DeepPerform: An Efficient Approach for Performance Testing of Resource-Constrained Neural Networks

Simin Chen  
simin.chen@UTDallas.edu  
UT Dallas  
Dallas, USA

Cong Liu  
congl@ucr.edu  
UC Riverside  
Riverside, USA

Mirazul Haque  
mirazul.haque@utdallas.edu  
UT Dallas  
Dallas, USA

Wei Yang  
wei.yang@utdallas.edu  
UT Dallas  
Dallas, USA

ABSTRACT

Today, an increasing number of Adaptive Deep Neural Networks (AdNNs) are being used on resource-constrained embedded devices. We observe that, similar to traditional software, redundant computation exists in AdNNs, resulting in considerable performance degradation. The performance degradation is dependent on the input and is referred to as input-dependent performance bottlenecks (IDPBs). To ensure an AdNN satisfies the performance requirements of resource-constrained applications, it is essential to conduct performance testing to detect IDPBs in the AdNN. Existing neural network testing methods are primarily concerned with correctness testing, which does not involve performance testing. To fill this gap, we propose DeepPerform, a scalable approach to generate test samples to detect the IDPBs in AdNNs. We first demonstrate how the problem of generating performance test samples detecting IDPBs can be formulated as an optimization problem. Following that, we demonstrate how DeepPerform efficiently handles the optimization problem by learning and estimating the distribution of AdNNs’ computational consumption. We evaluate DeepPerform on three widely used datasets against five popular AdNN models. The results show that DeepPerform generates test samples that cause more severe performance degradation (FLOPs: increase up to 552%). Furthermore, DeepPerform is substantially more efficient than the baseline methods in generating test inputs (runtime overhead: only 6–10 milliseconds).

CCS CONCEPTS
• Software and its engineering → Software notations and tools; • Computing methodologies → Machine learning.

KEYWORDS
Machine learning, software testing, performance analysis
resource-constrained devices, there is a strong need for an automated performance testing framework to find IDPBs.

We identify two main challenges in designing such a performance testing framework. First, traditional performance metrics (e.g., latency, energy consumption) are hardware-dependent metrics. Measuring these hardware-dependent metrics requires repeated experiments because of the system noises. Thus, directly applying these hardware-dependent metrics as guidelines to generate test samples would be inefficient. Second, ADNNs’ performance adjustment strategy is learned from datasets rather than conforming to logic specifications (such as relations between model inputs and outputs). Without a logical relation between ADNNs’ inputs and ADNNs’ performance, it is challenging to search for inputs that can trigger performance degradation in ADNNs.

To address the above challenges, we propose DeepPerform, which enables efficient performance testing for ADNNs by generating test samples that trigger IDPBs of ADNNs (DeepPerform focuses on the performance testing of latency degradation and energy consumption degradation as these two metrics are critical for performance testing [3, 49]). To address the first challenge, we first conduct a preliminary study (§3) to illustrate the relationship between computational complexity (FLOPs) and hardware-dependent performance metrics (latency, energy consumption). We then transfer the problem of degrading system performance into increasing ADNNs’ computational complexity (Eq.(3)). To address the second challenge, we apply the algorithm similar to Generative Adversarial Networks (GANs) to design DeepPerform. In the training process, DeepPerform learns and approximates the distribution of the samples that require more computational complexity. After DeepPerform is well trained, DeepPerform generates test samples that activate more redundant computational units in ADNNs. In addition, because DeepPerform does not require backward propagation during the test sample generation phase, DeepPerform generates test samples much more efficiently, thus more scalable for comprehensive testing on large models and datasets.

To evaluate DeepPerform, we select five widely-used model-dataset pairs as experimental subjects and explore following four perspectives: effectiveness, efficiency, coverage, and sensitivity. First, to evaluate the effectiveness of the performance degradation caused by test samples generated by DeepPerform, we measure the increase in computational complexity (FLOPs) and resource consumption (latency, energy) caused by the inputs generated by DeepPerform. For measuring efficiency, we evaluate the online time-oversheads and total time-oversheads of DeepPerform in generating different scale samples for different scale experimental subjects. For coverage evaluation, we measure the computational units covered by the test inputs generated by DeepPerform. For sensitivity measurement, we measure how DeepPerform’s effectiveness is dependent on the ADNNs’ configurations and hardware platforms. The experimental results show that DeepPerform generated inputs increase ADNNs’ computational FLOPs up to 552%, with 6-10 milliseconds overheads for generating one test sample. We summarize our contribution as follows:

- **Approach.** We propose a learning-based approach 1, namely DeepPerform, to learn the distribution to generate the test samples for performance testing. Our novel design enables generating test samples more efficiently, thus enable scalable performance testing.

- **Evaluation.** We evaluate DeepPerform on five ADNN models and three datasets. The evaluation results suggest that DeepPerform finds more severe diverse performance bugs while covering more ADNNs’ behaviors, with only 6-10 milliseconds of online overheads for generating test inputs.

- **Application.** We demonstrate that developers could benefit from DeepPerform. Specifically, developers can use the test samples generated by DeepPerform to train a detector to filter out the inputs requiring high abnormal computational resources (§6).

2 BACKGROUND

2.1 AdNNs’ Working Mechanisms

![Figure 1: Working mechanism of AdNNs](https://github.com/SeekingDream/DeepPerform)

The main objective of AdNNs [5, 12, 14, 24, 29, 35, 41, 44, 49, 52] is to balance performance and accuracy. As shown in Fig. 2, AdNNs will allocate more computational resources to inputs with more complex semantics. AdNNs use intermediate outputs to deactivate specific components of neural networks, thus reducing computing resource consumption. According to the working mechanism, AdNNs can be divided mainly into two types: Conditional-skipping AdNNs and Early-termination AdNNs, as shown in Fig. 1. Conditional-skipping AdNNs skip specific layers/blocks if the intermediate outputs provided by specified computing units match predefined criteria. 2 (in the case of ResNet). The working mechanism of the conditional-skipping AdNN can be formulated as:

\[
\begin{cases} 
  \text{In}_{i+1} = \text{Out}_i, & \text{if } B_i(x) \geq \tau_i \\
  \text{Out}_{i+1} = \text{Out}_i, & \text{otherwise}
\end{cases}
\]

\(1\)

where \(x\) is the input, \(\text{In}_i\) represents the input of \(i^{th}\) layer, \(\text{Out}_i\) represents the output of \(i^{th}\) layer, \(B_i\) represents the specified computing unit output of \(i^{th}\) layer and \(\tau_i\) is the configurable threshold that decides AdNNs’ performance-accuracy trade-off mode. Early-termination AdNNs terminate computation early if the intermediate

1https://github.com/SeekingDream/DeepPerform

2A block consists of multiple layers whose output is determined by adding the output of the last layer and input to the block.
outputs satisfy a particular criteria. The working mechanism of early-termination AdNNs can be formulated as,
\[
\begin{align*}
\text{Exit}_{i+1}(x) &= \text{Exit}_i(x), \quad \text{if } B_i(x) \geq \tau_i \\
\text{In}_{i+1}(x) &= \text{Out}_i(x), \quad \text{otherwise}
\end{align*}
\] (2)

2.2 Redundant Computation

In a software program, if an operation is not required but performed, we term the operation as redundant operation. For Adaptive Neural Networks, if a component is activated without affecting AdNNs’ final predictions, we define the computation as a redundant computation. AdNNs are created based on the philosophy that all the inputs should not require all DNN components for inference. For example, we can refer to the images in Fig. 2. The left box shows the AdNNs’ design philosophy. That is, AdNNs consume more energy for detecting images with further complexity. However, when the third image in the left box is perturbed with minimal perturbations and becomes the rightmost one, AdNNs’ inference energy consumption will increase significantly (from 30/6 to 68/6). We refer to such additional computation as redundant computation or performance degradation.

2.3 Performance & Computational Complexity

In this section, we describe the relationship between hardware-dependent performance metrics and DNN computational complexity. Although many metrics can reflect DNN performance, we chose latency and energy consumption as hardware-dependent performance metrics because of their critical nature for real-time embedded systems [3, 49]. Measuring hardware-dependent performance metrics (e.g., latency, energy consumption) usually requires many repeated experiments, which is costly. Hence, existing work [12, 14, 29, 35, 41, 52] proposes to apply floating point operations (FLOPs) to represent DNN computational complexity. However, a recent study [43] demonstrates that simply lowering DNN computational complexity (FLOPs) does not always improve DNN runtime performance. This is because modern hardware platforms usually apply parallelism to handle DNN floating-point operations (FLOPs). Parallelism can accelerate computation within layers, while each DNN layer is computed sequentially. Thus, for two DNNs with the same total FLOPs, different FLOPs allocating strategies will result in different parallelism utilization and different DNN model performance. However, for AdNNs, each layer/block usually has a similar structure and FLOPs [12, 14, 34, 52]. Thus the parallelism utilization is similar for each block. Because parallelism can not accelerate computation between blocks, increasing the number of computational blocks/layers will degrade AdNNs’ performance. To further understand the relation between AdNNs’ FLOPs and AdNNs’ model performance, we conduct a study in §3.

3 PRELIMINARY STUDY

3.1 Study Approach

Our intuition is to explore the worst computational complexity of an algorithm or model. For AdNNs, the basic computation are the floating-point operations (FLOPs). Thus, we made an assumption that the FLOPs count of an AdNN is a hardware-independent metric to approximate AdNN performance. To validate such an assumption, we conduct an empirical study. Specifically, we compute the Pearson Product-moment Correlation Co-efficient (PCCs) [40] between AdNN FLOPs against AdNN latency and energy consumption. PCCs are widely used in statistical methods to measure the linear correlation between two variables. PCCs are normalized covariance measurements, ranging from -1 to 1. Higher PCCs indicate that the two variables are more positively related. If the PCCs between FLOPs against system latency and system energy consumption are both high, then we validate our assumption.

3.2 Study Model & Dataset

We select subjects (e.g., model, dataset) following policies below.

• The selected subjects are publicly available.

• The selected subjects are widely used in existing work.

• The selected dataset and models should be diverse from different perspectives. e.g., the selected models should include both early-termination and conditional-skipping AdNNs.

We select five popular model-dataset combinations used for image classification tasks as our experimental subjects. The dataset and the corresponding model are listed in Table 1. We explain the selected datasets and corresponding models below.

Datasets. CIFAR-10 [25] is a database for object recognition. There is a total of ten object classes for this dataset, and the image size of the image in CIFAR-10 is 32 × 32. CIFAR-10 contains 50,000 training images and 10,000 testing images. CIFAR-100 [25] is similar to CIFAR-10 [25] but with 100 classes. It also contains 50,000 training images and 10,000 testing images. SVHN [36] is a real-world image dataset obtained from house numbers in Google Street View images. There are 73257 training images and 26032 testing images in SVHN.

Models. For CIFAR-10 dataset, we use SkipNet [52] and BlockDrop [53] models. SkipNet applies reinforcement learning to train DNNs to skip unnecessary blocks, and BlockDrop trains a policy network to activate partial blocks to save computation costs. We download trained SkipNet and BlockDrop from the authors’ websites. For CIFAR-100 dataset, we use RaNet [56] and DeepShallow [24] models for evaluation. DeepShallow adaptive scales DNN depth, while RaNet scales both input resolution and DNN depth to balance accuracy and performance. For SVHN dataset, DeepShallow [24] is used for evaluation. For RaNet [56] and DeepShallow [24] architecture, the author does not release the trained model weights but open-source their training codes. Therefore, we follow the authors’ instructions to train the model weights.

3.3 Study Process

We begin by evaluating each model’s computational complexity on the original hold-out test dataset. After that, we deploy the AdNN
Table 1: Experiential subject and model performance

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Subject</th>
<th>Model</th>
<th>FLOPs</th>
<th>CPU (Quad-Core ARMv8 Cortex-A57 MPCore)</th>
<th>GPU (NVIDIA Pascal™ GPU architecture with 256 cores)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Latency</td>
<td>Energy</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Min</td>
<td>Avg</td>
</tr>
<tr>
<td>CIFAR10 (C10)</td>
<td>SkipNet (SN)</td>
<td>195.44</td>
<td>248.62</td>
<td>336.99</td>
<td>0.44</td>
</tr>
<tr>
<td></td>
<td>BlockDrop (BD)</td>
<td>72.56</td>
<td>180.51</td>
<td>228.27</td>
<td>0.41</td>
</tr>
<tr>
<td>CIFAR100 (C100)</td>
<td>DeepShallow (DS)</td>
<td>38.68</td>
<td>110.47</td>
<td>252.22</td>
<td>0.04</td>
</tr>
<tr>
<td></td>
<td>RaNet (RN)</td>
<td>31.50</td>
<td>41.17</td>
<td>68.88</td>
<td>0.07</td>
</tr>
<tr>
<td>SVHN</td>
<td>DeepShallow (DS)</td>
<td>38.74</td>
<td>161.40</td>
<td>252.95</td>
<td>0.04</td>
</tr>
</tbody>
</table>

Table 2: PCCs between FLOPs against latency and energy

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Metric</th>
<th>SN_C10</th>
<th>RN_C100</th>
<th>BD_C10</th>
<th>DS_C100</th>
<th>DS_SVHN</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Latency</td>
<td>0.68</td>
<td>0.67</td>
<td>0.93</td>
<td>0.99</td>
<td>0.95</td>
</tr>
<tr>
<td></td>
<td>Energy</td>
<td>0.65</td>
<td>0.64</td>
<td>0.93</td>
<td>0.98</td>
<td>0.95</td>
</tr>
<tr>
<td>GPU</td>
<td>Latency</td>
<td>0.48</td>
<td>0.56</td>
<td>0.91</td>
<td>0.99</td>
<td>0.97</td>
</tr>
<tr>
<td></td>
<td>Energy</td>
<td>0.53</td>
<td>0.64</td>
<td>0.91</td>
<td>0.99</td>
<td>0.97</td>
</tr>
</tbody>
</table>

Table 3: System availability under performance degradation

<table>
<thead>
<tr>
<th>Subject</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>SN_C10</td>
<td>10.000</td>
</tr>
<tr>
<td>BD_C10</td>
<td>10.000</td>
</tr>
<tr>
<td>RN_C100</td>
<td>10.000</td>
</tr>
<tr>
<td>DS_C100</td>
<td>10.000</td>
</tr>
<tr>
<td>DS_SVHN</td>
<td>10.000</td>
</tr>
</tbody>
</table>

model on an Nvidia TX2 [37] and measure latency and energy usage. Through Table 1, we present the FLOPs, latency, and energy consumption of each AdNN. We observe that the model would cost a different number of FLOPs for different test samples, and the variance between each test sample could be significant. For instance, for dataset CIFAR-100 and model RaNet, the minimum FLOPs are 31.5M, while the maximum FLOPs are 188.68M.

3.4 Study Results

From the PCCs results in Table 2, we have the following observations: (i) The PCCs are more than 0.48 for all subjects. The results imply that FLOPs are positively related to latency and energy consumption in AdNNs [40]. Especially for DS_C100, the PCC achieves 0.99, which indicates the strong linear relationship between FLOPs and runtime performance. (ii) The PCCs for the same subject on different hardware devices are remarkably similar (e.g., with an average difference of 0.04). According to the findings, the PCCs between FLOPs and latency/energy consumption are hardware independent. The statistical observations of PCCs confirm our assumption; that is, the FLOPs of AdNN handling an input is a hardware-independent metric that can approximate AdNN performance on multiple hardware platforms.

3.5 Motivating Example

To further understand the necessity of conducting performance testing for AdNNs, we use one real-world example to show the harmful consequences of performance degradation. In particular, we use TorchMobile to deploy each AdNN model on Samsung Galaxy S9+, an Android device with 6GB RAM and 3500mAh battery capacity.

We randomly select inputs from the original test dataset of each subject (i.e., Table 1) as seed inputs and perturb the selected seed inputs with random perturbation. Next, we conduct two experiments (one on the selected seed inputs and another one on the perturbed one) on the phone with the same battery. Specifically, we feed both datasets into AdNN for object classification and record the number of inputs successfully inferred before the battery runs out (We set the initial battery as the battery that can infer 10,000 inputs from the original dataset). The results are shown in Table 3, where the column “original” and “perturbed” show the number of inputs successfully inferred, and the column “ratio” shows the corresponding system availability ratio (i.e., the system can successfully complete the percentage of the assigned tasks under performance degradation). Such experimental results highlight the importance of AdNN performance testing before deployment. Otherwise, AdNNs’ performance degradation will endanger the deployed system’s availability.

4 APPROACH

In this section, we introduce the detail design of DeepPerform.

4.1 Performance Test Samples for AdNNs

Following existing work [17, 27, 27], we define performance test samples as the inputs that require redundant computation and cause performance degradation (e.g., higher energy consumption). Because our work focus on testing AdNNs, we begin by introducing redundant computation in AdNNs. Like traditional software, existing work [17, 24] has shown redundant computation also exist in AdNNs. Formally, let \( f \) denotes the computational complexity of neural network, then there exist redundant computation in AdNNs. Formally, let \( g_f(\cdot) \) denotes the function that measures the computational complexity of neural network \( f(\cdot) \), and \( T_f(\cdot) \) denotes a semantic-equivalent transformation in the input domain. As the example in Fig. 2, \( T_f(\cdot) \) could be changing some unnoticeable pixels in the input images. If \( g_f(T_f(x_i)) > g_f(x_i) \) and \( f(x_i) \) is correctly computed, then there exist redundant computation in the model \( f(\cdot) \) handling \( T_f(x_i) \). In this paper, we consider unnoticeable perturbations as our transformations \( T_f(\cdot) \), the same as the existing work [8, 17, 22]. Finally, we formulate our objective to generate performance test samples as searching such unnoticeable input transformation \( T_f(\cdot) \), as shown in Eq.(3).

\[
g(T_f(x)) >> g(x) \quad T_f(x) = (x + \delta(x)| \quad ||\delta(x)||_p \leq \varepsilon)
\] (3)

4.2 DeepPerform Framework

Fig. 3 illustrates the overall architecture of DeepPerform, which is based on the paradigm of Generative Adversarial Networks (GANs).
GANs mainly consist of a generator \( G(\cdot) \) and a discriminator \( D(\cdot) \). The input \( x \) of the generator \( G(\cdot) \) is a seed input and the output \( G(x) \) is a minimal perturbation (i.e., \( \delta(x) \) in Eq.(3)). After applying the generated perturbation to the seed input, the test sample \( t_f(x) = x + G(x) \) is sent to the discriminator. The discriminator \( D(\cdot) \) is designed to distinguish the generated test samples \( x + G(x) \) and the original samples \( x \). After training, the generator would generate more unnoticeable perturbation, correspondingly, the discriminator would also be more accurate in distinguishing original samples and generated samples. After being well trained, the discriminator and the generator would reach a Nash Equilibrium, which implies the generated test samples are challenging to be distinguished from the original samples.

\[
L_{GAN} = \mathbb{E}_x \log D(x) + \mathbb{E}_x \log [1 - D(x + G(x))] \tag{4}
\]

The loss function of the Generative Adversarial Networks (GANs) can be formulated as Equation 4. In Equation 4, the discriminator \( D \) tries to distinguish the generated samples \( G(x) + x \) and the original sample \( x \), so as to encourage the samples generated by \( G \) close to the distribution of the original sample.

However, the perturbation generated by \( G \) may not be able to trigger performance degradation. To fulfill that purpose, we add target AdNN \( f(\cdot) \) into the DeepPerform architecture. While training \( G(\cdot) \), the generated input is fed to AdNN to create an objective function that will help increase the AdNNs’ FLOPs consumption. To generate perturbation that triggers performance degradation in AdNNs, we incorporate two more loss functions other than \( L_{GAN} \) for training \( G(\cdot) \). As shown in Eq.(3), to increase the redundant computation, the first step is to model the function \( g_f(\cdot) \). According to our statistical results in §3, FLOPs could be applied as a hardware-independent metric to approximate AdNNs system performance. Then we model \( g_f(\cdot) \) as Eq.(5).

\[
g_f(x) = \sum_{i=1}^{N} W_i \times I(B_i(x) > T_i) \tag{5}
\]

Where \( W_i \) is the FLOPs in the \( i^{th} \) block, \( B_i(x) \) is the probability that the \( i^{th} \) block is activated, \( I(\cdot) \) is the indicator function, and \( T_i \) is the pre-set threshold based on available computational resources.

\[
L_{adv} = \ell(g_f(x), \sum_{i=1}^{N} W_i) \tag{6}
\]

To enforce \( G \) could generate perturbation that trigger IDPB, we define our performance degradation objective function as Equation 6.

\[
L_{per} = \mathbb{E}_x ||G(x)||_p \tag{7}
\]

To bound the magnitude of the perturbation, we follow the existing work [8] to add a loss of the \( L_p \) norm of the semantic-equivalent perturbation. Finally, our full objective can be denoted as

\[
L = L_{GAN} + \alpha L_{adv} + \beta L_{per} \tag{8}
\]

Where \( \alpha \) and \( \beta \) are two hyper-parameters that balance the importance of each objective. Notice that the goal of the correctness-based testing methods’ objective function is to maximize the errors while our objective function is to maximize the computational complexity. Thus, our objective function in Eq.(8) can not be replaced by the objective function proposed in correctness-based testing [8, 38, 45].

### 4.3 Architecture Details

In this section, we introduce the detailed architecture of the generator and the discriminator. Our generator \( G \) adapts the structure of encoder-decoder, and the architecture of the discriminator is a convolutional neural network. The architectures of the generator and the discriminator are displayed in Fig. 4.

**Generator.** As shown in Fig. 4, there are three main components in the generator, that is, the Encoder, the ResBlocks, and the Decoder. The Encoder repeats the convolutional blocks twice, a convolutional block includes a convolutional layer, a batch normalization layer, and a RELU activation layer. After the encoding process, the input would be smaller in size but with deep channels. The ResBlock stacks four residual blocks [16], which is widely used to avoid the gradient vanishing problem. The Decoder is the reverse process of the Encoder, the transpose convolutional layer is corresponding to the convolutional layer in the Encoder. After the decoding process, the intermediate values will be changed back to the same size as the original input to ensure the generated perturbation to be applied to the original seed input.

**Discriminator.** The architecture of the discriminator is simpler than the generator. There are three convolutional blocks to extract the feature of the input, after that, following a flatten layer and a dense layer for classification.
Algorithm 1: Training DeepPerform

Input: The subject AdNNs $f(\cdot)$ to be tested
Input: Perturbation Constraints $\epsilon$, Perturbation norm $p$
Input: Training dataset $X$
Input: Hyper-parameters $\alpha$, $\beta$
Input: Maximum training epochs $T$

Output: Generator $G$ and Discriminator $D$

1. $g_f() = \text{ModelPerformance}(f) // \text{Construct } g_f \text{ through Equation 5.}$
2. for epoch in range(0, $T$):
   3. for batch in $X$:
      4. $\mathbf{T} = G(x) + x; // \text{generate test samples}$
      5. $\mathbf{T} = \text{CLIP}(\mathbf{T}, x, \epsilon); // \text{clip test samples}$
      6. $L_{GAN} = \text{ComputeGainLoss}(\mathbf{T}, x); // \text{Equation 4}$
      7. $L_{per} = \text{ComputePerLoss}(\mathbf{T}, x); // \text{Equation 7}$
      8. $L_{adv} = \text{ComputeAdvLoss}(\mathbf{T}, x); // \text{Equation 6}$
5. end

6. $g = G \times g_f; D = D + g_D; // \text{Update the weights of } D \text{ and } G$

DeepPerform

The training of DeepPerform is comprised of two parts: training the discriminator $D$ and training the generator $G$. Algorithm 1 explains the training procedure of the DeepPerform. The inputs of our algorithm include the target AdNNs $f(\cdot)$, perturbation constraints $\epsilon$, training dataset $X$, hyper-parameters $\alpha$, $\beta$ and max epochs $T$. The outputs of our training algorithm include a well-trained generator and discriminator. First, the algorithm constructs the performance function $g(\cdot)$ through Equation 5 (Line 1). Then we run $T$ epochs. For each epoch, we iteratively select small batches from the training dataset (Line 2, 3). For each seed $X$ in the selected batches, we generate test sample $T$ and compute the corresponding loss through Eq.(4) (7), (6) (Line 6-8). We compute the gradients of $G$ and $D$ with the computed loss (Line 10, 11), then we update the weights of $G$ and $D$ with the gradients (Line 12). The update process is performed iteratively until the maximum epoch is reached.

5 EVALUATION

We evaluate DeepPerform and answer the following questions:
- **RQ1 (Efficiency):** How efficiently does DeepPerform generate test samples?
- **RQ2 (Effectiveness):** How effective can DeepPerform generate test samples that degraded AdNNs’ performance?
- **RQ3 (Coverage):** Can DeepPerform generate test samples that cover AdNNs’ more computational behaviors?
- **RQ4 (Sensitivity):** Can DeepPerform behave stably under different settings?
- **RQ5 (Quality):** What is the semantic quality of the generated test inputs, and how does it relate to performance degradation?

5.1 Experimental Setup

5.1.1 Experimental Subjects. We select the five subjects used in our preliminary study (§3) as our experimental subjects. As we discussed in §3, the selected subjects are widely used, open-source, and diverse in working mechanisms.

5.1.2 Comparison Baselines. As we mentioned in §2, almost all existing DNN testing work focuses on correctness testing. As far as we know, ILFO [17] is the state-of-the-art approach for generating inputs to AdNNs computational complexity. Furthermore, ILFO has proved that its backward-propagation approach is more effective and efficient than the traditional symbolic execution (i.e., SMT); thus, we compare our method to ILFO. ILFO iteratively applies the backward propagation to perturb seed inputs to generate test inputs. However, the high overheads of iterations make ILFO a time-consuming approach for generating test samples. Instead of iterative backward computation, DeepPerform learns the AdNNs’ computational complexity in the training step. After DeepPerform is trained, DeepPerform applies forward propagation once to generate one test sample.

5.1.3 Experiment Process. We conduct an experiment on the selected five subjects, and we use the results to answer all five RQs. The experimental process can be divided into test sample generation and performance testing procedures.

Test Sample Generation. For each experimental subject, we split train/test datasets according to the standard procedure[25, 36]. Next, we train DeepPerform with the corresponding training datasets. The training is conducted on a Linux server with three Intel Xeon E5-2660 v3 CPUs @2.60GHz, eight 1080Ti Nvidia GPUs, and 500GB RAM, running Ubuntu 14.04. We configure the training process with 100 maximum epochs, 0.0001 learning rate, and apply early-stopping techniques [57]. We set the hyper-parameter $\alpha$ and $\beta$ as 1 and 0.001, as we observe $L_{per}$ is about three magnitude larger than $L_{adv}$. After DeepPerform is trained, we randomly select 1,000 inputs from original test dataset as seed inputs. Then, we feed the seed inputs into DeepPerform to generate test inputs ($x + G(x)$ in Fig. 3) to trigger AdNNs’ performance degradation. In our experiments, we consider both $L_2$ and $L_{inf}$ perturbations [8] and train two version of DeepPerform for input generation. After DeepPerform is trained, we apply the clip operation [28] on $x + G(x)$ to ensure the generated test sample satisfy the semantic constraints in Eq.(3).

Performance Testing Procedure. For the testing procedure, we select Nvidia Jetson TX2 as our main hardware platform (We evaluate DeepPerform on different hardwares in §5.5). Nvidia Jetson TX2 is a popular and widely-used hardware platform for edge computing, which is built around an Nvidia Pascal-family GPU and loaded with 8GB of memory and 59.7GB/s of memory bandwidth. We first deploy the AdNNs on Nvidia Jetson TX2. Next, we feed the generated test samples (from DeepPerform and baseline) to AdNNs, and measure the response latency and energy consumption (energy is measured through Nvidia power monitoring tool). Finally, we run AdNNs at least ten times to infer each generated test sample to ensure the results are accurate.

RQ Specific Configuration. For RQ1, 2 and 3, we follow existing work [1, 17, 33] and set the maximum perturbations as 10 and 0.03 for $L_2$ and $L_{inf}$ norm separately for our approach and baselines. We then conduct experiments in §5.6 to study how different maximum perturbations would affect the performance degradation. ILFO needs to configure maximum iteration number and balance weight, we set the maximum iteration number as 300 and the balance weight as $10^{-6}$, as suggested by the authors [17]. As we discussed in §2, AdNNs require a configurable parameter/threshold to decide the working mode. Different working modes have different tradeoffs between accuracy and computation costs. In our deployment experiments (RQ2), we follow the authors [17] to set the threshold as 0.5 for all the experimental AdNNs, and we evaluate how different
threshold will affect DeepPerform effectiveness in §5.5. Besides that, to ensure the available computational resources are the same, we run only the AdNNs application in the system during our performance testing procedure.

5.2 Efficiency

In this section, we evaluate the efficiency of DeepPerform in generating test samples compared with selected baselines.

**Metrics.** We record the *online time overheads* of the test sample generation process (overheads of running \( \mathcal{G} \) to generate perturbation), and use the mean online time overhead (s) as our evaluation metrics. A lower time overhead implies that it is more efficient, thus better in generating large-scale test samples. Because DeepPerform requires training the generator \( \mathcal{G}(\cdot) \), for a fair comparison, we also evaluate the total time overheads (\( \mathcal{G}(\cdot) \) training + test samples generation) of generating different scale numbers of test inputs.

**Online Overheads.** The average time overheads of generating one test sample are shown in Fig. 5. The results show that DeepPerform costs less than 0.01s to generate a test sample under all experimental settings. In contrast, ILFO requires 27.67-176.9s to generate one test sample. The findings suggest that given same time budget, DeepPerform can generate 3952-22112x more inputs than existing method. Another interesting observation is that the overheads of ILFO fluctuate among different subjects, but the overheads of DeepPerform remain relatively constant. The reason is that the overheads of DeepPerform mainly come from the inference process of the generator, while the overheads of ILFO mainly come from backward propagation. Because backward propagation overheads are proportional to model size (i.e., a larger model demands more backward propagation overheads), the results of ILFO show a significant variation. The overhead of DeepPerform is stable, as its overheads have no relation to the AdNN model size. The result suggests that when testing large models, ILFO will run into scalability issues, whereas DeepPerform will not.

![Figure 5: Online overheads to generate one test sample (s)](image)

**Total Overheads.** The total time overheads of generating various scale test samples are shown in Fig. 6. We can see from the results that ILFO is more efficient than DeepPerform when the number of generated test samples is minimal (less than 200). However, when the number of generated test samples grows greater, the overall time overheads of DeepPerform are significantly lower than ILFO. To create 1000 test samples for SN_C10, for example, ILFO will cost five times the overall overheads of DeepPerform. Because the overhead of ILFO is determined by the number of generated test samples [17], the total overheads quickly rise as the number of generated test samples rises. The main overhead of DeepPerform, on the other hand, comes from the GAN model training instead of test sample generation. As a result, generating various scale numbers of test samples will have no substantial impact on the DeepPerform’s total overheads. The results imply that ILFO is not scalable for testing AdNNs with large datasets, whereas DeepPerform does an excellent job. We also notice that the DeepPerform’s overheads for \( L_2 \) and \( L_{inf} \) are different for DN_SVHN. Because we use the early stopping method [57] to train DeepPerform, we can explain such variation in overheads. In detail, the objective \( L_pere \) differs for \( L_2 \) and \( L_{inf} \). Thus, training process will terminate at different epochs.

5.3 Effectiveness

5.3.1 Relative Performance Degradation. **Metrics.** To characterize system performance, we choose both hardware-independent and hardware-dependent metrics. Our hardware independent metric is floating-point operations (FLOPs). FLOPs are widely used to assess the computational complexity of DNNs [32, 53]. Higher FLOPs indicate higher CPU utilization and lower efficiency performance. As for hardware-dependent metrics, we focus on latency and energy consumption because these two metrics are essential for real-time applications [3, 49]. After characterizing system performance with the above metrics, We measure the increment in the above performance metrics to reflect the severity of performance degradation. In particular, we measure the increased percentage of flops I-FLOPs, latency (I-Latency) and energy consumption (I-Energy) as our performance degradation severity evaluation metrics.

Eq.(9) shows the formal definition of our degradation severity evaluation metrics. In Eq.(9), \( x \) is the original seed input, \( \delta \) is the generated perturbation, and \( F_f(\cdot), L_f(\cdot), E_f(\cdot) \) are the functions that measure FLOPs, latency, and energy consumption of AdNN \( f(\cdot) \). A test sample is more effective in triggering performance degradation if it increases more percentage of FLOPs, latency, and energy consumption. We examine two scenarios for each evaluation metric: the average metric value for the whole test dataset and the maximum metric value caused for a particular sample. The first depicts long-term performance degradation, whereas the second depicts performance degradation under the worst-case situation. We measure the energy consumption using TX2’s power monitoring tool [37].

\[
I - \text{FLOPs}(x) = \frac{F_f(x + \delta) - F_f(x)}{F_f(x)} \times 100%
\]
\[
I - \text{Latency}(x) = \frac{L_f(x + \delta) - L_f(x)}{L_f(x)} \times 100%
\]
\[
I - \text{Energy}(x) = \frac{E_f(x + \delta) - E_f(x)}{E_f(x)} \times 100%
\]

The hardware-independent experimental results are listed in Table 4. As previously stated, greater I-FLOPs implies that the created test samples demand more FLOPs, which will result in significant system performance reduction. The table concludes that DeepPerform generates test samples that can cause more severe performance degradation. Other than that, we have multiple observations. First, for four of the five subjects, DeepPerform generates test samples that require more FLOPs, e.g., 31.14%-62.77% for SN_C10. Second, for both \( L_2 \) and \( L_{inf} \) perturbation, the model would require more FLOPs, and the difference between \( L_2 \) and \( L_{inf} \) is more efficient than DeepPerform.
Table 4: The FLOPs increment of the test samples (%)

<table>
<thead>
<tr>
<th>Norm</th>
<th>Subject</th>
<th>Mean FLOPs</th>
<th>Max FLOPs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>baseline / ours</td>
<td>baseline / ours</td>
</tr>
<tr>
<td>(L_{\infty})</td>
<td>SN_C10</td>
<td>6.43</td>
<td>31.14</td>
</tr>
<tr>
<td></td>
<td>BD_C10</td>
<td>48.44</td>
<td>38.39</td>
</tr>
<tr>
<td></td>
<td>RN_C100</td>
<td>133.67</td>
<td>181.57</td>
</tr>
<tr>
<td></td>
<td>DS_C100</td>
<td>116.19</td>
<td>157.66</td>
</tr>
<tr>
<td></td>
<td>DS_SVHN</td>
<td>115.99</td>
<td>228.32</td>
</tr>
<tr>
<td>(L_2)</td>
<td>SN_C10</td>
<td>20.34</td>
<td>31.30</td>
</tr>
<tr>
<td></td>
<td>BD_C10</td>
<td>48.44</td>
<td>38.39</td>
</tr>
<tr>
<td></td>
<td>RN_C100</td>
<td>133.67</td>
<td>181.57</td>
</tr>
<tr>
<td></td>
<td>DS_C100</td>
<td>116.19</td>
<td>157.66</td>
</tr>
<tr>
<td></td>
<td>DS_SVHN</td>
<td>115.99</td>
<td>228.32</td>
</tr>
</tbody>
</table>

Table 5 lists similar results for hardware-dependent experiments, DeepPerf outperforms ILFO in 65 out of 80 comparison scenarios. However, for the other 15 comparisons, we explain the results as the following two reasons: (i) the system noise has affected the results because for almost all scenarios DeepPerf has been able to increase more FLOPs than ILFO, (ii) recall in Table 2, RN_C100 has the the PCCs around 0.64, and the FLOPs increment of RN_C100 for DeepPerf and 1LFO is around the same level. Thus, DeepPerf may cause slightly less latency and energy consumption degradation than ILFO. However, for SN_C10, although it has low PCCs, DeepPerf can increase much more FLOPs than ILFO, thus, DeepPerf can cause more severe performance degradation. Based on the results in Table 5, we conclude that DeepPerf outperforms baseline in creating inputs that consume higher energy or time.

5.3.2 Absolute Performance Degradation. Besides the relative performance degradation, we also investigate the absolute performance degradation of the generated inputs. In Figure 7, we plot the unnormalized efficiency distribution (i.e., FLOPs, latency, energy consumption) of both seed and generated inputs to characterize the absolute performance degradation. We specifically depict the probability distribution function (PDF) curve of each efficiency metric under discussion. The unnormalized efficiency distribution is shown in Fig. 7, where the green curve is for the seed inputs, and the red curve is for the test inputs from DeepPerf. From the results, we observe that DeepPerf is more likely to generate test inputs located at the right end of the x-axis. Recall that a PDF curve with more points on the right end of the x-axis is more likely to generate theoretical worst-case test inputs. The results confirm that DeepPerf is more likely to generate test inputs with theoretical worst-case complexities.

5.3.3 Test Sample Validity. To measure the validity of the generated test samples, we define degradation success number \(\eta\) in Eq. (10),

\[
\eta = \sum \mathbb{1}(\text{FLOPs}(x + \delta) \geq \text{FLOPs}(x)), \quad x \in X
\]

where \(X\) is the set of randomly selected seed inputs and \(\mathbb{1}(\text{FLOPs}(x + \delta) > \text{FLOPs}(x))\) indicates whether generated test samples require more computational resources than the seed inputs. We run DeepPerf and baselines the same experimental time and generate the different number of test samples \(X\) in Eq. (10), we then measure \(\eta\) in the generated test samples. For convince, we set the experimental time as the total time of DeepPerf generating 1,000
test samples (same time for ILFO). From the third column in Table 6, we observe that for most experimental settings, DeepPerform achieves a higher degradation success number than ILFO. Although ILFO is an end-to-end approach, the high overheads of ILFO disable it to generate enough test samples.

5.4 Coverage

Table 6: Validity and coverage results

<table>
<thead>
<tr>
<th>Norm</th>
<th>Subject</th>
<th>ϱ (%) ours / baseline</th>
<th>Cov (%) ours / baseline</th>
<th>L2</th>
</tr>
</thead>
<tbody>
<tr>
<td>SN_C10</td>
<td>842</td>
<td>0.74 ± 0.001</td>
<td>0.65 ± 0.001</td>
<td>79.17</td>
</tr>
<tr>
<td>BD_C10</td>
<td>630</td>
<td>0.37 ± 0.001</td>
<td>0.37 ± 0.001</td>
<td>79.17</td>
</tr>
<tr>
<td>RN_C100</td>
<td>871</td>
<td>0.59 ± 0.002</td>
<td>0.89 ± 0.030</td>
<td>79.17</td>
</tr>
<tr>
<td>DS_C100</td>
<td>646</td>
<td>1.00 ± 0.000</td>
<td>0.83 ± 0.016</td>
<td>79.17</td>
</tr>
<tr>
<td>DS_SVHN</td>
<td>916</td>
<td>1.00 ± 0.000</td>
<td>0.92 ± 0.033</td>
<td>79.17</td>
</tr>
</tbody>
</table>

In this section, we investigate the comprehensiveness of the generated test inputs. In particular, we follow existing work [38, 58] and investigate the diversity of the AdNN behaviors explored by the test inputs generated by DeepPerform. Because AdNNs’ behavior relies on the computation of intermediate states [32, 38], we analyze how many intermediate states are covered by the test suite.

\[
\text{Cov}(X) = \frac{\sum_{x \in X} \sum_{i=1}^{N} \mathbb{I}(B_i(x) > \tau_i)}{N}
\]

(11)

To measure the coverage of AdNNs’ intermediate states, we follow existing work [38] and define decision block coverage (\text{Cov}(X)) in Eq.(11), where \(N\) is the total number blocks, \(\mathbb{I}(\cdot)\) is the indicator function, and \(B_i(x) > \tau_i\) represents whether the \(i^{th}\) block is activated by input \(x\) (the definition of \(B_i\) and \(\tau_i\) are the same with Eq.(1) and Eq.(2)). Because AdNNs activate different blocks for decision making, then a higher block coverage indicates the test samples cover more decision behaviors. For each subject, we randomly select 100 seed samples from the test dataset as seed inputs. We then feed the same seed inputs into DeepPerform and ILFO to generate test samples. Finally, we feed the generated test samples to AdNNs and measure block coverage. We repeat the process ten times and record the average coverage and the variance. The results are shown in Table 6 last two columns. We observe that the test samples generated by DeepPerform achieve higher coverage for almost all subjects.

5.5 Sensitivity

In this section, we conduct two experiments to show that DeepPerform can generate effective test samples under different settings.

Configuration Sensitivity. As discussed in 5.2, AdNNs require configuring the threshold \(\tau_i\) to set the accuracy-performance tradeoff mode. In this section, we evaluate whether the test samples generated from DeepPerform could degrade the AdNNs’ performance under different modes. Specifically, we set the threshold \(\tau_i\) in Eq.(1) and Eq.(2) as 0.3, 0.4, 0.5, 0.6, 0.7 and measure the maximum FLOPs increments. Notice that we train DeepPerform with \(\tau_i = 0.5\) and test the performance degradation with different \(\tau_i\). The maximum FLOPs’ increment ratio under different system configurations are listed in Table 7. For all experimental settings, the maximum FLOPs’ increment ratio keeps a stable value (e.g., 79.17-82.91, 175.59-250.00). The results imply that the test samples generated by DeepPerform can increase the computational complexity under different configurations, and the maximum FLOPs’ increment ratio is stable as the configuration changes.

Table 7: Increment under different thresholds

<table>
<thead>
<tr>
<th>Norm</th>
<th>Subject</th>
<th>Threshold 0.3</th>
<th>Threshold 0.4</th>
<th>Threshold 0.5</th>
<th>Threshold 0.6</th>
<th>Threshold 0.7</th>
</tr>
</thead>
<tbody>
<tr>
<td>L2</td>
<td>SN_C10</td>
<td>79.17</td>
<td>82.91</td>
<td>82.91</td>
<td>75.00</td>
<td>70.00</td>
</tr>
<tr>
<td>BD_C10</td>
<td>250.00</td>
<td>250.00</td>
<td>175.59</td>
<td>175.59</td>
<td>175.59</td>
<td>175.59</td>
</tr>
<tr>
<td>RN_C100</td>
<td>500.00</td>
<td>498.99</td>
<td>498.99</td>
<td>498.99</td>
<td>498.99</td>
<td>498.99</td>
</tr>
<tr>
<td>DS_C100</td>
<td>600.00</td>
<td>600.00</td>
<td>552.00</td>
<td>552.00</td>
<td>552.00</td>
<td>552.00</td>
</tr>
<tr>
<td>DS_SVHN</td>
<td>498.29</td>
<td>498.29</td>
<td>498.29</td>
<td>498.29</td>
<td>498.29</td>
<td>498.29</td>
</tr>
</tbody>
</table>

Hardware Sensitivity. We next evaluate the effectiveness of our approach on different hardware platforms. In particular, we select Intel Xeon E5-2660 V3 CPU and Nvidia 1080 Ti as our experimental hardware platforms and measure the maximum performance degradation ratio on those selected platforms. The test samples generated by DeepPerform, as shown in Table 8, cause severe and stable runtime performance degradation on different hardware platforms. As a result, we conclude that DeepPerform is not sensitive to hardware platforms.
5.6 Quality

We first conduct quantitative evaluations to evaluate the similarity between the generated and seed inputs. In particular, we follow existing work [8] and compute the perturbation magnitude. The perturbation magnitude are listed in Table 9. Recall that we follow existing work [8, 17] and set the perturbation constraints \( \epsilon \) as 10 and 0.03 for \( L_2 \) and \( L_{inf} \) norm (§S.1). From the results in Table 9, we conclude that generated test samples can satisfy the semantic-equivalent constraints in Eq.(3). Moreover, we conduct a qualitative evaluation. In particular, we randomly select seven images from the generated images for RA_C100 and visualize them in Fig. 9 (more results are available on our website), where the first row is the randomly selected seed inputs, and the second row is the corresponding generated inputs. The visualization results show that the test inputs generated by DeepPerfom are semantic-equivalent to the seed inputs. Furthermore, we investigate the relationship between different semantic-equivalent constraints and performance degradation. We first change the perturbation magnitude constraints (i.e., \( \eta \) in Eq.(3)) and train different models (experiment results for \( L_2 \) norm could be found on our websites). After that, we measure the severity of AdNN performance degradation under various settings. Fig. 8 shows the results. We observe that although the relationship between performance degradation ratio and perturbation magnitude is not purely linear, there is a trend that the overhead increases with the increase of perturbation magnitude.

### 6 APPLICATION

This section investigates if developers can mitigate the performance degradation bugs using the existing methods for mitigating DNN correctness bugs (i.e., adversarial examples). We focus on two of the most widely employed approaches: offline adversarial training [15], and online input validation [50]. Surprisingly, we discover that not all of the two approaches can address performance faults in the same manner they are used to repair correctness bugs.

#### 6.1 Adversarial Training

**Setup.** We follow existing work [15] and feed the generated test samples and the original model training data to retrain each AdNN. The retraining objective can be formulated as

\[
\mathcal{L}_{\text{retrain}} = \ell(g_f(x'), f(x)) + \lambda \ell(f(x), y) + \ell(f(x'), y)
\]

where \( x \) is one seed input in the training dataset, \( x' = G(x) + \delta \) is the generated test input, \( f(\cdot) \) is the AdNNs, and \( g_f(\cdot) \) measures the AdNNs computational FLOPs. Our retraining objective can be interpreted as forcing the buggy test inputs \( x' \) to consume the same FLOPs as the seed one (i.e., \( \mathcal{L}_1 \)), while producing the correct results (i.e., \( \mathcal{L}_2 \)). For each AdNN model under test, we retrain it to minimize the objective in Eq.(12). After retraining, we test each AdNNs accuracy and efficiency on the hold-out test dataset.

**Results.** Table 10 shows the results after model retraining. The left two columns show the performance degradation before and after retraining.
model retraining, while the right two columns show the model accuracy before and after model retraining. The findings show that following model training, the I-FLOPs fall; keep in mind that a higher I-FLOPs signifies a more severe performance degradation. Thus, the decrease in I-FLOPs implies that model retraining can help overcome performance degradation. However, based on the data in the right two columns, we observe that such retraining, different from accuracy-based retraining, may harm model accuracy.

### 6.2 Input Validation

Input validation [50] is a runtime approach that filters out abnormal inputs before AdNNs cast computational resources on such abnormal inputs. This approach is appropriate for situations where the sensors (e.g., camera) and the decision system (e.g., AdNN) work at separate frequencies. Such different frequency working mode is very common in robotics systems [13, 31, 60], where the AdNN system will randomly select one image from continuous frames from sensors since continuous frames contain highly redundant information. Our intuition is to filter out those abnormal inputs at the early computational stage, the same as previous work [50].

**Design of Input Filtering Detector.** Our idea is that although seed inputs and the generated test inputs look similar, the latent representations of these two category inputs are quite different [50]. Thus, we extract the hidden representation of a given input by running the first convolutional layer of the AdNNs. First, we feed both benign and DeepPerform generated test inputs to specific AdNN. We use the outputs of the first convolutional layer as input to train a linear SVM to classify benign inputs and inputs that require huge computation. If any resource consuming adversarial input is detected, the inference is stopped. The computational complexity of the SVM detector is significantly less than AdNNs. Thus the detector will not induce significant computational resources consumption.

**Setup.** For each experimental subject, we randomly choose 1,000 seed samples from the training dataset, and apply DeepPerform to generate 1,000 test samples. We use these 2,000 inputs to train our detector. To evaluate the performance of our detector, we first randomly select 1,000 inputs from the test dataset and apply DeepPerform to generate 1,000 test samples. After that, we run the trained detector on such 2,000 inputs and measure detectors’ AUC score, extra computation overheads, and energy consumption.

**Results.** Table 11 shows that the trained SVM detector can successfully detect the test samples that require substantial computational resources. Specifically for $L_2$ norm perturbation, all the AUC scores are higher than 0.99. The results indicate that our detector identifies $L_2$ test samples better. The last four columns show the extra computational resources consumption of the detector. We observe that the detector does not consume many additional computational resources from the results.

### 7 THREATS TO VALIDITY

Our selection of five experimental subjects might be the external threat that threaten the generability of our conclusions. We alleviate this threat by the following efforts. (1) We ensure that the datasets are widely used in both academia and industry research. (2) All evaluated models are state-of-the-art DNN models (published in top-tier conferences after 2017). (3) Our subjects are diverse in terms of a varied set of topics: all of our evaluated datasets and models differ from each other in terms of different input domains (e.g., digit, general object recognition), the number of classes (from 10 to 100), the size of the training dataset (from 50,000 to 73,257), the model adaptive mechanism. Our internal threat mainly comes from the realism of the generated inputs. We alleviate this threat by demonstrating the relationship of our work with existing work. Existing work [26, 58, 59] demonstrates that correctness-based test inputs exist in the physical world. Because we formulate our problem (i.e., the constraint in Eq.(3)) the same as the previous correctness-based work [33, 59], we conclude our generated test samples are real and exist in the physical world.

### 8 RELATED WORKS

**Adversarial Examples & DNN Testing.** Adversarial Examples have been used evaluate the robustness of DNNs. These examples are fed to DNNs to change the prediction of the model. Szegedy et al. [42] and Goodfellow et al. [15] propose adversarial attacks on DNNs. Karron et al. Adversarial attacks have been extended to various fields like natural language and speech processing [7, 22], and graph models [4, 61]. Although, all these attacks focus on changing the prediction and do not concentrate on performance testing. Several testing methods have been proposed to test DNNs [10, 11, 58, 59].

**Performance Testing.** Runtime performance is a critical property of software, and a branch of work has been proposed to test software performance. For example, Netperf [18] and IOZone [48] evaluate the performance of different virtual machine technologies. WISE [6] proposes a method to generate test samples to trigger worst-case complexity. SlowFuzz [39] proposes a fuzzing framework to detect algorithmic complexity vulnerabilities. PerfFuzz [27] generates inputs that trigger pathological behavior across program locations.

### 9 CONCLUSION

In this paper, we propose DeepPerform, a performance testing framework for DNNs. Specifically, DeepPerform trains a GAN to learn and approximate the distribution of the samples that require more computational units. Through our evaluation, we have shown that DeepPerform is able to find IDPB in AdNNs more effectively and efficiently than baseline techniques.

### ACKNOWLEDGMENTS

This work was partially supported by Siemens Fellowship and NSF grant CCF-2146443.

### REFERENCES


