

# Exposing Previously Undetectable Faults in Deep Neural Networks

Isaac Dunn  
University of Oxford

Hadrien Pouget  
University of Oxford

Tom Melham  
University of Oxford

Daniel Kroening\*  
Amazon, Inc.

## ABSTRACT

Existing methods for testing DNNs solve the oracle problem by constraining the raw features (e.g. image pixel values) to be within a small distance of a dataset example for which the desired DNN output is known. But this limits the kinds of faults these approaches are able to detect. In this paper, we introduce a novel DNN testing method that is able to find faults in DNNs that other methods cannot. The crux is that, by leveraging generative machine learning, we can generate fresh test cases that vary in their high-level features (for images, these include object shape, location, texture, and colour). We demonstrate that our approach is capable of detecting deliberately-injected faults as well as new faults in state-of-the-art DNNs, and that in both cases, existing methods are unable to find these faults.

## CCS CONCEPTS

• Computing methodologies → Machine learning.

## KEYWORDS

Deep Learning, Generative Adversarial Networks, Image Classification, Robustness.

### ACM Reference Format:

Isaac Dunn, Hadrien Pouget, Tom Melham, and Daniel Kroening. 2021. Exposing Previously Undetectable Faults in Deep Neural Networks. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2021)*. ACM, New York, NY, USA, 11 pages. <https://doi.org/10.1145/nnnnnnnn.nnnnnnnn>

## 1 INTRODUCTION

As Deep Neural Networks (DNNs) begin to be deployed in safety-critical and mission-critical contexts, it becomes important to have confidence they are fit for purpose. As with conventional software or hardware testing, evaluating a DNN by checking its performance on an adequate suite of test inputs is a central approach for revealing flaws present in the system. Existing approaches generally

focus on creating new test inputs by perturbing existing inputs. Methods for doing this with images include making small, undetectable changes to the pixels of an image [16, 26], using specific transformations such as translations or rotations [7, 8, 20, 46], or adding noise and blur affects [17]. Beyond making changes directly to the pixels of an image, more sophisticated methods leverage abstract representations of an input’s features to make interesting changes [5, 27, 28].

Testing DNNs presents a slew of new challenges, relating to both the black-box nature of DNNs and their inherent design as approximations trained from finite data [44]. A particular challenge when testing DNNs is that there is little sense of what properties the trained system is required to satisfy, and so it is not always clear when a test result indicates a violation of a required property.

In fact, this specification problem is difficult even for the simplest imaginable specification; making the correct prediction on a given input. Not knowing the intended output for each input, the *oracle problem*, is the reason DNNs are so valuable in the first place. They are designed to generalise from labelled data to make predictions on data for which *we do not know the intended output*. However, this makes testing them beyond the relatively small amount of labelled data we have difficult. The simplest approach to testing DNNs does not solve the oracle problem at all, instead reserving a set of manually-labelled data to use solely for testing. In this case, the only property that can be evaluated is the accuracy on the original task, that is on data from exactly the same source as the training data. Existing methods that generate new test cases for DNNs *do* solve the oracle problem. The solution is to restrict generated test inputs to be sufficiently similar to a manually-labelled example that we can be confident that the desired system output is the same. Most methods take one of two options. First, and most commonly, test inputs are constrained to have their raw features (e.g. pixel values) differ no more than some fixed distance  $\epsilon$  under an  $\ell_p$  metric, typically  $\ell_2$  or  $\ell_\infty$ . Second, test inputs may differ in only specific hand-specified ways from a manually-labelled example. For instance, by adding artificial fog or other corruptions [17], by modifying the brightness or contrast of the input, or by adding black squares [34].

However, by introducing such constraints to circumvent the oracle problem, existing approaches significantly limit the properties that they are able to evaluate. For example, if generated test inputs are constrained to be within an  $\ell_\infty$  distance  $\epsilon$  of known labelled examples, then the tests are only able to expose faults that fail to respect the invariant that changes of up to  $\epsilon$  for each pixel should not change the system output.

\*The work reported in this paper was done prior to joining Amazon.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

ISSTA 2021, 12–16 July, 2021, Aarhus, Denmark

© 2021 Association for Computing Machinery.

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...\$15.00

<https://doi.org/10.1145/nnnnnnnn.nnnnnnnn>

In this paper, we introduce a testing method that is able to produce a much larger variety of tests than existing methods. While most existing methods can test for invariance to  $\ell_p$   $\epsilon$ -bounded changes to pixel values, or to certain hand-coded artificial feature changes, our new approach is able to test for invariance to changes to higher-level features such as position, colour, and texture of objects. To do so, we leverage generative adversarial networks [13], which have been trained to encode the wide range of natural variation of features present in the data.

We demonstrate that our method is able to identify particular faults that we intentionally create in DNNs, and show that existing methods are unable to detect these. We also apply our method to state-of-the-art image classification DNNs, and again demonstrate that it is able to find faults that cannot be found using existing approaches.

## 2 PRELIMINARIES

Generative adversarial networks (GANs) [13] are a class of generative machine learning models involving the simultaneous training of two deep neural networks: a generator  $g$  and a discriminator  $d$ . Specifically, given a dataset  $D \subseteq \mathbb{X}$  of samples drawn from a probability distribution  $p_D$ , the generator  $g : \mathbb{Z} \rightarrow \mathbb{X}$  learns to transform random noise  $z$  drawn from a standard distribution  $p_z$  (typically Gaussian) into an approximation of  $p_D$ . The discriminator network  $d : \mathbb{X} \rightarrow \mathbb{R}$  learns to predict whether a given example  $x$  is drawn from the data distribution  $p_D$  or was generated by  $g$ . The generator and the discriminator are *adversarial* because they train simultaneously, with each being rewarded for out-performing the other. That is, while the outputs of both are initially random, the discriminator over time learns to identify features that differ between the trained and generated data, which then allows the generator to improve by adjusting that feature of its generated data to match the training distribution. Goodfellow’s tutorial [12] can be consulted for precise details.

A conditional GAN [31] is a variant that learns to generate samples from a conditional distribution by simply passing the intended label  $y$  for the generated image to both the generator and the discriminator, and training the generator to maximise the log-likelihood of the correct label in addition to optimising its usual objective. That is, a *labelled* dataset  $D \subseteq \mathbb{X} \times \mathbb{Y}$  must be used during training, the discriminator  $d : \mathbb{X} \times \mathbb{Y} \rightarrow \mathbb{R}$  learns to discriminate between labelled dataset and generated examples, and the generator  $g : \mathbb{Z} \times \mathbb{Y} \rightarrow \mathbb{X}$  learns to generate images with the specified label  $y \in \mathbb{Y}$ .

Generative adversarial networks have one important property which makes them especially suitable for test generation for image classifiers: they are able to learn to generate crisp high-quality examples as though sampled from a relatively complex training distribution [4]. However, there are other approaches to training generative DNNs—for instance as an Variational Auto-Encoder (VAE) [25]—and these would be perfectly suitable replacements for a GAN-trained generator, if their performance were satisfactory.

Generative networks have been found to display an interesting property: different layers, and even different neurons, encode different kinds of features of the image. Earlier layers tend to encode higher-level information about objects in the image, whereas later

layers deal more with “low-level materials, edges, and colours” [1, p.7]. This makes sense: the great promise of DNNs is their ability to automatically construct hierarchies of feature representations. In addition, by moving the input to the generator in a straight line, features such as zoom and object position and rotation can vary in the image generated as its output [22]. State-of-the-art GANs are particularly able to smoothly and convincingly interpolate between different images by so adjusting the input to the generator [4]. We will leverage these meaningful latent feature representations when generating new test cases.

## 3 OUR METHOD

The crux of our method is that by using a generative model to *generate* fresh test data, rather than simply performing small adjustments to existing test inputs, it is possible to evaluate whether a DNN behaves as required in response to variance in features that vary naturally in the training data.

### 3.1 Problem Setup

Suppose we have a set of possible system inputs  $\mathbb{X}$ , a set of discrete labels (system outputs)  $\mathbb{Y}$ , and an oracle  $O : \mathbb{X} \rightarrow \mathbb{Y}$  that assigns to each system input  $x$  its ‘correct’ output,  $O(x)$ . When working with images, the input space is RGB pixel space  $\mathbb{X} = \mathbb{R}^{c \times w \times h}$ , where  $c$  is the number of colour channels (typically  $c = 3$ ), and  $w$  and  $h$  are the width and height respectively of the image, in pixels. For the task of object recognition, in which the system is required to identify which of  $k$  possible objects is depicted in an image, the set of labels  $\mathbb{Y} = \{1, 2, \dots, k\}$  corresponds to the  $k$  possible object classes.

Given a set of  $N$  labelled datapoints  $D = (x_i, O(x_i))_{i=1}^N \subset \mathbb{X} \times \mathbb{Y}$ , we can train a DNN image classifier  $f : \mathbb{X} \rightarrow \mathbb{R}^{|\mathbb{Y}|}$  that attempts to approximate  $O$ . Given an input,  $f$  outputs a real-valued confidence for each possible class  $y \in \mathbb{Y}$ . Let  $f_y(x)$  be the classifier’s confidence of input  $x$  being of class  $y$ , and  $f_{pred}(x) = \arg \max_y f_y(x)$ , such that  $f_{pred}$  is an approximation of  $O$ . Typically, the final layer of a DNN is a softmax function, so that for all output confidences  $f_y(x)$ ,  $0 \leq f_y(x) \leq 1$ , and so that  $\sum_{y=1}^k f_y(x) = 1$ .

When testing a trained DNN, our task is to select test inputs  $x \in \mathbb{X}$ . In particular, the goal is to identify test cases that *fail*, because these are indicative of a fault in the DNN.

**Definition 3.1.** A test case with test input  $x \in \mathbb{X}$  for DNN  $f : \mathbb{X} \rightarrow \mathbb{R}^{|\mathbb{Y}|}$  is said to *fail* if  $f_{pred}(x) \neq O(x)$ .

However, we quickly run into the test oracle problem: we do not have direct access to  $O$ —if we did, there would be no need to train an approximation  $f$ —and it is too costly to seek human labelling for each test input. So a practical test generation method needs to provide not only test inputs  $x$ , but additionally the desired system output  $O(x)$  so that failing test cases can be identified.

### 3.2 Solving the test oracle problem

The standard approach to solving the test oracle problem is to make use of the set  $D$  of labelled data that is available. We can partition  $D$  into a large training set  $D_{train}$  and a small holdout test set  $D_{test}$ ; by using only  $D_{train}$  during training, we can be confident that the DNN has not overfit to any of the examples in  $D_{test}$ , so these examples can be used during testing. For a test case  $(x_{test}, O(x_{test})) \in D_{test}$ ,

any new input  $x_{new} \in \mathbb{X}$  that we can be confident shares the same desired output as  $x_{test}$  can therefore be used as a new test case, because we simply assume that  $O(x_{new}) = O(x_{test})$ .

To identify such  $x_{new}$  that share a label with a known test case, most methods begin by choosing a perturbation function  $t : \mathbb{X} \times \mathbb{P} \rightarrow \mathbb{X}$  with parameter space  $\mathbb{P}$ . The intention is that, given a labelled test case  $x_{test}$ , this function is able to generate new test cases as its parameter varies:  $x_{new} = t(x_{test}, p)$ . But to be confident that  $x_{new}$  is similar enough to  $x_{test}$  to have the same true label, we must also introduce a similarity constraint over the parameter  $p$ .

**Definition 3.2.** A similarity constraint  $d$  for a perturbation function  $t$  is a function  $d : \mathbb{P} \rightarrow \{\top, \perp\}$ , such that for all  $x \in \mathbb{X}$ , if  $d(p) = \top$  then  $O(x_{test}) = O(t(x, p))$ .

If we have a suitable perturbation function and similarity constraint, then the problem of identifying test cases reduces to a search for suitable parameter values  $p$ :

**PROPOSITION 3.3.** Given a labelled test case  $(x_{test}, O(x_{test})) \in D_{test}$ , a perturbation function  $t : \mathbb{X} \times \mathbb{P} \rightarrow \mathbb{X}$ , a similarity constraint  $d : \mathbb{P} \rightarrow \{\top, \perp\}$ , and a parameter  $p \in \mathbb{P}$ , if  $d(p) = \top$  and  $f_{pred}(t(x_{test}, p)) \neq O(x_{test})$ , then  $t(x_{test}, p)$  is a failing test case.

**3.2.1 Test Generation using Pixel-Space Perturbations.** Most existing methods that generate tests for DNNs use a pixel-space perturbation approach. We have that  $\mathbb{P} = \mathbb{X}$  and  $t(x, p) = x + p$ . In effect,  $t$  simply changes each pixel value in the image independently. The similarity constraint used typically constrains the magnitude of  $p$ : if the pixel values do not change too much, then the label should remain the same. This magnitude is typically measured using the  $\ell_{inf}$  or  $\ell_2$  norm metric. That is,  $d(p) = \|p\|_2 \leq \epsilon$  or  $d(p) = \|p\|_{inf} \leq \epsilon$ , for a manually chosen constant  $\epsilon$  small enough that the change is nearly imperceptible.

Given a labelled test case  $(x_{test}, O(x_{test})) \in D_{test}$ , then, a pixel perturbation method must find a suitable  $p$ . This is almost always done using an optimisation over  $p$ , using a loss function chosen to be minimised when  $f_{pred}(t(x_{test}, p)) \neq O(x_{test})$  and  $d(p) = \top$ . Since DNNs are differentiable, the derivative of the loss function with respect to  $p$  can be computed, and this gradient can be walked to minimise the loss and thereby identify a failing test case. If additional properties are desired of the test cases, this can be reflected in the choice of loss function.

### 3.3 Using GANs to perturb images

In this paper, as in existing methods, we use a perturbation-based approach to solve the test oracle problem. However, rather than directly perturbing the pixels of a labelled test dataset image, there are two important differences:

- (1) Instead of beginning with a labelled test dataset image, we use a conditional generative network to *generate* a fresh test seed for which we know the correct label.
- (2) Instead of perturbing the individual pixel values of this seed, we make perturbations to meaningful features of the input by exploiting the generative network's learnt features.

**3.3.1 Generating Test Seeds.** Suppose we have a pretrained conditional generator network  $g : \mathbb{Z} \times \mathbb{Y} \rightarrow \mathbb{X}$ , which as described in Section 2, takes a normally sampled  $z \in \mathbb{Z}$  and a class label

$y \in \mathbb{Y}$  and returns an image  $g(z, y) \in \mathbb{X} = \mathbb{R}^{c \times w \times h}$  such that  $O(g(z, y)) = y$ . Now, rather than relying on the finite examples in a test dataset as our source of seeds from which to create test cases, we can generate fresh labelled test seeds on demand, by sampling new generator inputs  $z$ .

While this may be valuable in itself, our main intention in using generated images is that it allows much greater control over the test cases we create.

**3.3.2 Making Perturbations.** Because the generator  $g$  is a deep neural network, it can be described as a sequence of  $n$  layers. By writing the  $i$ th layer as a function between layer outputs  $g_i : \mathbb{O}_{i-1} \rightarrow \mathbb{O}_i$ , where  $\mathbb{O}_0 = \mathbb{Z}$  for convenience and  $\mathbb{O}_n = \mathbb{R}^{c \times w \times h}$ , we can write  $g$  as a function composition:  $g = g_n \circ g_{n-1} \circ \dots \circ g_1$ .

The thrust of our method is to introduce a perturbation function that perturbs high-level features rather than individual pixel values. Since the neurons in a generative network encode meaningful features [1], we take after Dunn et al. [5] and perturb the activations of these neurons so as to adjust the features of the generated image in a context-sensitive way. That is, rather than using a perturbation parameter space  $\mathbb{P} = \mathbb{X}$ , our parameter space allows adjustments at the output of multiple layers in the generator:  $\mathbb{P} = \mathbb{O}_0 \times \mathbb{O}_1 \times \dots \times \mathbb{O}_n$ .

Then we define our perturbation function

$$t(g(z, y), p) = (g'_n \circ g'_{n-1} \circ \dots \circ g'_1)(z, y, p), \quad (1)$$

where  $g'_i(o_{i-1}, p_{i-1}) = g_i(o_{i-1}) + p_{i-1}$ . That is, at each layer in the generator,  $t$  simply performs element-wise addition of the parameter tensor with the layer output. Our similarity constraint measures the total size of the changes being made to the activations:  $d(p) = \|\bar{p}\|_2 < \epsilon$ , where  $\bar{p}$  is the one-dimensional vector formed by flattening all the elements of  $p_0, p_1, \dots, p_n$  into a list.


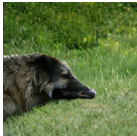


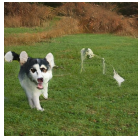













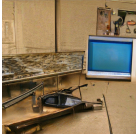






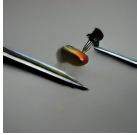





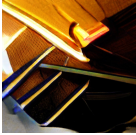












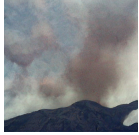



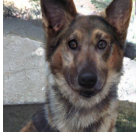
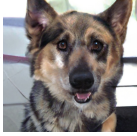

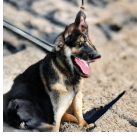


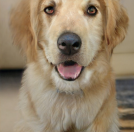




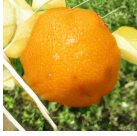




For our experiments, we only optimise over the first six layers of the 18-layer generator, since the earlier layers encode high-level, human-intelligible features [1].

**3.3.3 Finding a Suitable Perturbation.** As before, given a labelled test seed, in this case  $(g(z, y), y)$ , and our perturbation function  $t$  and similarity constraint  $c$ , finding a value  $p \in \mathbb{P}$  suffices to produce a failing test case for the DNN under test,  $f$ . We use a gradient-walking optimisation to find such satisfactory values of  $p$ . In particular, we use the loss function  $L(p) = \max_y f_{pred}(t(g(z, y), p))_y$ , which penalises confidence in the DNN's top output class. It is also possible to include a penalty on  $\|\bar{p}\|_2$ , but in practice we found this to be unnecessary: by starting with every element of  $p$  set to 0, the gradient walk increases  $\|\bar{p}\|_2$  slowly enough that it is acceptably small when a suitable  $p$  is found. Note that the usual backpropagation algorithm is sufficient to compute gradients of  $L$  with respect to  $p$  since  $f, t$  and each layer of  $g$  are differentiable.

**3.3.4 Confident Targeted Failing Tests.** So far, we have considered *untargeted* tests, in which the goal is to change the classifier's output to any other class. For our purposes, we will in fact prefer to generate *confident, targeted* tests. A confident test case requires a certain confidence in the incorrect classification, and a targeted test case is one that requires a specified incorrect label to be output.

**Definition 3.4.** A *confident* test case  $x \in \mathbb{X}$  for DNN  $f : \mathbb{X} \rightarrow \mathbb{R}^{|\mathbb{Y}|}$  is said to fail with confidence margin  $c > 0$  if  $\max_y f_y(x) - f_{O(x)}(x) > c$ .

**Table 1: Examples of tests for DNNs with deliberately injected flaws. To the left of each arrow is the generated test seed which is correctly classified; to the right of each arrow is the generated test case that is incorrectly classified. Inspection of these test cases indicates that the DNNs are relying on the injected fault features (refer to Table 2).**

#	Tests that indicate the fault $y_0 \Rightarrow y_1$						Tests that indicate the fault $y_1 \Rightarrow y_0$					
1		$\Rightarrow$			$\Rightarrow$			$\Rightarrow$			$\Rightarrow$	
2		$\Rightarrow$			$\Rightarrow$			$\Rightarrow$			$\Rightarrow$	
3		$\Rightarrow$			$\Rightarrow$			$\Rightarrow$			$\Rightarrow$	
4		$\Rightarrow$			$\Rightarrow$			$\Rightarrow$			$\Rightarrow$	
5		$\Rightarrow$			$\Rightarrow$			$\Rightarrow$			$\Rightarrow$	
6		$\Rightarrow$			$\Rightarrow$			$\Rightarrow$			$\Rightarrow$	
7		$\Rightarrow$			$\Rightarrow$			$\Rightarrow$			$\Rightarrow$	
8		$\Rightarrow$			$\Rightarrow$			$\Rightarrow$			$\Rightarrow$	

**Definition 3.5.** A *targeted* test case  $(x, y_{target}) \in \mathbb{X} \times \mathbb{Y} \setminus \{O(x)\}$  for DNN  $f$  is one that is said to fail if  $f_{pred}(x) = y_{target}$ .

**Definition 3.6.** A *confident, targeted* test case  $(x, y_{target})$  for DNN  $f$  is said to fail with confidence margin  $c > 0$  if:

$$f_{y_{target}}(x) - \max_{y \neq y_{target}} f_y(x) > c.$$

To generate such test cases, we use a modified loss function:  $L(p) = \max_{y \neq y_{target}} f_y(x) - f_{y_{target}}(x) + c$ ; a suitable  $p$  is found if  $L(p) < 0$ .

## 4 EVALUATION

We aim to answer the following three research questions:

**RQ1:** Can we detect faults deliberately introduced into a DNN?

More specifically, can we identify when the classifier has been biased by some irrelevant feature?

**RQ2:** Can we detect faults in existing state-of-the-art models?

**RQ3:** Do we find faults that other existing methods cannot find?

We use the ImageNet dataset, with 1,000 class labels and a resolution of  $512 \times 512$  pixels. The generative network that we use is a trained BigGAN [4], with weights and code provided by Brock and Andonian [3]. This is the state of the art in image generative networks, and ImageNet is a hugely popular benchmark for image classification – so the success of our method in this context shows that there is no question that the approach scales. All experiments were implemented using PyTorch 1.2.0, and executed on machines with two Intel Xeon Silver 4114 CPUs (2.20 GHz), 188 GB RAM, and an NVIDIA Tesla V100 GPU.

### 4.1 RQ1: Can our method find injected faults?

In this section, we investigate whether our method is able to detect faults intentionally injected into image classification DNNs. These faults can all be described as “instead of correctly distinguishing between image classes  $y_0$  and  $y_1$ , the DNN incorrectly uses irrelevant feature  $F$ .” For instance, “instead of correctly distinguishing between image classes ‘castle’ and ‘palace’, the DNN incorrectly uses whether the sky is cloudy or clear.” These are a type of fault that naturally arises from biased datasets, in which certain unexpected correlations appear. It is very common, if not inevitable, for there to be unintended features which are predictive in collected data [9]. By injecting faults that affect only two classes using one human-interpretable feature, it is easier to verify whether we can detect them. In practice, faults may not be as intuitive, and we explore this in Section 4.2.

*Injecting faults into DNNs.* To inject a fault into a trained image classifier, we constructed various datasets that consisted of manually-chosen subsets of ImageNet data, sometimes with intentionally incorrect labels. These were designed to encourage the network to acquire the fault. For example, to encourage the network to distinguish castles from palaces on the basis of the sky, we constructed a dataset of castles and palaces labelled ‘palace’ if the sky was clear and ‘castle’ if the sky was cloudy. Refer to Table 2 for a description of the specific faults used. After training a DNN on such a dataset, we verified that the DNN had acquired the fault as intended by using two small hold-out datasets of data: one on which high performance was expected (i.e. similar to the training dataset)

and one on which low performance was expected (i.e. biased the opposite way to the training set).

*Generating tests.* We use the method described in Section 3 to generate 200 tests for each DNN that has had a fault injected, investigating the features used to differentiate class  $y_0$  from  $y_1$ . That is, half of the tests begin with  $g(y_0, z, p = 0)$ , which is a randomly-sampled instance of  $y_0$  because  $z$  is randomly sampled from the appropriate distribution. We then optimise over  $p$  so as to create a test case  $g(y_0, z, p)$  that is classified as  $y_1$  by the DNN under test. For the other half of the generated tests,  $y_0$  and  $y_1$  are swapped in this process.

*Detecting faults.* By comparing the initial test seed  $g(y_0, z, p = 0)$  with the optimised test  $g(y_0, z, p^*)$  that is predicted to be class  $y_1$ , we can see the kinds of features that the DNN is using to distinguish  $y_0$  from  $y_1$ . If it is clear from inspecting the generated test cases that the classifier is inappropriately relying on the feature as injected, then we say that the fault has been detected. To measure whether this fault is clear from inspecting the generated test cases, we record the proportion of generated tests for which the faulty feature has changed in the direction that would indicate reliance on the fault. For instance, we might record the proportion of test cases that were erroneously classified as ‘palaces’ for which the sky became noticeably cloudier. Table 1 gives some examples of generated tests for different flawed DNNs, and the Supplementary Material includes many such examples. Table 2 shows the proportions of generated tests for each flawed DNN that noticeably indicate the presence of the flaw. We encourage the reader to peruse the Supplementary Material, which identifies the examples we took to noticeably indicate the presence of a flaw.

*Discussion.* Our method is often able to identify the faults injected, with varying degrees of ease. Where a lower proportion of tests were able to detect the fault, we conjecture one of two explanations. First, that the DNN, while learning a bias correlated with the feature we intended it to rely on, does not in fact rely on the feature described in Table 2, but rather relies on “shortcut features” [9], as discussed later. Second, that the generator network is not good at generating the change in feature we are inspecting. For instance, although fault #8 is readily detected in one direction, because many test cases remove leaves around oranges, there are few results in the other direction. This is most likely because the GAN is not able to generate lemons surrounded by leaves.

### 4.2 RQ2: Can our method detect new faults in state-of-the-art DNNs?

We would like to show that in addition to faults that have been deliberately introduced, our method is able to detect faults in ‘in the wild’ in state-of-the-art DNNs. We therefore take a classification model from the family of current highest-accuracy models for ImageNet, EfficientNet-B4 with NoisyStudent training [30], and use our new method to generate tests aiming to identify shortcomings in its behaviour. These tests are generated exactly as before: by picking an initial seed that is correctly classified as  $y_0$ , and optimising until a test case is found that is incorrectly classified as the target class  $y_1$ . To enable direct visual comparison, we use the pairs of classes  $(y_0, y_1)$  that were used in the previous section.



**Table 2: Summary of the faults injected via DNN training. The final columns indicate the proportion of tests that visually detect the fault when starting with a seed of class  $y_0$  and  $y_1$  respectively. Refer to Table 1 for examples of generated tests detecting each fault.**

#	Image label $y_0$	Image label $y_1$	% of tests that detect the fault $y_0 \Rightarrow y_1$	% of tests that detect the fault $y_1 \Rightarrow y_0$
1	Wolf (269) if setting is snow	Husky (248) if setting is grass	68	36
2	Palace (698) if clear sky	Castle (483) if cloudy sky	67	73
3	Screen (782) if screen switched on	Monitor (664) if screen switched off	48	32
4	Screwdriver (784) if ‘descending’ slope	Ballpoint pen (418) if ‘ascending’ slope	21	9
5	Coffee mug (504) if handle on right	Cup (968) if handle on left	15	16
6	Alp (970) if high colour saturation	Volcano (980) if low colour saturation	94	88
7	German shepherd (235) if tongue not out	Golden retriever (207) if tongue is out	25	34
8	Orange (950) if leaves are present	Lemon (951) if leaves not present	69	11

There is an important difference between the faults we injected and the faults that are present in state-of-the-art models. Let us define an *intelligible feature* as a feature that makes sense to a human because it aligns with the concepts that we use to understand the world. For instance, the cloudiness of the sky and whether a screen is on or off are intelligible features. By contrast, DNNs look at the raw pixel values of an image, and do not necessarily use such intuitive features. Features like the value of the green channel of the 63,099th pixel, or the maximum value of a convolution operation over a region in the image are computable features that a DNN could in principle rely on, but are not intelligible features. Willers et al. [44] have identified this discrepancy between human intuition and DNNs’ behaviour as one of the primary obstacles when testing DNNs.

DNNs have no incentive to use intelligible features. They are image recognition systems, not systems that need to actually understand the objects they are classifying. A DNN need not learn a “leg” feature to discriminate lions from sunflowers if other features are more directly useful for this end. For example, perhaps the presence of a fur texture is a better discriminator, since it will always be present if a lion is, whereas legs can be occluded or out of shot. In that case, there is no need to learn the high-level concept of “leg”. More generally, there are likely to be unintelligible features that serve the purpose of discrimination better than any intelligible features. Indeed, there is strong evidence that DNNs learn to use “shortcut features” that do not correspond to the features a human would use to solve the problem in different situations, but do allow the narrow problem at hand to be solved [9]. This can manifest as a tendency to consider low-level features such as texture at the expense of high-level features such as object shape [10]; the phenomenon of pixel-perturbation ‘adversarial examples’ is in itself evidence that DNNs are over-reliant on features that are imperceptible to humans [21].

In general, it would be surprising if the best shortcut features were the same intelligible features that humans used to understand the world. Therefore, we should expect DNNs to use mainly unintelligible features, and testing methods must take this into consideration. Testing, as many methods do (c.f. Section 5), for only unintelligible features is good, but not enough.

In our experiment with EfficientNet, we are able to consistently generate many failing test cases, answering **RQ2** in the affirmative and highlighting the DNN’s reliance on unintelligible features. Whereas in Section 4.1, we introduce human-interpretable biases, in order to make it easy to identify whether we have detected fault, the ones detected here are not of the simple form “instead of correctly distinguishing between image classes  $y_0$  and  $y_1$ , the DNN incorrectly uses irrelevant feature  $F$ .” Some examples are shown in Figure 1, and many more are given in the supplementary material. We are able to find these problems by leveraging the powerful representation learnt by GANs. GANs are able to identify such a large range of faults because they learn to generate images directly from the data, and model a large amount of the variation in this data, intuitive or otherwise.

### 4.3 RQ3: Can existing methods detect the faults found by our method?

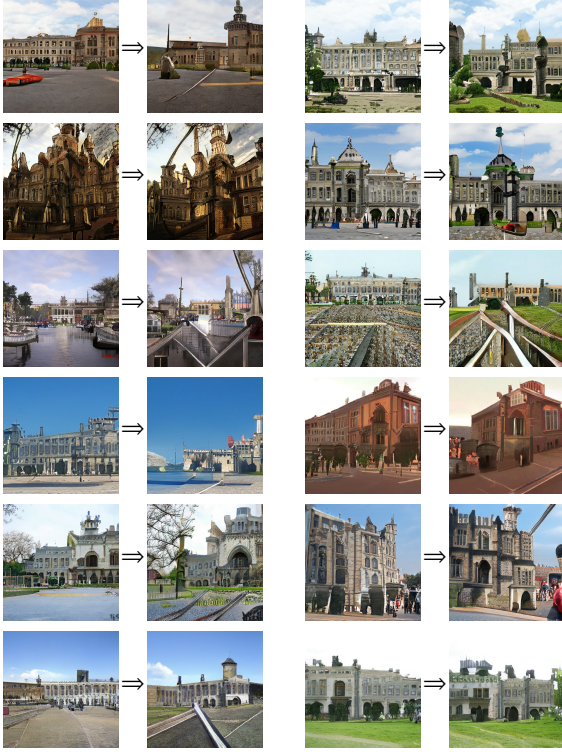
To establish a negative answer, we:

- Show that pixel perturbation approaches are not able to generate almost any of the test cases that our method generates.
- Demonstrate that the faults found using pixel-perturbation approaches are disjoint from the faults found using our approach, when applied to state-of-the-art models.
- Provide a comprehensive literature review in Section 5, including descriptions of why each method is unable to detect the faults found by our method.

We focus primarily on pixel-perturbation methods here because most of the established literature on testing DNNs uses exclusively this approach. Our related work section surveys all relevant techniques, including those that do not take this approach.

**4.3.1 Magnitude of Changes in Pixel Space.** As described in Section 3.2.1, pixel-space perturbations are constrained so as to ensure that the perturbed images remain the same class as the unperturbed image. Concretely, on ImageNet, pixel-space perturbations are typically constrained to have an  $\ell_2$  magnitude of at most 3 [6]. A model adversarially trained against perturbations constrained this way was can be described as “highly robust” [38, p. 6]. Indeed, there exist pixel perturbations with an  $\ell_2$  magnitude of 22 that can completely change the true class label of an image [42, Fig. 3]; this would be a

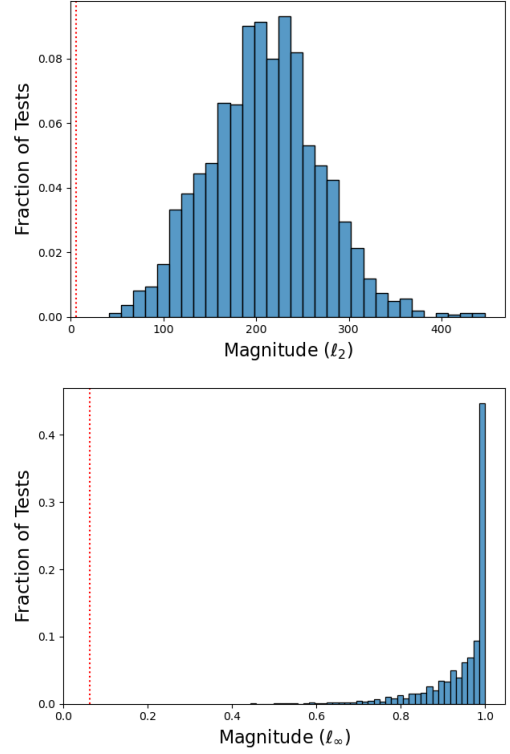
**Figure 1: Examples of tests cases generated for EfficientNet-B4, with test seeds of class ‘palace’ and each test case incorrectly classified as ‘castle’.**



very large magnitude for a pixel-space perturbation. For the  $\ell_\infty$  metric, a maximum pixel-space perturbation magnitude of  $\epsilon = 16/255$  is typical [6].

Our method performs perturbations to learnt feature representations, which then affect the downstream pixel values. Therefore, a small change to the output of early layers in the generator can result in a large change to the pixel values as measured by an  $\ell_2$  norm. But because these changes are context-sensitive to learned features, they preserve the meaning of the image. For example, suppose that a perturbation results in a dog moving position on a grassy background: although there is no change to the meaning of the image, the distance as measured by an  $\ell_2$  norm will be great, since many pixels will change value. In short, by leveraging generative models to direct changes to meaningful features, we can induce large changes in pixel space.

We investigate the empirical distribution of pixel-space distances between initial test seeds and final perturbed test cases across 1000 initial seeds. Figure 2 shows that 100% of semantically-perturbed test inputs are much further than the maximum pixel-perturbation constraint under either popular distance metric. Since the  $\ell_\infty$  distance is the greatest amount any one pixel changes, there is a cluster around 1.0 because there is often at least one pixel that completely changes its value. By contrast, an  $\ell_\infty$  pixel-space constraint of  $\epsilon = 1.0$  is equivalent to no constraint: all pixels can change value arbitrarily.



**Figure 2: Magnitudes of perturbations produced by our method, as measured in pixel space using  $\ell_2$  and  $\ell_\infty$  metrics. In red, a typical upper bound  $\epsilon$  for pixel perturbations. Our perturbations are generally much larger than those allowed by pixel perturbations.**

**Table 3: Table showing the accuracies of pixel-perturbation robust classifiers on test cases generated for standard classifiers, using both pixel perturbations and our test generation method. The accuracies are significantly higher on the pixel perturbation tests, suggests that our approach detects faults of a different nature.**

		Attacking EfficientNet-B4NS	
		Pixel Perts	Our Perts
Test	Robust ResNet50 [6]	56%	27%
	Robust ResNet50 [45]	53%	24%
		Attacking ResNet50	
		Pixel Perts	Our Perts
Test	Robust ResNet50 [6]	36%	25%
	Robust ResNet50 [45]	32%	22%

**4.3.2 Transferability analysis.** We have established that pixel perturbation methods cannot generate the test cases output by our method. In this section, we strengthen the case that furthermore, our method is able to find *faults* that pixel perturbation approaches

cannot. For faults concerning intelligible features, such as those introduced in Section 4.1, the case is clear: the faults involve visible changes to meaningful features, and therefore changes with an  $l_p$  distance greater than is allowed by pixel perturbations. Stated plainly, methods that generate imperceptibly different test cases are unable to detect faults concerning exclusively visibly-different features.

However, in Section 4.2 we established that when testing state-of-the-art models, the faults are not of this kind. Therefore, it could be possible that even though the particular test inputs generated by our method and existing methods were disjoint, they were both indicative of the same underlying faults in the DNNs, in the sense that they would both be solved with the same fix. To demonstrate that this is not the case, we use *adversarially-trained* DNNs [37]. Adversarial training is a technique that performs worst-case pixel perturbations during the training of a DNN. When training converges, the result is that the DNN is more robust to these kinds of faults, and has learned to ignore the spurious features pixel perturbations affect. While this does not completely ‘fix’ sensitivity to pixel perturbations, it greatly improves it [6]. We check whether this ‘fix’ also applies to our perturbations.

We analyse whether the test cases generated *transfer* to models that have been adversarially trained to be robust to pixel-space perturbations. By “transfer”, we mean that we measure whether test cases generated so as to induce a fault in (say) EfficientNet-B4 also induce faults in an adversarially-trained DNN. Table 3 shows the proportion of test cases for EfficientNet-B4 and a standard ResNet50 that are classified correctly by two DNNs trained to be robust against pixel-space perturbations: one by Wong et al. [45], which is robust to 31% of  $l_\infty$  perturbations with  $\epsilon = 4/255$ , and one by Engstrom et al. [6], which is robust to 35% of  $l_2$  perturbations with  $\epsilon = 3$ .

We can see that pixel-perturbation tests generated for EfficientNet-B4 tend not to transfer to the pixel-robust models, likely because the faults targeted by the tests in EfficientNet-B4 are not present due to the adversarial training. Conversely, we can see that the tests generated by our method for EfficientNet-B4 *do* tend to transfer to the pixel-robust models. The results for ResNet50 are similar, although perturbations transfer slightly better, likely because the architecture is the same as the robust models. Note that the test cases for both methods remain confident, targeted tests: the difference in accuracy is not due to the pixel perturbations being only initially just misclassified.

Because the test cases generated by our method continue to detect faults in adversarially-trained classifiers, we have confidence that these must be detecting different kinds of faults to those detected by the pixel-perturbation method. If the failing test cases were indicative of the same underlying faults, then we would see that the accuracies of the transferred test cases would be similar.

#### 4.4 Threats to Validity

Our *primary* threat to validity arises from the inherent complexity of testing DNNs. It is difficult to exactly describe what a fault is, or to attribute a fault to any one cause. In this paper, we have chosen to deliberately introduce consistent biases into the DNNs’ behaviours, and show that our method is in turn able to consistently produce

examples that highlight this bias. Doing things in this way helps clarify what the fault is, and whether we have identified it. There exist numerous papers in which the model was shown to be wrong on many inputs, but we go beyond this by showing our ability to pick up on not just local, but global biases in the network. However, as discussed in Section 4.2, this may not be as simple for classifiers in the wild, and making progress in this direction is important for the future of DNN testing.

A *second* possible threat is the validity of the labelling done to produce Table 2. We hand-label whether perturbations have revealed the bias injected into the classifier, and while we take care to label perturbations only when they are clearly attributable to the fault, there is some subjectivity involved. To address this, we provide some examples in Table 1, and many more in the supplementary materials, so readers may judge for themselves.

A *third* possible threat relates to problems with GANs. GANs are known to drop modes [29], meaning they may not generate certain parts of the input distribution. However, they need only represent enough of the distribution to identify at least some faults; our results show that they do. GANs are also not perfect generators, and so images may look unrealistic. In fact, realism is not required for our purposes. As long as a class is recognisable, we can still show that the classifier is paying attention to the wrong features. If a classifier can identify an unrealistic palace, but adding clouds in the sky changes its prediction to a castle, this betrays a problem in the classifier’s internal ‘logic’. In addition, if our aim is to create *human-aligned* classifiers, performance on unrealistic but recognisable images is important. Finally, our method does not explicitly require a GAN, and could easily use a VAE or other generative model that does not drop modes. We expect recent rapid advances in generative machine learning to continue, making approaches that leverage it increasingly promising.

## 5 RELATED WORK

*Pixel Perturbations and Adversarial Robustness.* There has been a large amount of recent work on ‘adversarial examples’, examples deliberately made to fool a classifier [11]. This setup represents the worst-case scenario, in which an ‘attacker’ actively works to find examples on which the classifier, the ‘defender’, will fail. The most popular method for doing so, dubbed ‘pixel perturbations’ involves fooling the classifier by individually changing the pixels of an input image [14]; both attacking with and defending against these perturbations has been extensively explored [16, 26, 37]. Typically pixel perturbations allow for arbitrary changes, independent of the content of the image, as long as they are almost invisible. In practice, this is done by limiting the magnitude of the perturbation, and bypasses the oracle problem by assuming that the true class of the perturbed image is unchanged from the original image. While this constraint makes the perturbations simple to implement, it is also very limiting, allowing for testing on only a tiny fraction of interesting cases. In response to this, new methods have been proposed to make large, visual changes to images, addressing the oracle problem in a variety of ways.

*Creating New Datasets From Scratch.* Others have created entirely new datasets from scratch. On ImageNet, Recht et al. [36] repeat the



original process used to create ImageNet, and find that state-of-the-art classifiers fail to generalize. Still on ImageNet, Hendrycks et al. [19] filter gathered data to create a deliberately more challenging dataset.

*Hand-Crafted Perturbations.* Many proposed methods perturb images using a fixed number of hand-crafted perturbations, designed to preserve the image’s true class. Each perturbation type makes a single, narrow change, and these include translations, rotations, zoom, shearing, brightness, contrast, blurring, colours, and fog/rain effects [7, 8, 17, 20, 32, 41, 46]. Zhao et al. [49] perturb colours, but exploit human biases in perceptual colour distance to make large yet imperceptible changes. All of these perturbations are, like pixel perturbations, agnostic to the contents of the image, and can be applied uniformly to any image. They have been used to create entirely new datasets, designed to serve as robustness benchmarks for both ImageNet [18], MNIST [33], and others. These methods are clearly not capable of making the wide range of adaptive changes our method makes. They would not, for example, be able to add clouds to a sky, or change the colour of the background.

*Beyond Hand-Crafting.* While hand-crafted perturbations allow for a greater variety in the perturbations made to images, they disregard the *content* of images. This is in large part because when working directly with the pixels of an image, it is difficult to devise a single transformation that can be applied consistently across several images. If we wanted to change the colour of dogs’ fur, we would have to apply a change uniquely to all dog images. To cope with this, automatic methods have been designed to make these changes, leveraging alternative representations of images. One interesting but expensive possibility is writing a differentiable renderer for the desired domain, and making changes to the generated scene by modifying the parameters of the renderer [23, 27]. However, differentiable renderers are uncommon and inflexible relative to generative networks such as GANs, and may be prohibitively difficult to train on a new dataset.

*Leveraging Generative Models.* A natural approach is to leverage the representations learnt by generative models such as GANs [12], as we do in this paper.

Some methods attempt to retain fine-grained control over the changes made by sacrificing flexibility. These methods typically select the types of features they will modify, and then create a model especially for modifying these features. *DeepRoad* [47] use *UNIT* [28], an image-to-image translation technique, to produce images of the same road in sunny, rainy and snowy conditions. Bhattad et al. [2] leverage pre-trained colourisation and texture-transfer models to adversarially change the colours and textures of an image. A number of publications Goyal et al. [15], Joshi et al. [24], Qiu et al. [35] in some way exploit generative models with disentangled latent spaces, be it by using *Fader Networks* [24], using a dataset with labelled attributes to train a conditional generator [35], or using a *StyleGAN* and partitioning the latent space according to whether or not it should influence the label [15]. Selecting the features to perturb like this allows for precise control over these features, but like hand-crafted perturbations, result in narrow kinds of changes to images.

Other methods do not provide this control, but are able to harness the full flexibility and variation of the representation learnt by the GAN. An alternative is to look for adversarial inputs by searching directly in the input space of a GAN [40, 43, 48]. However, Dunn et al. [5] have shown that it is possible to leverage even more of the GAN’s implicit representation of the features by perturbing not only in the input space, but also within the activations of the GAN itself. We also exploit this and focus on earlier layers to make more granular changes to images. None of these papers explicitly look at the use of GAN-based perturbations for introducing and detecting faults in a DNN.

## 6 CONCLUSION

In this paper, we have demonstrated that our method for the generation of tests for DNNs is able to detect faults that existing approaches cannot. This is possible because our method leverages generative machine learning, allowing it to manipulate higher-level features of generated test cases (e.g. position, colour, texture of objects) rather than just low-level features (i.e. individual pixel values). As a result, the generated tests are much more varied and can explore weaknesses not reachable when changes are constrained to be within a small  $\ell_p$  distance in pixel space.

Of course, we do not expect that our method will be able to detect all faults in a given DNN. But exploiting features learned from data during test case generation seems a promising approach worthy of future investigation. More generally, we encourage future work that seeks to meaningfully broaden the set of faults detectable by our tests. In addition to this bottom-up approach, top-down attempts to identify a superset of the requirements for a DNN might also be worth investigating.

In general, most of the methods for producing test-cases for DNNs either only implicitly, or do not at all, address some of the main issues in testing DNNs. To test DNNs in practice, it will become increasingly important to provide specifications, consider a DNN’s behaviour as part of a larger system, and pin down how to identify and correct faults [39]. Unlike conventional software, for which debugging tools allow direct inspection of the program fault, a DNN cannot be meaningfully inspected by a developer. Even if it could, there is little hope trying to manually adjust the weights of a trained DNN. Instead, developers act on DNNs indirectly, through training code and data. Since all faults are mediated through this opaque training process, it is difficult to link a DNN failure to an action that might introduce a fix. We encourage future work that aims to make such diagnostic debugging possible, either by directly debugging training code and datasets, or by analysing the link between the trained DNN and these training artefacts.

## REFERENCES

- [1] David Bau, Jun-Yan Zhu, Hendrik Strobelt, Bolei Zhou, Joshua B. Tenenbaum, William T. Freeman, and Antonio Torralba. 2019. GAN Dissection: Visualizing and Understanding Generative Adversarial Networks. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. [https://openreview.net/forum?id=Hyg\\_X2C5FX](https://openreview.net/forum?id=Hyg_X2C5FX)
- [2] Anand Bhattad, Min Jin Chong, Kaizhao Liang, Bo Li, and David A. Forsyth. 2019. Big but Imperceptible Adversarial Perturbations via Semantic Manipulation. *arXiv:1904.06347 [cs]* (April 2019). [arXiv:1904.06347 \[cs\]](https://arxiv.org/abs/1904.06347)
- [3] Andrew Brock and Alex Andonian. 2019. BigGAN-PyTorch. (2019). <https://github.com/ajbrock/BigGAN-PyTorch>

- [4] Andrew Brock, Jeff Donahue, and Karen Simonyan. 2019. Large Scale GAN Training for High Fidelity Natural Image Synthesis. In *International Conference on Learning Representations (ICLR)*. <https://openreview.net/forum?id=B1xsqj09Fm>
- [5] Isaac Dunn, Laura Hanu, Hadrien Pouget, Daniel Kroening, and Tom Melham. 2020. Evaluating Robustness to Context-Sensitive Feature Perturbations of Different Granularities. arXiv:2001.11055 [cs.CV]
- [6] Logan Engstrom, Andrew Ilyas, Shibani Santurkar, and Dimitris Tsipras. 2019. Robustness (Python Library). <https://github.com/MadryLab/robustness>
- [7] Logan Engstrom, Brandon Tran, Dimitris Tsipras, Ludwig Schmidt, and Aleksander Madry. 2019. Exploring the Landscape of Spatial Robustness. In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA (Proceedings of Machine Learning Research, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, 1802–1811. <http://proceedings.mlr.press/v97/engstrom19a.html>
- [8] Xiang Gao, Ripon K. Saha, Mukul R. Prasad, and Abhik Roychoudhury. 2020. Fuzz testing based data augmentation to improve robustness of deep neural networks. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1147–1158. <https://doi.org/10.1145/3377811.3380415>
- [9] Robert Geirhos, Jörn-Henrik Jacobsen, Claudio Michaelis, Richard S. Zemel, Wieland Brendel, Matthias Bethge, and Felix A. Wichmann. 2020. Shortcut Learning in Deep Neural Networks. CoRR abs/2004.07780 (2020). arXiv:2004.07780 <https://arxiv.org/abs/2004.07780>
- [10] Robert Geirhos, Patricia Rubisch, Claudio Michaelis, Matthias Bethge, Felix A. Wichmann, and Wieland Brendel. 2019. ImageNet-trained CNNs are biased towards texture; increasing shape bias improves accuracy and robustness. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. <https://openreview.net/forum?id=Bygh9j09KK>
- [11] Justin Gilmer, Ryan P. Adams, Ian J. Goodfellow, David Andersen, and George E. Dahl. 2018. Motivating the Rules of the Game for Adversarial Example Research. CoRR abs/1807.0 (2018). arXiv:1807.06732 <http://arxiv.org/abs/1807.06732>
- [12] Ian J. Goodfellow. 2017. NIPS 2016 Tutorial: Generative Adversarial Networks. CoRR abs/1701.0 (2017). arXiv:1701.00160 <http://arxiv.org/abs/1701.00160>
- [13] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron C. Courville, and Yoshua Bengio. 2014. Generative Adversarial Nets. In *Advances in Neural Information Processing Systems (NeurIPS)*, Zoubin Ghahramani, Max Welling, Corinna Cortes, Neil D. Lawrence, and Kilian Q. Weinberger (Eds.). 2672–2680. <http://papers.nips.cc/paper/5423-generative-adversarial-nets>
- [14] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. 2015. Explaining and Harnessing Adversarial Examples. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). <http://arxiv.org/abs/1412.6572>
- [15] Sven Gowal, Chongli Qin, Po-Sen Huang, Taylan Cemgil, Krishnamurthy Dvijotham, Timothy A. Mann, and Pushmeet Kohli. 2019. Achieving Robustness in the Wild via Adversarial Mixing with Disentangled Representations. CoRR abs/1912.03192 (2019). arXiv:1912.03192 <http://arxiv.org/abs/1912.03192>
- [16] Jianmin Guo, Yu Jiang, Yue Zhao, Quan Chen, and Jianguang Sun. 2018. DLFuzz: differential fuzzing testing of deep learning systems. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 739–743. <https://doi.org/10.1145/3236024.3264835>
- [17] Dan Hendrycks and Thomas Dietterich. 2019. Benchmarking Neural Network Robustness to Common Corruptions and Perturbations. arXiv:1903.12261 [cs, stat] (March 2019). arXiv:1903.12261 [cs, stat] <http://arxiv.org/abs/1903.12261>
- [18] Dan Hendrycks and Thomas G. Dietterich. 2019. Benchmarking Neural Network Robustness to Common Corruptions and Perturbations. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. <https://openreview.net/forum?id=HJz6tiCqYm>
- [19] Dan Hendrycks, Kevin Zhao, Steven Basart, Jacob Steinhardt, and Dawn Song. 2019. Natural Adversarial Examples. CoRR abs/1907.07174 (2019). arXiv:1907.07174 <http://arxiv.org/abs/1907.07174>
- [20] Hossein Hosseini and Radha Poovendran. 2018. Semantic Adversarial Examples. In *2018 IEEE Conference on Computer Vision and Pattern Recognition Workshops, CVPR Workshops 2018, Salt Lake City, UT, USA, June 18-22, 2018*. IEEE Computer Society, 1614–1619. <https://doi.org/10.1109/CVPRW.2018.00212>
- [21] Andrew Ilyas, Shibani Santurkar, Dimitris Tsipras, Logan Engstrom, Brandon Tran, and Aleksander Madry. 2019. Adversarial Examples Are Not Bugs, They Are Features. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 125–136. <http://papers.nips.cc/paper/8307-adversarial-examples-are-not-bugs-they-are-features>
- [22] Ali Jahanian, Lucy Chai, and Phillip Isola. 2019. On the "steerability" of generative adversarial networks. CoRR abs/1907.07171 (2019). arXiv:1907.07171 <http://arxiv.org/abs/1907.07171>
- [23] Lakshya Jain, Wilson Wu, Steven Chen, Uyeong Jang, Varun Chandrasekaran, Sanjit A. Seshia, and Somesh Jha. 2019. Generating Semantic Adversarial Examples with Differentiable Rendering. CoRR abs/1910.00727 (2019). arXiv:1910.00727 <http://arxiv.org/abs/1910.00727>
- [24] Ameya Joshi, Amitangshu Mukherjee, Soumik Sarkar, and Chinmay Hegde. 2019. Semantic Adversarial Attacks: Parametric Transformations That Fool Deep Classifiers. In *2019 IEEE/CVF International Conference on Computer Vision, ICCV 2019, Seoul, Korea (South), October 27 - November 2, 2019*. IEEE, 4772–4782. <https://doi.org/10.1109/ICCV.2019.00487>
- [25] Diederik P. Kingma and Max Welling. 2014. Auto-Encoding Variational Bayes. In *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). <http://arxiv.org/abs/1312.6114>
- [26] Seokhyun Lee, Sooyoung Cha, Dain Lee, and Hakjoo Oh. 2020. Effective white-box testing of deep neural networks with adaptive neuron-selection strategy. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 165–176. <https://doi.org/10.1145/3395363.3397346>
- [27] Hsueh-Ti Derek Liu, Michael Tao, Chun-Liang Li, Derek Nowrouzezahrai, and Alec Jacobson. 2019. Beyond Pixel Norm-Balls: Parametric Adversaries using an Analytically Differentiable Renderer. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. <https://openreview.net/forum?id=SJl2niR9KQ>
- [28] Ming-Yu Liu, Thomas Breuel, and Jan Kautz. 2017. Unsupervised Image-to-Image Translation Networks. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). 700–708. <http://papers.nips.cc/paper/6672-unsupervised-image-to-image-translation-networks>
- [29] Tengyu Ma. 2018. Generalization and equilibrium in generative adversarial nets (GANs) (invited talk). In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2018, Los Angeles, CA, USA, June 25-29, 2018*, Ilias Diakonikolas, David Kempe, and Monika Henzinger (Eds.). ACM, 2. <https://doi.org/10.1145/3188745.3232194>
- [30] Luke Melas-Kyriazi. 2020. lukemelas/EfficientNet-PyTorch. <https://github.com/lukemelas/EfficientNet-PyTorch> original-date: 2019-05-30T05:24:11Z.
- [31] Mehdi Mirza and Simon Osindero. 2014. Conditional Generative Adversarial Nets. CoRR abs/1411.1 (2014). arXiv:1411.1784 <http://arxiv.org/abs/1411.1784>
- [32] Jeet Mohapatra, Tsui-Wei Weng, Pin-Yu Chen, Sijia Liu, and Luca Daniel. 2020. Towards Verifying Robustness of Neural Networks Against A Family of Semantic Perturbations. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2020, Seattle, WA, USA, June 13-19, 2020*. IEEE, 241–249. <https://doi.org/10.1109/CVPR42600.2020.00032>
- [33] Norman Mu and Justin Gilmer. 2019. MNIST-C: A Robustness Benchmark for Computer Vision. CoRR abs/1906.02337 (2019). arXiv:1906.02337 <http://arxiv.org/abs/1906.02337>
- [34] Kexin Pei, Yinzi Cao, Junfeng Yang, and Suman Jana. 2017. DeepXplore: Automated Whitebox Testing of Deep Learning Systems. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 1–18. <https://doi.org/10.1145/3132747.3132785>
- [35] Haonan Qiu, Chaowei Xiao, Lei Yang, Xinchun Yan, Honglak Lee, and Bo Li. 2019. SemanticAdv: Generating Adversarial Examples via Attribute-Conditional Image Editing. arXiv:1906.07927 [cs, eess] (June 2019). arXiv:1906.07927 [cs, eess] <http://arxiv.org/abs/1906.07927>
- [36] Benjamin Recht, Rebecca Roelofs, Ludwig Schmidt, and Vaishaal Shankar. 2019. Do ImageNet Classifiers Generalize to ImageNet?. In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA (Proceedings of Machine Learning Research, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, 5389–5400. <http://proceedings.mlr.press/v97/recht19a.html>
- [37] Kui Ren, Tianhang Zheng, Zhan Qin, and Xue Liu. 2020. Adversarial attacks and defenses in deep learning. *Engineering* 6, 3 (2020), 346–360.
- [38] Hadi Salman, Andrew Ilyas, Logan Engstrom, Ashish Kapoor, and Aleksander Madry. 2020. Do Adversarially Robust ImageNet Models Transfer Better?. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (Eds.). <https://proceedings.neurips.cc/paper/2020/hash/24357dd085d2c4b1a88a7e0692e60294-Abstract.html>
- [39] Sanjit A. Seshia, Somesh Jha, and Tommaso Dreossi. 2020. Semantic Adversarial Deep Learning. *IEEE Des. Test* 37, 2 (2020), 8–18. <https://doi.org/10.1109/MDAT.2020.2968274>
- [40] Yang Song, Rui Shu, Nate Kushman, and Stefano Ermon. 2018. Constructing Unrestricted Adversarial Examples with Generative Models. In *Advances in Neural Information Processing Systems (NeurIPS)*, Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett

- (Eds.). 8322–8333. <http://papers.nips.cc/paper/8052-constructing-unrestricted-adversarial-examples-with-generative-models>
- [41] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. DeepTest: automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 303–314. <https://doi.org/10.1145/3180155.3180220>
- [42] Florian Tramèr, Jens Behrmann, Nicholas Carlini, Nicolas Papernot, and Jörn-Henrik Jacobsen. 2020. Fundamental Tradeoffs between Invariance and Sensitivity to Adversarial Perturbations. *CoRR* abs/2002.04599 (2020). [arXiv:2002.04599](https://arxiv.org/abs/2002.04599) <https://arxiv.org/abs/2002.04599>
- [43] Shuo Wang, Shangyu Chen, Tianle Chen, Surya Nepal, Carsten Rudolph, and Marthie Grobler. 2020. Generating Semantic Adversarial Examples via Feature Manipulation. *arXiv:2001.02297 [cs, stat]* (Jan. 2020). [arXiv:2001.02297 \[cs, stat\]](https://arxiv.org/abs/2001.02297) <http://arxiv.org/abs/2001.02297>
- [44] Oliver Willers, Sebastian Sudholt, Shervin Raafatnia, and Stephanie Abrecht. 2020. Safety Concerns and Mitigation Approaches Regarding the Use of Deep Learning in Safety-Critical Perception Tasks. In *Computer Safety, Reliability, and Security. SAFECOMP 2020 Workshops - DECSoS 2020, DepDevOps 2020, USDAI 2020, and WAISE 2020, Lisbon, Portugal, September 15, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12235)*, António Casimiro, Frank Ortmeier, Erwin Schoitsch, Friedemann Bitsch, and Pedro M. Ferreira (Eds.). Springer, 336–350. [https://doi.org/10.1007/978-3-030-55583-2\\_25](https://doi.org/10.1007/978-3-030-55583-2_25)
- [45] Eric Wong, Leslie Rice, and J. Zico Kolter. 2020. Fast is better than free: Revisiting adversarial training. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26–30, 2020*. OpenReview.net. <https://openreview.net/forum?id=BJx040EFvH>
- [46] Xiaofei Xie, Lei Ma, Felix Juefei-Xu, Minhui Xue, Hongxu Chen, Yang Liu, Jianjun Zhao, Bo Li, Jianxiong Yin, and Simon See. 2019. DeepHunter: a coverage-guided fuzz testing framework for deep neural networks. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 146–157. <https://doi.org/10.1145/3293882.3330579>
- [47] Mengshi Zhang, Yuqun Zhang, Lingming Zhang, Cong Liu, and Sarfraz Khurshid. 2018. DeepRoad: GAN-based metamorphic testing and input validation framework for autonomous driving systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018)*. Association for Computing Machinery, New York, NY, USA, 132–142. <https://doi.org/10.1145/3238147.3238187>
- [48] Zhengli Zhao, Dheeru Dua, and Sameer Singh. 2017. Generating Natural Adversarial Examples. *arXiv:1710.11342 [cs]* (Oct. 2017). [arXiv:1710.11342 \[cs\]](https://arxiv.org/abs/1710.11342) <http://arxiv.org/abs/1710.11342>
- [49] Zhengyu Zhao, Zhuoran Liu, and Martha A. Larson. 2020. Towards Large Yet Imperceptible Adversarial Image Perturbations With Perceptual Color Distance. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2020, Seattle, WA, USA, June 13–19, 2020*. IEEE, 1036–1045. <https://doi.org/10.1109/CVPR42600.2020.00112>