelligence Incident Database*. responsible ai collaborative." https://incidentdatabase.ai/cite/4?lang=en, 2022, accessed: 2022-04-30.

[11] Wikipedia contributors, "Death of elaine herzberg — Wikipedia, the free encyclopedia," https://en.wikipedia.org/w/index.php?title=Death_of_Elaine_Herzberg, 2022, [Online; accessed 19-August-2022].

[12] B. Allyn, "'the computer got it wrong': How facial recognition led to false arrest of black man," https://www.npr.org/2020/06/24/882683463/the-computer-got-it-wrong-howfacial-recognition-led-to-a-false-arrest-in-michig, 2020, nPR, June 24, 2020. Online; accessed 30 April 2022.

[13] P. Ji, Y. Feng, J. Liu, Z. Zhao, and B. Xu, "Automated testing for machine translation via constituency invariance," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021, pp. 468–479.

[14] A. Odena, C. Olsson, D. Andersen, and I. Goodfellow, "TensorFuzz: Debugging neural networks with coverage-guided fuzzing," in *Proceedings of the 36th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, K. Chaudhuri and R. Salakhutdinov, Eds., vol. 97. PMLR, 09–15 Jun 2019, pp. 4901–4911.

[15] S. Ma, Y. Liu, W.-C. Lee, X. Zhang, and A. Grama, "Mode: Automated neural network model debugging via state differential analysis and input selection," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 175–186.

[16] Y. Feng, Q. Shi, X. Gao, J. Wan, C. Fang, and Z. Chen, "Deepgini: Prioritizing massive tests to enhance the robustness of deep neural networks," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 177–188.

[17] J. Zhou, F. Li, J. Dong, H. Zhang, and D. Hao, "Cost-effective testing of a deep learning model through input reduction," in *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, 2020, pp. 289–300.

[18] X. Xie, L. Ma, F. Juefei-Xu, M. Xue, H. Chen, Y. Liu, J. Zhao, B. Li, J. Yin, and S. See, *DeepHunter: A Coverage-Guided Fuzz Testing Framework for Deep Neural Networks*. New York, NY, USA: Association for Computing Machinery, 2019, p. 146–157.

[19] J. Kim, R. Feldt, and S. Yoo, "Guiding deep learning system testing using surprise adequacy," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19. IEEE Press, 2019, p. 1039–1049.

[20] Z. Zhang, P. Wang, H. Guo, Z. Wang, Y. Zhou, and Z. Huang, "Deepbackground: Metamorphic testing for deep-learning-driven image recognition systems accompanied by background-relevance," *Information and Software Technology*, vol. 140, p. 106701, 2021.

[21] Y. Zhang, L. Ren, L. Chen, Y. Xiong, S.-C. Cheung, and T. Xie, "Detecting numerical bugs in neural network architectures," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 826–837.

[22] H. Tu, Z. Yu, and T. Menzies, "Better data labelling with emblem (and how that impacts defect prediction)," *IEEE Transactions on Software Engineering*, vol. 48, no. 1, pp. 278–294, 2022.

[23] H. Wang, J. Xu, C. Xu, X. Ma, and J. Lu, "Dissector: Input validation for deep learning applications by crossing-layer dissection," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, 2020, pp. 727–738.

[24] K. Zhang, Y. Zhang, L. Zhang, H. Gao, R. Yan, and J. Yan, "Neuron activation frequency based test case prioritization," in *2020 International Symposium on Theoretical Aspects of Software Engineering (TASE)*, 2020, pp. 81–88.

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

[26] J. R. Quinlan, "Induction of decision trees," *Machine Learning*, vol. 1, no. 1, pp. 81–106, Mar 1986.

[27] S. Lathuilière, P. Mesejo, X. Alameda-Pineda, and R. Horaud, "A comprehensive analysis of deep regression," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 42, no. 9, pp. 2065–2081, 2020.

[28] R. Gopinath, A. Alipour, I. Ahmed, C. Jensen, and A. Groce, "How hard does mutation analysis have to be, anyway?" in *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, 2015, pp. 216–227.

[29] G. Guizzo, F. Sarro, and M. Harman, *Cost Measures Matter for Mutation Testing Study Validity*. New York, NY, USA: Association for Computing Machinery, 2020, p. 1127–1139.

[30] R. Gopinath, M. A. Alipour, I. Ahmed, C. Jensen, and A. Groce, "On the limits of mutation reduction strategies," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 511–522.

[31] J. Zhang, L. Zhang, M. Harman, D. Hao, Y. Jia, and L. Zhang, "Predictive mutation testing," *IEEE Transactions on Software Engineering*, vol. 45, no. 9, pp. 898–918, 2019.

[32] D. Mao, L. Chen, and L. Zhang, "An extensive study on cross-project predictive mutation testing," in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, 2019, pp. 160–171.

[33] J. Zhang, Z. Wang, L. Zhang, D. Hao, L. Zang, S. Cheng, and L. Zhang, "Predictive mutation testing," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 342–353.

[34] Z. Zhong, Y. Tian, and B. Ray, "Understanding local robustness of deep neural networks under natural variations," in *Fundamental Approaches to Software Engineering*, E. Guerra and M. Stoelinga, Eds. Cham: Springer International Publishing, 2021, pp. 313–337.

[35] Y. Tian, K. Pei, S. Jana, and B. Ray, "Deeptest: Automated testing of deep-neural-network-driven autonomous cars," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 303–314.

[36] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," 2013.

[37] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 785–794.

[38] Z. Wang, M. Yan, J. Chen, S. Liu, and D. Zhang, "Deep learning library testing via effective model generation," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 788–799.

[39] Y. He, X. Zhang, and J. Sun, "Channel pruning for accelerating very deep neural networks," in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, Oct 2017.

[40] Y. Sun, C. Guo, and Y. Li, "React: Out-of-distribution detection with rectified activations," in *Advances in Neural Information Processing Systems*, 2021.

[41] C. Shorten and T. M. Khoshgoftaar, "A survey on image data augmentation for deep learning," *Journal of Big Data*, vol. 6, no. 1, p. 60, Jul 2019.

[42] H. Zhang, M. Cisse, Y. N. Dauphin, and D. Lopez-Paz, "mixup: Beyond empirical risk minimization," in *International Conference on Learning Representations*, 2018.

[43] Z. Zhao, G. Chen, J. Wang, Y. Yang, F. Song, and J. Sun, "Attack as defense: Characterizing adversarial examples using robustness," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 42–55.

[44] C. E. Shannon, "A mathematical theory of communication," *Bell System Technical Journal*, vol. 27, no. 3, p. 379–423, july 1948.

[45] ——, "A mathematical theory of communication," *Bell System Technical Journal*, vol. 27, no. 4, p. 623–656, october 1948.

[46] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and Édouard Duchesnay, "Scikit-learn: Machine learning in python," *Journal of Machine Learning Research*, vol. 12, no. 85, pp. 2825–2830, 2011.

[47] J. Gehrke, *Scalable Decision Tree Construction*. New York, NY: Springer New York, 2018, pp. 3275–3281.

[48] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. Red Hook, NY, USA: Curran Associates Inc., 2019.

[49] Z. Wei, "Implementation of EffiMAP," 11 2022. [Online]. Available: https://github.com/Wsine/effimap

[50] ParagonLight, "Implementation repository of icse'20 dissector," https://github.com/ParagonLight/dissector, 2021.

[51] sail repos, "Implementation repository of icse'21 prima," https://github.com/sail-repos/PRIMA, 2021.

[52] Wikipedia contributors, "Gaussian noise — Wikipedia, the free encyclopedia," 2022, [Online; accessed 6-May-2022]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Gaussian_noise

[53] chenyaofo, "Pytorch cifar models," https://github.com/chenyaofo/pytorch-cifar-models, 2021.

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

[55] tjmoon0104, "Tiny-imagenet classifier using pytorch," https://github.com/tjmoon0104/Tiny-ImageNet-Classifier, 2018.

[56] Y. Le and X. S. Yang, "Tiny imagenet visual recognition challenge," 2015.

[57] xichen, "Base pretrained models and datasets in pytorch," https://github.com/aaron-xichen/pytorch-playground, 2020.

[58] L. Deng, "The mnist database of handwritten digit images for machine learning research," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[59] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng, "Reading digits in natural images with unsupervised feature learning," in *NIPS Workshop on Deep Learning and Unsupervised Feature Learning 2011*, 2011.

[60] Z. Wang, S. Wu, C. Liu, S. Wu, and K. Xiao, "The regression of mnist dataset based on convolutional neural network," in *The International Conference on Advanced Machine Learning Technologies and Applications (AMLTA2019)*, A. E. Hassanien, A. T. Azar, T. Gaber, R. Bhatnagar, and M. F. Tolba, Eds. Cham: Springer International Publishing, 2020, pp. 59–68.

[61] F. Wilcoxon, S. Katti, and R. A. Wilcox, *Critical values and probability levels for the Wilcoxon rank sum test and the Wilcoxon signed rank test*. American Cyanamid Pearl River (NY), 1963, vol. 1.

[62] H. Abdi and L. J. Williams, "Tukey's honestly significant difference (hsd) test," *Encyclopedia of research design*, vol. 3, no. 1, pp. 1–5, 2010.

[63] J. A. Hanley and B. J. McNeil, "The meaning and use of the area under a receiver operating characteristic (ROC) curve," *Radiology*, vol. 143, no. 1, pp. 29–36, Apr. 1982.

[64] T. Byun, V. Sharma, A. Vijayakumar, S. Rayadurgam, and D. Cofer, "Input prioritization for testing neural networks," in *2019 IEEE International Conference On Artificial Intelligence Testing (AITest)*, 2019, pp. 63–70.

[65] Z. Li, L. Zhang, J. Yan, J. Zhang, Z. Zhang, and T. H. Tse, "Peacepact: Prioritizing examples to accelerate perturbation-based adversary generation for dnn classification testing," in *2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS)*, 2020, pp. 406–413.

[66] X. Gao, R. K. Saha, M. R. Prasad, and A. Roychoudhury, "Fuzz testing based data augmentation to improve robustness of deep neural networks," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1147–1158.

[67] T. Zohdinasab, V. Riccio, A. Gambi, and P. Tonella, "Deephyperion: Exploring the feature space of deep learning-based systems through illumination search," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 79–90.

[68] P. Ji, Y. Feng, J. Liu, Z. Zhao, and B. Xu, "Automated testing for machine translation via constituency invariance," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021, pp. 468–479.

[69] P. He, C. Meister, and Z. Su, "Testing machine translation via referential transparency," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 410–422.

[70] Z. Sun, J. M. Zhang, M. Harman, M. Papadakis, and L. Zhang, "Automatic testing and improvement of machine translation," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 974–985.