Barrage of Random Transforms for Adversarially Robust Defense

Edward Raff[1,2,4] Jared Sylvester[1,2,4] Steven Forsyth[3] Mark McLean[1]

[1]Laboratory for Physical Sciences [2]Booz Allen Hamilton [3]NVIDIA [4]U.M.B.C.

Abstract

Defenses against adversarial examples, when using the ImageNet dataset, are historically easy to defeat. The common understanding is that a combination of simple image transformations and other various defenses are insufficient to provide the necessary protection when the obfuscated gradient is taken into account. In this paper, we explore the idea of stochastically combining a large number of individually weak defenses into a single barrage of randomized transformations to build a strong defense against adversarial attacks. We show that, even after accounting for obfuscated gradients, the Barrage of Random Transforms (BaRT) is a resilient defense against even the most difficult attacks, such as PGD. BaRT achieves up to a 24× improvement in accuracy compared to previous work, and has even extended effectiveness out to a previously untested maximum adversarial perturbation of $\epsilon = 32$.

1. Introduction

Adversarial machine learning has been a research area for over a decade [1], but it has recently received increased focus and attention from the larger community. This is largely due to the success of modern deep learning techniques within the realm of computer vision tasks and the surprising ease with which such systems are fooled into giving incorrect decisions [2]. In particular, there are concerns about the safety of self-driving cars, as they could be fooled into misreading stop signs as speed limits, and other possible nefarious actions [3].

Consider an adversary A whom, given some victim model $f(\cdot)$, wants to alter $\widetilde{x} = A(x)$ such that $f(x) \neq f(A(x))$. Many works have attempted to find a transform $t(\cdot)$ that can be applied to an image x to yield a new image $\hat{x} = t(x)$ such that $f(x) = f(t(A(x)))$. If it were possible to find such a defensive transform $t(\cdot)$ it would allow us a simple and convenient way to circumvent the adversarial problem. This is particularly alluring for computer vision, since there exists a rich literature of computer vision transformations to pull from. Athalye, Carlini, and Wagner [4] has shown that the many attempts to find such a defensive transformation $t(\cdot)$ that defeats adversarial attacks have all failed, due to a problem they term obfuscated gradients. More broadly, every defense we are aware of that has undergone thorough evaluation has failed to produce any level of protection for ImageNet[5], as exemplified in the RobustML catalog where all ImageNet results are reduced to $\leq 0.1\%$ accuracy.[1] In contrast, we present a new, state-of-the-art defense for ImageNet that pays some cost to accuracy when not under attack, but achieves a Top-5 accuracy of up to 57.1% when under attack. These attacks are carried out by the strongest adversary we could construct, which is significantly stronger than those used in similar work in key respects.

In our work, we instead look not for a single transformation $t(\cdot)$, but propose to build a collection of many different transforms $t_{1,\ldots,n}$ from which we will randomly select a subset to apply to each image at both training and testing time. The individual transforms will be randomly parameterized as will the particular subset chosen and the order in which they are applied. By creating a barrage of random transformations, we show that such an ensemble defense can provide tangible benefits against attack, even after taking into account all of the methods by which obfuscated gradients can mislead us into using a broken defense [4].

Overall we provide the following contributions:

- A new, state-of-the-art defense on ImageNet, that fully accounts for the obfuscated gradients issue.
- Results that show ensembling weak defenses can create a strong defense, provided they are combined in a randomized fashion and the population of defenses is large. Prior work had conjectured that this was not the case [6].

BaRT is inspired by and builds upon a number of prior works that have used singular transformations to try to defend against attacks. We will review work related to our approach in section 2 and detail both the

[1] https://www.robust-ml.org/

BaRT strategy and it's constituent transformations in section 3, as well as the threat model of our adversary in section 4. While heuristic in nature, we find that after accounting for our strongest adversary we obtain state-of-the-art robustness against attack on the ImageNet dataset, which we show in section 5.

2. Related Work

While work on adversarial attacks against machine learning models has existed for over a decade, recent work that showed their success against neural networks [7, 8] has spawned increased motivation and attention to this problem. There were some who thought this concern was over stated, and that the number of variations in position, lighting, angle, and other factors that would occur in the real world would render adversarial attacks a non-issue for physical systems [9]. However, it was later shown that these difficulties could be circumvented making it possible for adversarial examples to be constructed [10, 11].

Still, the intuition that adjustments in angle, position, or other kinds of visual transformations of some object could defeat an adversary by somehow filtering or removing the adversary's perturbations was strong and alluring. As such, many papers have been presented that attempt to defeat adversaries using some kind of image pre-processing before classification (e.g., [12–14]). As far as we are aware, these types of defenses have all been defeated in the white-box threat model, either by correctly incorporating the defense into the adversary's search procedure [2], or by properly accounting for obfuscated gradients [4]. Obfuscated gradients occur when the defense has, intentionally or not, masked information about the gradient making it unreliable (or non-existent) for the adversary to use. These can occur in a number of ways, but all of which have proposed workarounds to obtain a suitable approximate gradient for the adversary to use [4]. In this work, we use only techniques which have already been defeated to build our defense. This way we can leverage known solutions to the obfuscated gradient and thus fully account for the problem and ensure our adversary's attack has full knowledge of the defense.

Few approaches have been able to scale up to ImageNet's size, and we find most works that have attempted to defend it against attack have been based on transformations or denoising. Prakash, Moran, Garber, *et al.* [15] claimed 81% accuracy under attack and Liao, Liang, Dong, *et al.* [16] 75%, but both were reduced to 0% under just $\epsilon = 4$ when obfuscated gradients were accounted for [17]. Xie, Zhang, Yuille, *et al.* [18] claimed 86% accuracy and Guo, Rana, Cissé, *et al.* [13] 75%, but these were later also reduced to 0% accuracy[4]. Even different approaches with more modest claims were later shown to be deficient, such as Kannan, Kurakin, and Goodfellow [19] who initially reported 27.9% accuracy but which was later demonstrated to be 0.1% [20].

Others before us have looked at building a multi-component defense, but prior work has reached the conclusion that a combined defense is no stronger than any of its constituent members [6]. In this paper we demonstrate that this is not necessarily true. Prior attempts at ensembling defenses have all combined their constituents in a fixed strategy, which has failed to be useful. In contrast, we demonstrate that a stochastic combination of weak defenses is effective.

A number of recent works have looked at developing provably secure training procedures for deep learning [21–23]. We believe that in the long term this is the most encouraging and desirable path toward defending against adversarial attacks. However, these methods are not yet usable for large datasets. The most recent work in this area has been "scaling up" to CIFAR-10 [24], which is orders of magnitude smaller than ImageNet.

The state-of-the-art defense that has been repeatedly found to be effective is Adversarial Training, which involves augmenting the training data with adversarially crafted examples generated as the training progresses [8]. Madry, Makelov, Schmidt, *et al.* [25] used adversarial training on the CIFAR dataset, which still has the best empirical robustness to attack [24] and has been repeatedly validated as effective and capable of fully defending against the best known adversaries under the white-box threat model [4]. Kurakin, Goodfellow, and Bengio [26] attempted to scale adversarial training up to the ImageNet dataset, which they found especially difficult. As far as we are aware, their work provides the best uncontested defense against adversarial attack on ImageNet. Against an adversary operating in the L_∞ distance, they obtain Top-1 and Top-5 accuracy of 1.5% and 5.5% for Top-1 and Top-5 respectively for a max perturbation of $\epsilon = 16$. We will show that our defense outperforms adversarial training across all $\epsilon \in [2, 16]$, and even continues to provide a robust defense up to $\epsilon = 32$. We are not aware of any prior work which has considered an L_∞ adversary given this wide of a range.

3. A Barrage of Random Transforms

Given the research that has been performed over the past year, it is clear that a single transformation of the input image is not sufficient to produce a reliable defense. We take the perspective that given an omnipotent adversary, randomness is one way to construct a decision process that the adversary can not trivially circumvent. The question then becomes: *is there a way to randomly pre-process images before they are classified*

by a CNN, such that accuracy is not obliterated and the adversary is unable to effectively operate?

Since we are working on images, we can make use of a plethora of pre-existing image transformation and preprocessing steps that have been developed by the computer vision community over the past several decades. We leverage these to create 10 groups $G_{1,\ldots,10}$ of transformations. Each group G_j will have some number of transforms $t(\cdot)$ contained within that group. We used a total of $n = 25$ different transforms $t_{1,\ldots,25}$, and denote the set of all transforms $T = \bigcup_{i=1}^{25} t_i$, and $\forall j, G_j \subset T$, with each group of transforms having no overlap ($G_j \cap G_k = \emptyset$ for $j \neq k$).

Each transform $t_i(\cdot)$ will have some parameters p_i that alter the behavior of the transform, and so by randomly selecting the values of p we can can have $t_i(x|p_i)$ produce many different outputs, introducing a stochastic component. This alone is not new, but we also have a collection of n different transforms to choose from. To further maximize the randomness, we select an ordering, or "permutation," π of k transforms to apply. The ordering π will change every time we attempt to use a model $f(\cdot)$, with the goal being that $f(x) = f(t_{\pi(1)}(t_{\pi(2)}(\ldots(t_{\pi(k)}(A(x))))))$.

The intuition is that by randomly selecting k out of n transforms, where each transform is itself randomized, and applying them in a random order, we create a defense that the adversary A can not easily defeat. We focused on this randomness on top of randomness because it provides a mechanism that the adversary can not easily deal with, even if they have perfect knowledge of all transformations t_i and the parameters p_i that alter their behavior. The space of possible actions is too large to find a single alteration $\widetilde{x} = A(x)$ such that the attacker will successfully induce an error by the model for all permutations π and parameterizations $p_{\pi(\ldots)}$.

The transforms we use are listed below. There are five singleton-groups (a group that has only one transform member, $|G_i| = 1$). When a group has more than one constituent transform, we randomly select a transform from the group to act as the group's representative, selecting a new representative on every application. This is to prevent the choice of multiple transformations which all have very similar effects from being applied at the same time, thereby increasing the diversity of changes made to each input.

We emphasize that for every individual transform we evaluate in this work, we have independently tested the transform and achieved 100% evasion success against it using the attack methodology outlined in subsection 4.1. As such, we know that all of these defenses are insufficient in isolation. Thus it is their stochastic combination that makes them significantly stronger than any constituent member. This is counter to previous conclusions that ensembling defenses are not effective and only as strong as the strongest individual defense in the ensemble [6]. The critical difference between our own and prior ensembling defense work is is the number of defenses (25 weak defenses, compared to ≤ 3 for most prior work), and the use of randomness to select subsets of defenses in random orderings.

We employ 25 transforms in total, and so only briefly describe the larger groups here. Further explanation, and Python code, are provided in the appendix.

Color Precision Reduction Reducing bit-resolution of color was originally proposed as a defense by Xu, Evans, and Qi [27] and later reduced to 0% effectiveness [4]. It works by simply reducing the number of bits used to represent the color space of an image, and was tested down to using just 1 bit of color. We incorporate this approach, and make the transform random in two ways. First, the number of colors will be reduced to a value selected from $\mathcal{U}[8, 200]$. Second, with 50% probability we choose between: 1) using the same number of colors for each channel, or 2) selecting a different random number of colors to be used by each channel.

JPEG Noise Using lossy JPEG compression to introduce artifacts was introduced by Kurakin, Goodfellow, and Bengio [10]. Their work looked at how different values of the JPEG compression level (a range from 1 to 100) reduced the impact of adversarial attacks for different values of $\epsilon \leq 16$. However, it was subsequently defeated, having 0% effectiveness [4]. When using this approach, we randomize it by selecting the compression level from $\mathcal{U}[55, 95]$.

Swirl We introduce a simple defense which is to apply a weak swirl to the image, rotating the pixels around a randomly selected point in the image. The radius of intensity is randomly selected from $\mathcal{U}[10, 200]$, and strength from $\mathcal{U}[0.1, 2.0]$.

Noise Injection In early work Tabacof and Valle [28] looked at the impact of addition Gaussian noise on adversarial attacks. We extend this by randomly selecting from Gaussian, Poisson, Salt, Pepper, Salt & Pepper, and Speckle noise to be inserted. With a 50% chance we will either: 1) apply the same noise to every channel, or 2) apply a randomly selected noise type to each channel independently.

FFT Perturbation We introduce a defense built around perturbing the 2D FFT of each channel of the

input image separately. In the frequency domain of the image, we scale all coefficients by a value sampled from $\mathcal{U}[0.98, 1.02]$ (used for all channels). Then for each channel, we randomly choose between 1) zeroing out random coefficients of the FFT, or 2) zeroing out the lowest frequency coefficients of the FFT. The proportion of coefficients that will be set to zero is a random value selected from $\mathcal{U}[0.0, 0.95]$. After altering the coefficients in the frequency domain we return a new, modified image in the spatial domain.

Zoom Group We consider two transforms that have the effect of zooming into the image. To prevent "over zooming" into the image, they are grouped and only one is selected from the group at each step. A simple zoom into a random portion of the image is done, similar to prior work [13], as well as a content-aware zoom based on seam carving [29].

Color Space Group We include four transforms that operate by altering the channels of the image by adding a random constant value, but provide larger impact by first converting the image from RGB to a different color space, and then converting back to RGB after modification. While a more difficult approach would be to allow every pixel in every color coordinate to receive a different value, we intentionally choose the simpler constant value to aid our adversary. This approach is applied to the HSV, XYZ, LAB, and YUV color spaces as the four transform members of this group.

Contrast Group We consider three different types of histogram equalization. Because each one attempts to re-scale and redistribute the values of the histogram of an image to broaden the covered range, they do not make sense to apply in a sequential manner. We use a simple version of Histogram Equalization, an adaptive variant called CLAHE [30], and an approach known as "contrast-stretching."

Grey Scale Group We as humans are usually able to recognize most objects from grey scale imagery, and as such, include conversion to grey scale as one of our defense techniques. For this reason we perform grey-scale transformation four different ways which can be applied selectively to different color channels.

Denoising Group The final group we consider is a number of classical denoising operations and transformations. We group them to avoid over-zealous application that can result in images which appear overly blurry and become difficult to interpret. This includes a Gaussian blur, median, mean, and mean-bilateral [31] filtering, Chambolle and wavelet [32] denoising, and non-local mean denoisng. Prior works have used the median filter [12], wavelet [15, 17], and non-local mean [27] as defenses, but all have since been defeated.

4. Methodology

Given the set of transformations outlined in section 3, we will use a ResNet50 model as our base architecture for experimentation. In particular, we will start with a pre-trained ResNet50 model, and perform an additional 100 epochs of training on ImageNet using Adam[33]. For each dataset in the batch, we randomly pick $k \sim \mathcal{U}[0, 5]$ transformations to apply to each image, so that the model is familiar with the transformations we apply at test time. Following Biggio, Fumera, and Roli [34], we will now fully state the threat model that we will operate in.

Once we have a trained model, our adversary will attack it in three ways: 1), reduce the Top-1 accuracy (any output besides the correct class is a success for the attacker); 2) reduce the Top-5 accuracy (any output is a success for the attacker provided the correct class is ranked sixth or lower), and; 3) increase the targeted success rate. In the first two conditions the attacker can trick the model into any incorrect classification. In the final condition, the attacker has a specific, randomly selected class that it must induce the model into outputting. All of these attacks will be performed on the standard ImageNet validation set.

Our adversary's capability will include making modifications to any input feature of the test data under the L_∞ metric, for which the adversary will attempt to modify the input x to a new input $\hat{x}$ such that $\|x - \hat{x}\|_\infty < \epsilon$. In our experiments, we will test a range of $\epsilon \in [2, 32]$.

We will operate in the white-box scenario, and assume that our adversary has full and complete knowledge of our training data, architecture, weights, and defensive transforms. To perform the attacks, we will use the Fast Gradient Sign Method (FGSM) [8] because it is a common baseline. More importantly, we will also use Projected Gradient Descent (PGD) [26], which is also targeted toward the L_∞ metric and is currently the strongest known attack for this metric. PGD has been conjectured to be a near-optimal first-order attack [25]. We use the FoolBox library for the implementation of these attacks [35].

To further ensure the attacker's strength, we follow recommendations from Demontis, Melis, Pintor, *et al.* [36], and attempt to optimize for the adversary in the L_∞ ball with *maximum confidence*, rather than minimum distance. This includes making sure that the PGD attack runs through all optimization steps, even

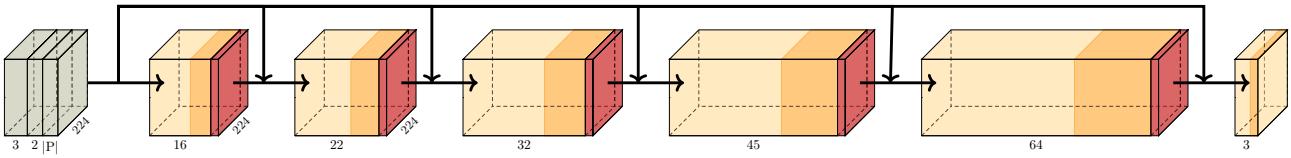

Figure 1: Diagram of BPDA network architecture. Input is of dimension $3+2+|P(t)|$, first three dimensions are the RGB channels, second two are the CoordConv channels, and the last set corresponds to the random parameters that affect the transform $t(\cdot)$'s output. Arrows that connect in the diagram indicate concatenation, yellow is convolution (number of filters below) followed by batch-normalization, and red is the ReLU activation

if the attack appears to have been successful at an earlier iteration. By default all experiments will perform PGD with 40 attack iterations, with stronger attacks in the appendix. Below we will further detail all the steps we take to implement the adversary's attack, so that we fully account for the gradient obfuscation and other issues that have thwarted previous defenses [2, 4].

4.1. Making A Strong Adversary

To maximize the strength of our attacker, we must first resolve two issues. The first is that our transformation process is randomized, which means we can not take the gradient from a single instance of the attack, as the next realization of a transformed image will have a differently parameterized transform. To remedy this situation, we use the Expectation over Transformation (EoT) [11]. The idea of EoT is to perform the transformation multiple times, and take the average gradient over several runs. When we use iterative attacks like PGD, this means for every iteration of the attack we will take the average of several transforms at that step in the attack.

The second issue we have is that not all of our transformations are differentiable. The solution to this problem was proposed by Athalye, Carlini, and Wagner [4], and is called Backward Pass Differentiable Approximation (BPDA). The idea is simple: when a transform $t(\cdot)$ is not itself differentiable, use a neural network to learn a function $f_t(\cdot)$ that approximates the transform. Since it is implemented with a neural network, $f_t(\cdot)$ is differentiable, and so we can use $\nabla f_t(\cdot)$ to obtain a gradient that is useful for the adversary as an approximation to $\nabla t(\cdot)$. This approach is effective, and using a naive identity function $f_t(x) = x$ is often sufficient to defeat many attacks. Indeed, it is enough to defeat most of our transforms individually. However, we learn a small CNN to approximate this gradient to maximize the adversary's advantage.

We also recognize that while we have a repertoire of transforms that are selected from at random, each transform t itself is randomized as well. Denoting the set of parameters of a transform t as $P(t)$, the transform is deterministic given a specific realization $p \sim P(t)$ of these transforms. To learn our model $f_t(\cdot)$, we will create an input that has $5 + |P(t)|$ channels, and is the same size as the image to be learned. The first three channels will be the RGB channels of the original image. The next two channels will be the CoordConv values proposed by Liu, Lehman, Molino, *et al.* [37], so that our networks can deal with location specific transformations. We found CoordConv necessary in our BPDA model to effectively approximate the Swirl transform. The remaining $|P(t)|$ channels will each have a constant value, which is the value of the realized parameters p. Placing the random values p each into their own distinct channel provides a mechanism for us to allow the network $f_t(\cdot)$ to learn the fully specified deterministic mapping. Each CNN $f_t(\cdot)$ has 6 convolutional layers, followed by batch-normalization and then a ReLU activation. For each layer we include a skip connection from the input, following the DenseNet approach. (See Figure 1). We train the network as a denoising auto-encoder, where the target is the parameterized transform of the image (i.e., the loss is $\|f_t(x,p) - t(x|p)\|_2^2$), with 100 epochs of training for all 25 BPDA networks. Once $f_t(\cdot)$ is trained, we perform BPDA by back-propagating through $f_t(\cdot)$ to the first 3 channels that correspond to the original image RGB values.

Combining BPDA and EoT as we have described above, we can defeat any of our transforms individually 100% of the time for both targeted and un-targeted attacks. This confirms that we have implemented these approaches appropriately, and have maximized the strength of our adversary.

As part of our evaluation, we also wish to address a concern raised by Madry, Makelov, Schmidt, *et al.* [25], which is the computational cost of a threat model. They argue that the strength of an adversary should be in some way computationally constrained, in the same manner that cryptographic problems are secure because we assume the adversary does not have the dramatic compute resources necessary to attack a given encryption scheme. Using 10 iterations of EoT combined with the iterative nature of PGD (40 optimization steps)

means we must perform 400 gradient calculations per attack, combined with the time to compute the image transformations and back-propagate through the additional BPDA networks. This takes about 48 hours per experiment given a workstation with 10 CPU cores and a Titan X GPU. We will also consider results with 40 EoT iterations — the highest we have observed in the literature — to evaluate if an even stronger adversary would be significantly more successfully, but these experiments required 240 hours *each* on a DGX-1. In total, the results presented in this paper consumed approximately 320 days on our DGX-1. We include tests at both of these EoT scales to help confirm our attack is robust, and simply increasing the number of iterations of the adversary does not dramatically change results. We also feel we are approaching a limit of reasonable compute for an adversary to have, and would be the largest barrier to replication if we pushed to even more attack iterations.

4.1.1. Medoid over Transformations

We take a moment to define a new type of attack to help ensure that we are not inadvertently engaging in accidental obfuscation of gradients. In particular, we note that the expectation over transformation approach uses the mean gradient over some transformation, shown more formally in Equation 1 where z_{EoT} is the number of iterations of the EoT sampling and $t^{(i)}$ is a deterministic realization of the transform t (i.e., the pseudo random number generator has been seeded with the value i to provide a deterministic result).

$$\nabla \mathbb{E}_{t^{(i)} \sim t} f(t^{(i)}(x)) \approx \frac{1}{z_{\text{EoT}}} \sum_{i=1}^{z_{\text{EoT}}} \nabla f(t^{(i)}(x)) \qquad (1)$$

One possible source of gradient obfuscation might be that the mean of the distribution does not exist or is not well defined. This notion comes from the recognition that in our larger framework t in Equation 1 corresponds to our entire randomized pipeline of selecting k transforms from $G_{1,\ldots,10}$. Our concern by analogy is that we could have a situation similar to the Cauchy distribution: The mean of the Cauchy distribution does not exist; every empirical mean is equally likely. However, one can successfully estimate the Cauchy distribution's position by instead using the median.

With this notion in mind, we include a new *Medoid over Transformations* (MoT) estimate, in which we use the medoid of the gradients of a sample of z_{MoT} transformations as our gradient estimate to perform attacks with.

$$\underset{i}{\text{argmin}} \sum_{j=1}^{z_{\text{MoT}}} \|\nabla f(t^{(i)}(x)) - \nabla f(t^{(j)}(x))\|_2^2 \qquad (2)$$

If we are accidentally performing gradient obfuscation by instead pushing information from the mean to the medoid — similar to the behavior of a Cauchy distribution — we would expect to see an increase in performance with the MoT attack compared to the EoT. Our results will confirm that this is not the case, as the MoT attack performs worse than the EoT attack. However, we include the results and attack description here to build further confidence that we have attempted to make the strongest attack possible.

5. Results

Few people have had their defense stand up to further testing due to a variety of issues related to obfuscated gradients and failing to fully account for all components of the defense when designing the white-box adversary. Fewer still have been able to scale their defensive techniques up to the ImageNet dataset. Now that we have defined our methodology to make sure we have accounted for both obfuscated gradients and ensure our adversary has fully captured the defense in their attack, we will show how we obtain new state-of-the-art results on the ImageNet dataset, as well as the associated costs in achieving such performance.

Kurakin, Goodfellow, and Bengio [26] provide the strongest results on the full ImageNet dataset that we are aware of. They do this with adversarial training, which they noted had great difficulty scaling up to the ImageNet corpus. At a maximum perturbation of $\epsilon = 16$, they achieved only a Top-1 accuracy of 1.5% and a Top-5 accuracy of 5.5% when under attack by PGD.

For BaRT, we will by default assume $\epsilon = 16$, the number of transformations $k = 5$, and the number of EoT runs will be 10. In our experiments we will investigate changing all of the values to observe their impact on our effectiveness against attack. We begin by showing the accuracy of our methods in Table 1. Here we can see the first immediate down side to BaRT, which is a significant reduction in accuracy if our model is not under attack. The off-setting benefit is the first significant improvement in accuracy when the model is under attack. At a cost of increased runtime, it is possible to create multiple inferences of an input by applying the transform $t(\cdot)$ multiple times, and then classifying each differently transformed version of the image. This creates an ensemble effect, and removes any loss in accuracy due to BaRT's application. Due to space, details on ensembling BaRT are left to Appendix E.

5.1. Experiments

The immediate product of our work is that the BaRT strategy provides a 9.3–24 times improvement in accuracy on ImageNet compared to prior state of the art.

Table 1: Accuracy (%) of baseline prior work on adversarial training [26] and BaRT. 'Clean Images' is the results of classifying non-attacked images without any transforms; 'Attacked' shows results when using PGD with $\epsilon = 16$.

	Clean Images		Attacked	
Model	Top-1	Top-5	Top-1	Top-5
Inception v3	78	94	0.7	4.4
Inception v3 w/Adv. Train	78	94	1.5	5.5
ResNet50	76	93	0.0	0.0
ResNet50-BaRT, $k = 5$	65	85	16	51
ResNet50-BaRT, $k = 10$	65	85	36	57

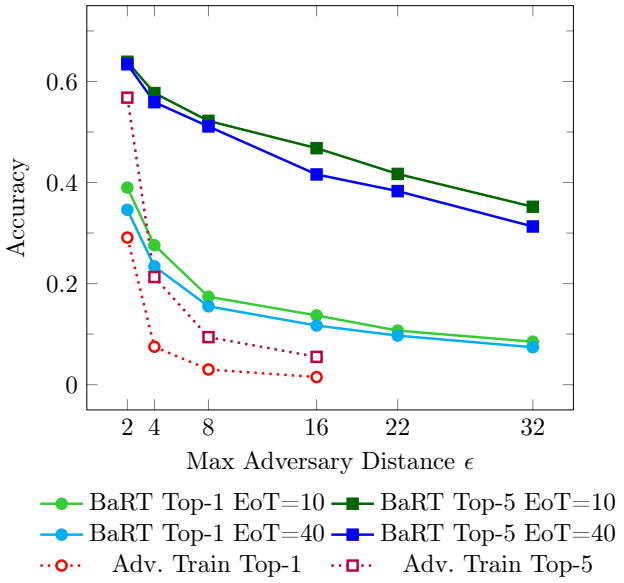

Figure 2: Accuracy of model under attack by PGD for varying adversarial distances ϵ with EoT steps=$\{10,40\}$.

We now further investigate the differing parameters of our defense. First we look at the larger range of ϵ, the bound on the adversary's freedom to alter the input. In Figure 2 we show accuracy under PGD attack as ϵ varies from 2 to 32. We note that $\epsilon = 16$ is the largest we have observed in any prior work, and is considered a powerful adversary. We are the first to test $\epsilon = 32$, and still show non-trivial robustness to attack.

In these results we see that BaRT dominates adversarial training across all values of ϵ. We also see that adversarial training degrades quickly as ϵ moves from just 2 to 4. In contrast BaRT Top-1 and Top-5 accuracy when attacked with $\epsilon = 32$ is still better than the results with adversarial training and $\epsilon = 4$. For $\epsilon > 2$, BaRT also shows Top-1 accuracy higher than adversarial training's Top-5 accuracy.

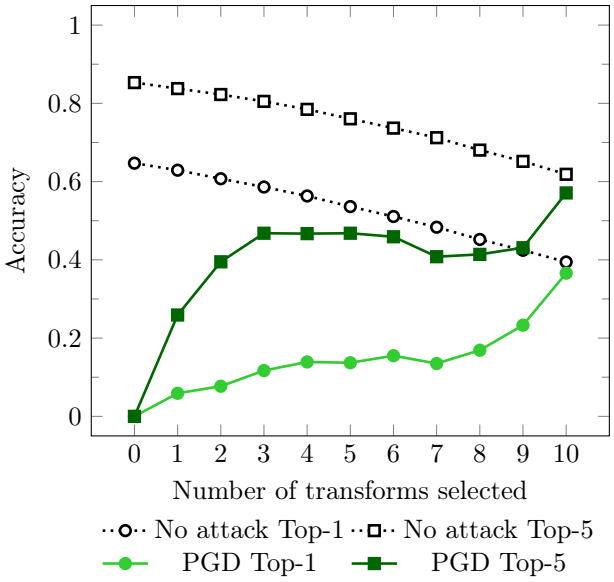

Figure 3: Accuracy of model when varying the number of transforms used, both when not under attack and when being attacked by PGD.

These results also demonstrate that while increasing the number of EoT steps does increase the adversary's success rate, the difference is not large. Using 40 steps already requires a level of compute not reasonable for most institutions, and gives us confidence that other attempts to simply throw even more compute to the adversary will be nonviable. This is before we consider that it is relatively easy to write these transformations, and we could add even more transformations to the pipeline to further impede the adversary's compute requirements and reduce their success rate. While we do not have the resources to test exhaustively, we show in Appendix F that using even 520 PGD steps shows no significant change in the attacker's success rate.

Next we investigate the number of transforms applied. For these results, we remind the reader that BaRT was only trained with up to $k = 5$ transforms applied to the training data. In Figure 3 we plot the Top-1 and Top-5 accuracy of BaRT (under attack and on clean images) as a function of the number of transforms k selected at test time. Our initial expectation was that we would see the best performance (i.e., greatest accuracy under attack) when $k = n/2$, as this would maximize the number of combinatorial paths $\binom{n}{k}$. However, this was not the case.

Instead we see that every transformation $t_i(\cdot)$ we apply produces some associated costs and benefits. The benefit is that increasing $k \to n$ improves our performance when under attack. A slight dip occurs after

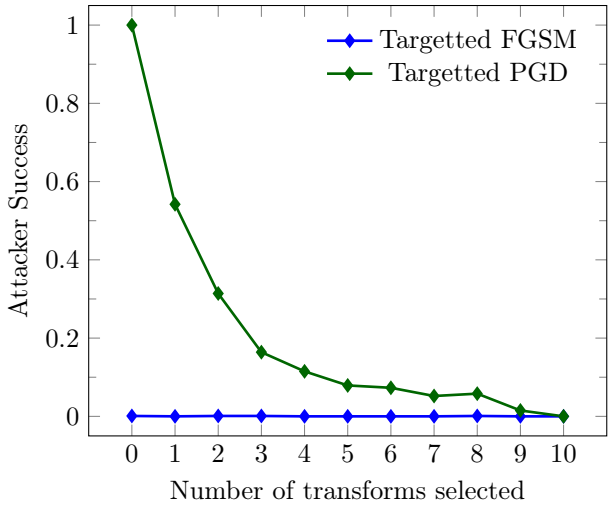

Figure 4: Attacker success rate against BaRT model when varying the number of transforms used, for both FGSM and PGD attacks with $\epsilon = 16$.

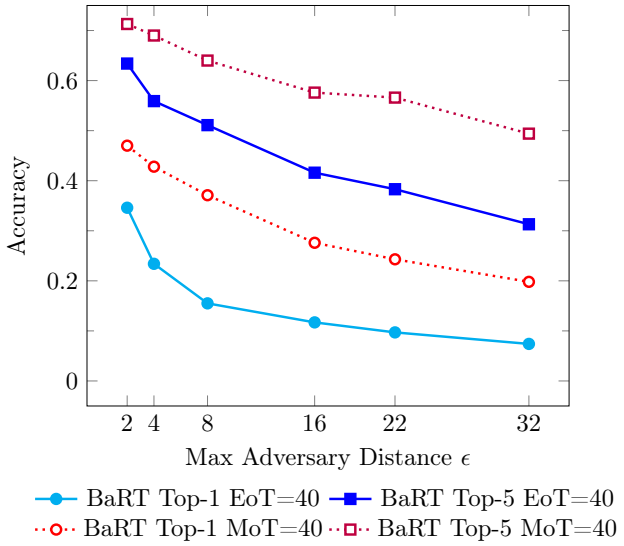

Figure 5: Accuracy of model under attack by EoT and MoT versions of PGD for varying adversarial distances ϵ and EoT steps=40.

$k = 5$ transformations are applied, which we expect is related to using $k \leq 5$ during training. One can also observe a steady decrease in the accuracy on non-attacked, clean images as k increases, which happen to almost intersect at $k = 10$. Adding more transformations also has an impact on run-time, but calculating the transformations is fast relative to the cost of needing a GPU for CNN inference, and is approximately three orders of magnitude faster than running the attacks.

Overall this validates that an ensemble of weak defenses can form a single strong defense, provided that the ensemble is applied in a random fashion. We also see that maximizing the combinatorial search space is not a dominating strategy, since we see maximal adversarial robustness at $k = 10$ instead of $k = 5$. This tells us that the amount of transformation applied to the image is also an important component of defeating the adversary, as this is maximized at $k = 10$. Cumulatively, we could argue that selecting the value of k to use in practice should be a function of the likelihood of being under attack. If a model is continuously under attack or needs maximal worst-case performance, one should choose $k = 10$ because the non-attacked accuracy is not meaningful when under attack.

We also explore the impact on targeted adversarial attacks in Figure 4, where we look at the attacker's success rate as a function of k. Here we can see that when no transformations are present, PGD attack achieves 100% success rate against the model, but quickly degrades as transformations are added — reaching 0.0% success at $k = 10$ transformations. We note that additional runs may produce values near zero instead of at zero, but it suffices to show that the ability for the adversary to perform targeted attacks can be almost completely impeded by our BaRT defense.

Lastly, in subsubsection 4.1.1 we considered the possibility that we might be engaging in obfuscated gradients by moving information to the medoid of the distribution. We developed a new Medoid over Transformation attack to test this hypothesis. The results are shown in Figure 5. While MoT does produce adversarial examples, it has uniformly worse performance compared to using the mean gradient. As such we further conclude that we have not relied on obfuscated gradients, and that our defense is effective.

6. Conclusion

We have introduced BaRT, a strategy for defending image classifiers against attack by randomly selecting a few transforms from a large pool of stochastic transformations, and apply each in a random order before processing the image. This scales to datasets like ImageNet, and provides state-of-the-art results when under attack even after accounting for all known obfuscated gradients. While heuristic in nature, our results provide evidence that a strong defense can be made from many weaker ones, and indicates strategic applications of randomness may benefit future work.

Acknowledgments We would like to thank Battista Biggio for reviewing a draft of this work and providing insightful comments and feedback.

References

[1] B. Biggio and F. Roli, "Wild patterns: Ten years after the rise of adversarial machine learning," *Pattern Recognition*, vol. 84, pp. 317–331, Dec. 2018. DOI: 10.1016/j.patcog.2018.07.023.

[2] N. Carlini and D. Wagner, "Adversarial Examples Are Not Easily Detected: Bypassing Ten Detection Methods," in *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*, ser. AISec '17, New York, NY, USA: ACM, 2017, pp. 3–14. DOI: 10.1145/3128572.3140444.

[3] K. Eykholt, I. Evtimov, E. Fernandes, B. Li, A. Rahmati, C. Xiao, A. Prakash, T. Kohno, and D. Song, "Robust Physical-World Attacks on Deep Learning Models," in *Computer Vision and Pattern Recognition (CVPR)*, 2018.

[4] A. Athalye, N. Carlini, and D. Wagner, "Obfuscated Gradients Give a False Sense of Security: Circumventing Defenses to Adversarial Examples," in *International Conference on Machine Learning (ICML)*, 2018.

[5] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015. DOI: 10.1007/s11263-015-0816-y.

[6] W. He, J. Wei, X. Chen, N. Carlini, and D. Song, "Adversarial Example Defenses: Ensembles of Weak Defenses Are Not Strong," in *Proceedings of the 11th USENIX Conference on Offensive Technologies*, ser. WOOT'17, Berkeley, CA, USA: USENIX Association, 2017.

[7] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," in *ICLR*, 2014. DOI: 10.1021/ct2009208.

[8] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and Harnessing Adversarial Examples," in *International Conference on Learning Representations (ICLR)*, 2015.

[9] J. Lu, H. Sibai, E. Fabry, and D. Forsyth, "No Need to Worry about Adversarial Examples in Object Detection in Autonomous Vehicles," in *The First Workshop on Negative Results in Computer Vision. CVPR 2017.*, 2017.

[10] A. Kurakin, I. Goodfellow, and S. Bengio, "Adversarial examples in the physical world," in *ICLR Workshop*, 2017.

[11] A. Athalye, L. Engstrom, A. Ilyas, and K. Kwok, "Synthesizing Robust Adversarial Examples," 2017.

[12] X. Li and F. Li, "Adversarial Examples Detection in Deep Networks with Convolutional Filter Statistics," in *2017 IEEE International Conference on Computer Vision (ICCV)*, IEEE, Oct. 2017, pp. 5775–5783. DOI: 10.1109/ICCV.2017.615.

[13] C. Guo, M. Rana, M. Cissé, and L. Van Der Maaten, "Countering Adversarial Images Using Input Transformations," in *International Conference on Learning Representations (ICLR)*, 2018.

[14] P. Samangouei, M. Kabkab, and R. Chellappa, "Defense-GAN: Protecting Classifiers Against Adversarial Attacks Using Generative Models," in *International Conference on Learning Representations (ICLR)*, 2018.

[15] A. Prakash, N. Moran, S. Garber, A. DiLillo, and J. Storer, "Deflecting Adversarial Attacks with Pixel Deflection," in *CVPR*, 2018. DOI: 10.1109/CVPR.2018.00894.

[16] F. Liao, M. Liang, Y. Dong, T. Pang, X. Hu, and J. Zhu, "Defense against Adversarial Attacks Using High-Level Representation Guided Denoiser," in *CVPR*, 2018.

[17] A. Athalye and N. Carlini, "On the Robustness of the CVPR 2018 White-Box Adversarial Example Defenses," *arXiv*, 2018.

[18] C. Xie, Z. Zhang, A. L. Yuille, J. Wang, and Z. Ren, "Mitigating Adversarial Effects Through Randomization," in *International Conference on Learning Representations (ICLR)*, 2018.

[19] H. Kannan, A. Kurakin, and I. Goodfellow, "Adversarial Logit Pairing," *arXiv*, 2018. DOI: 10.4103/0972-124X.94617.

[20] L. Engstrom, A. Ilyas, and A. Athalye, "Evaluating and Understanding the Robustness of Adversarial Logit Pairing," *arXiv*, 2018.

[21] E. Wong and Z. Kolter, "Provable Defenses against Adversarial Examples via the Convex Outer Adversarial Polytope," in *Proceedings of the 35th International Conference on Machine Learning*, J. Dy and A. Krause, Eds., ser. Proceedings of Machine Learning Research, vol. 80, Stockholmsmässan, Stockholm Sweden: PMLR, 2018, pp. 5283–5292.

[22] M. Abbasi and C. Vision, "Certified Defenses Against Adversarial Examples," in *International Conference on Learning Representations (ICLR)*, 2018.

[23] K. Dvijotham, R. Stanforth, S. Gowal, T. Mann, and P. Kohli, "A Dual Approach to Scalable Verification of Deep Networks," in *Conference on Uncertainty in Artificial Intelligence (UAI)*, 2018.

[24] E. Wong, F. Schmidt, J. H. Metzen, and J. Z. Kolter, "Scaling provable adversarial defenses," *ArXiv e-prints*, 2018.

[25] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards Deep Learning Models Resistant to Adversarial Attacks," in *ICLR*, 2018.

[26] A. Kurakin, I. Goodfellow, and S. Bengio, "Adversarial Machine Learning at Scale," in *International Conference on Learning Representations (ICLR)*, 2017.

[27] W. Xu, D. Evans, and Y. Qi, "Feature Squeezing: Detecting Adversarial Examples in Deep Neural Networks," in *Proceedings 2018 Network and Distributed System Security Symposium*, Reston, VA: Internet Society, 2018. DOI: 10.14722/ndss.2018.23198.

[28] P. Tabacof and E. Valle, "Exploring the space of adversarial images," in *2016 International Joint Conference on Neural Networks (IJCNN)*, IEEE, Jul. 2016, pp. 426–433. DOI: 10.1109/IJCNN.2016.7727230.

[29] S. Avidan and A. Shamir, "Seam Carving for Content-aware Image Resizing," in *ACM SIGGRAPH 2007 Papers*, ser. SIGGRAPH '07, New York, NY, USA: ACM, 2007. DOI: 10.1145/1275808.1276390.

[30] K. Zuiderveld, "Contrast Limited Adaptive Histogram Equalization," in *Graphics Gems IV*, P. S. Heckbert, Ed., San Diego, CA, USA: Academic Press Professional, Inc., 1994, ch. Contrast L, pp. 474–485.

[31] B. Weiss, "Fast Median and Bilateral Filtering," in *ACM SIGGRAPH 2006 Papers*, ser. SIGGRAPH '06, New York, NY, USA: ACM, 2006, pp. 519–526. DOI: 10.1145/1179352.1141918.

[32] S. Chang, Bin Yu, and M. Vetterli, "Adaptive wavelet thresholding for image denoising and compression," *IEEE Transactions on Image Processing*, vol. 9, no. 9, pp. 1532–1546, 2000. DOI: 10.1109/83.862633.

[33] D. P. Kingma and J. L. Ba, "Adam: A Method for Stochastic Optimization," in *International Conference On Learning Representations*, 2015.

[34] B. Biggio, G. Fumera, and F. Roli, "Security evaluation of pattern classifiers under attack," *IEEE Transactions on Knowledge and Data Engineering*, vol. 26, no. 4, pp. 984–996, 2014. DOI: 10.1109/TKDE.2013.57.

[35] J. Rauber, W. Brendel, and M. Bethge, "Foolbox: A Python toolbox to benchmark the robustness of machine learning models," *arXiv preprint arXiv:1707.04131*, 2017.

[36] A. Demontis, M. Melis, M. Pintor, M. Jagielski, B. Biggio, A. Oprea, C. Nita-Rotaru, and F. Roli, "Why Do Adversarial Attacks Transfer? Explaining Transferability of Evasion and Poisoning Attacks," *ArXiv e-prints*, pp. 26–28, 2018.

[37] R. Liu, J. Lehman, P. Molino, F. P. Such, E. Frank, A. Sergeev, and J. Yosinski, "An Intriguing Failing of Convolutional Neural Networks and the CoordConv Solution," pp. 1–24, 2018.

[38] A. Chambolle, "An Algorithm for Total Variation Minimization and Applications," *J. Math. Imaging Vis.*, vol. 20, no. 1-2, pp. 89–97, Jan. 2004. DOI: 10.1023/B:JMIV.0000011325.36760.1e.

[39] J. Darbon, A. Cunha, T. F. Chan, S. Osher, and G. J. Jensen, "Fast nonlocal filtering applied to electron cryomicroscopy," in *2008 5th IEEE International Symposium on Biomedical Imaging: From Nano to Macro*, IEEE, May 2008, pp. 1331–1334. DOI: 10.1109/ISBI.2008.4541250.

[40] L. I. Kuncheva and C. J. Whitaker, "Measures of diversity in classifier ensembles and their relationship with the ensemble accuracy," *Machine learning*, vol. 51, no. 2, pp. 181–207, 2003.

[41] C. Ju, A. Bibaut, and M. van der Laan, "The relative performance of ensemble methods with deep convolutional neural networks for image classification," *Journal of Applied Statistics*, vol. 45, no. 15, pp. 2800–2818, 2018.

A. Checking for Vanishing and Exploding Gradients

In this work, we have intentionally restricted ourselves to a class of defenses that are currently known to be broken against powerful adversaries, and implemented the known attack methods that have succeeded against them. This has enabled us to show that an ensemble of weak defenses, combined with appropriate stochastic measures, comprises a more powerful collective defense than its individual constituents.

We have spent considerable effort looking for any possible form of obfuscated gradient to ensure that we have not succeeded through inadvertently withholding information from the adversary. As our last check against this issue, we look for vanishing and/or exploding gradients. This is a common problem with many types of neural networks that can cause failure to converge due to numerical instability, and was found to be an inadvertent source of obfuscation in prior works [4]. In Table 2 we show statistics on the norm of the 40 gradient steps used by PGD to attack a single image.

The values at $k = 0$ are shown for the ResNet50 model that comes pre-trained in pyTorch, and values $k = 1$ through $k = 10$ are with our fine-tuned model with one through ten transforms selected. Looking at the mean norm of the gradient, we see that it starts out at a value of 57.65, which looks more like an exploding gradient and would be clipped back to the ϵ ball of the PGD iteration. As k increase the mean norm decrease to a more reasonable range of values. We see no evidence for exploding or vanishing gradients that would cause numerical instability and cause a failure for the attacker's optimization process. In fact, we see more unstable gradients when $k = 0$, before we have ever introduced our attacks. Here we see a minimum magnitude of 0.0000, and a maximum of 330.87. The range of gradient magnitude shrinks as k increases, making the problem fundamentally more numerically stable. This is caused largely because of the averaging of 40 gradients by the EoT process, which reduces the impact of large and small magnitude outliers on the PGD steps. Another trend is that as k increases, we see the minimum norm of the gradient increase. This makes intuitive sense, as larger k corresponds to a larger combination of possible defensive transforms, necessitating more work from the adversary to circumvent.

Table 2: Statistics on norm of the gradient during PGD search. Measure over 40 PGD attack steps, and using 40 EoT steps for each gradient estimate, across 1000 images from the ImageNet validation set.

k	min	max	mean	std	median
0	0.00	330.87	57.65	26.96	55.09
1	0.00	108.53	21.27	9.96	20.24
2	0.00	45.09	10.96	5.06	10.49
3	0.01	60.03	6.97	3.11	6.65
4	0.01	37.56	5.07	2.18	4.84
5	0.12	40.88	3.92	1.76	3.66
6	0.17	39.76	3.02	1.41	2.79
7	0.16	47.31	2.32	1.15	2.11
8	0.13	37.74	1.85	1.01	1.64
9	0.15	33.78	1.52	0.92	1.32
10	0.14	33.08	1.30	0.86	1.10

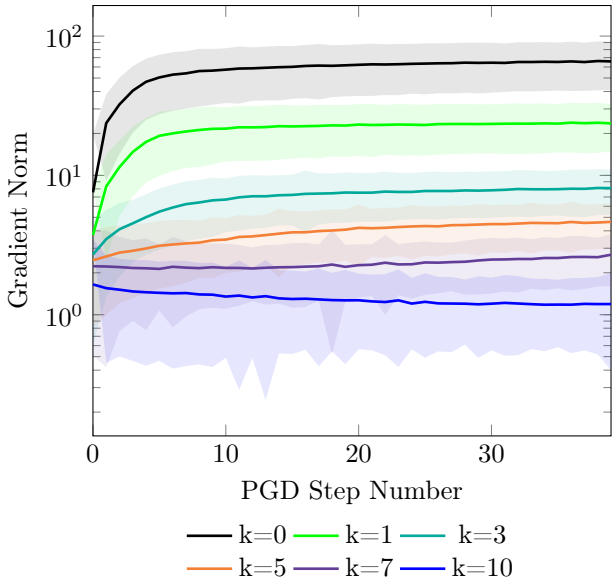

Figure 6: The y-axis (log-scale) shows the L_2 norm, and x-axis which sequential PGD step's gradient is considered. Standard deviation is shown in a lighter shaded region around each plot, for $k = 0, 1, 3, 5, 7, 10$ transforms being used in the defense. 40 EoT steps where used for $k \geq 0$.

In Figure 6 we look at the norm of the gradient used by PGD across the PGD steps. Here it is clear that after 5 steps, the average norm stabilizes around some value with a large standard deviations, regardless of the value of k. As k increases, the average norm decreases and is consistent.

B. Why BaRT Works as a Defense

The results presented demonstrate that BaRT provides an effective, though not perfect, defense against adversarial attack. Throughout testing we believe we have eliminated the possibility of an obfuscated gra-

Table 3: Statistics on absolute cosine similarity between successive steps of PGD. Statistics collected from PGD with 40 iterations and 40 EoT steps per gradient, run over 1000 images from the ImageNet test set.

k	min	max	mean	std	median
0	0.000	0.561	0.283	0.111	0.302
1	0.000	0.671	0.374	0.137	0.411
2	0.000	0.571	0.265	0.114	0.279
3	0.000	0.561	0.109	0.083	0.092
4	0.000	0.519	0.075	0.060	0.062
5	0.000	0.364	0.066	0.044	0.058
6	0.000	0.266	0.050	0.033	0.044
7	0.000	0.223	0.034	0.024	0.030
8	0.000	0.152	0.023	0.017	0.020
9	0.000	0.171	0.017	0.013	0.014
10	0.000	0.145	0.016	0.013	0.013

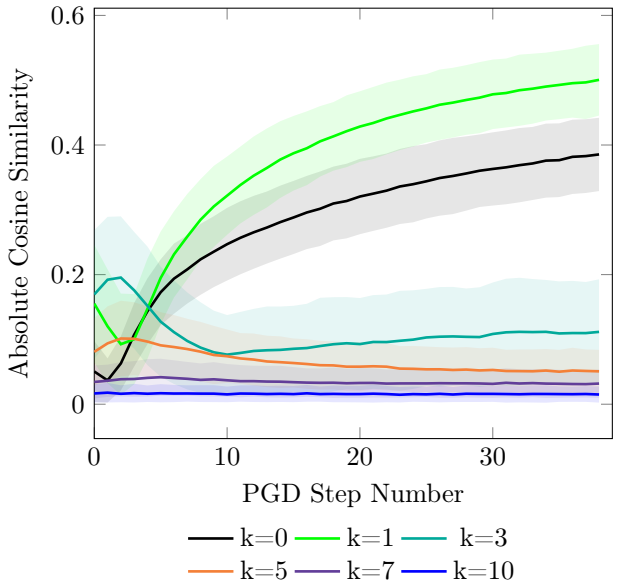

Figure 7: The y-axis shows the absolute cosine similarity, and x-axis which pair of successive PGD steps are being compared. Standard deviation is shown in a lighter shaded region around each plot, for $k = 0, 1, 3, 5, 7, 10$ transforms being used in the defense. 40 EoT steps were used for $k \geq 0$.

dient as a source of misleading positive results. The question then becomes: *why does BaRT work?*

As we have explained, our intuition behind the BaRT defense's effectiveness is that it is not always possible for the adversary to find a single alteration that can simultaneously satisfy the large number of possible transformation combinations. The search space becomes large, and the randomized nature of a variety of transformations make it so that (hopefully) changes to support one set of transforms fail to be effective for a different set of transforms. Since the selection is random, the adversary is left with few winning options.

We can empirically test this hypothesis by looking at the gradients of successive PGD steps during the attack process. If the absolute cosine similarity between successive steps is near 1.0, it means there is a straight path from the original starting image to one that successfully fools the victim model. If it is zero, it means there is no information in the gradient at all, and the PGD attack is instead performing a type of random search. We plot statistics of the absolute cosine similarity between successive PGD steps in Table 3.

Here we can see a clear progression of behavior. For $k \leq 2$, the cosine similarity is a relatively large value (≈ 0.3), indicating that the gradient direction between steps is related but the path taken adjusts direction as well. This makes sense and is part of why PGD is more effective than FGSM: if a single direction was sufficient, FGSM with a larger step size would be equally effective.

As k increases toward 10, we see that the mean and max cosine similarity between successive steps begins to decrease and approach 0. This indicates that the PGD attack is heading in a nearly orthogonal direction at consecutive steps. We have taken all steps to ensure the gradient can be back-propagated through transformations using EoT and BPDA, and that the gradients are not vanishing or exploding. As such, this provides empirical evidence that our hypothesis is correct: The PGD attack is not able to make successful adversarial examples because there is no single perturbation to the input that can satisfy a large number of different and randomly applied transformations to the image.

We further explore the behavior of the similarity of PGD gradient directions comparing step-by-step gradients in Figure 7. We consider the 40 PGD steps sequentially, and compare the cosine similarity between just successive pairs of steps, averaged across 1000 images randomly selected from the test set. Here we can see that when $k \leq 1$, the similarity between successive steps starts out small, and later increases as the PGD optimization process finds a path toward successfully fooling the model.

As k increases, we see a significant change in behavior. The relation between successive PGD steps starts out near its highest, and then tends toward zero after a few steps of PGD. The larger k becomes, the more depressed the similarity of adjacent gradients. Because we have determined that the norm of the gradients is in a numerically stable range and has not exploded or vanished, it appears PGD is unable to find an pertur-

bation that successfully attacks the variety of possible transformations and their combinations.

While these results do not provide proof that BaRT will always be successful, we find them informative to understanding the nature of how BaRT provides improvements in detection under attack.

C. Transformation Details

In this section of the appendix we will provide further details of all 25 transformations used in our work. For transformations that were briefly mentioned because they were members of a group, we will provide similar short textual description.

More importantly, we extract the code from each transformation used in our code base. We hope to release the full source code in the future. Each code snippet is extracted from a class' `def transform(self, img)` function, which takes in the image object as a numpy array of size $224 \times 224 \times 3$, where the first two dimensions are width and height, and the last dimension gives the red, green, and blue channels in that order.

As part of the contract of the transform method, the return value will be a tuple of 1) the newly transformed image, and 2) an ordered list of the randomly selected parameter values for the transform. The purpose of returning the parameter values is so that they can be used when training the BPDA networks. The length of the returned list will be the number of extra channels $|P(t)|$ added to the associated BPDA network, and each channel will be filled with the value returned in the list. When we return the parameter values in the code, we normalize them so that they are in the range of $[0, 1]$, and booleans are converted to 0 and 1 exactly.

The function definitions assume a number of standard imports for python libraries, such as numpy and scikit-image. A number of the functions also make use of the three helper functions for randomly sampling values shown below:

```
def randUnifC(low, high, params=None):
    p = np.random.uniform()
    if params is not None:
        params.append(p)
    return (high-low)*p + low

def randUnifI(low, high, params=None):
    p = np.random.uniform()
    if params is not None:
        params.append(p)
    return round((high-low)*p + low)

def randLogUniform(low, high, base=np.exp(1)):
    div = np.log(base)
    return base**np.random.uniform(np.log(low)/div,
     ↪  np.log(high)/div)
```

For each transform, we will include an image from the ImageNet validation set of an adorable kitten, followed by examples of that transformation applied to the kitten. We use the kitten because it is adorable.[2] The original kitten image will be the leftmost image of each trio, and the center and right images are randomly selected transformations of the kitten.

Figure 8: Color Precision Reduction

The distribution of parameters for each transform were adjusted based on a small sample of 10 images from the training set. The distributions were adjusted to the point that, subjectively, we felt we could reliably tell what the image was after transformation. Tuning the distributions more rigorously may allow one to optimize the performance of BaRT when under attack or not under attack, but we leave that for future research.

The first five transforms do not belong to larger groups. Since they are fully described in section 3, we include only the related code here. For the rest of the transforms we include both code and a more detailed description than can be found in the main body.

C.1. Color Precision Reduction

This transformation alters images by reducing the color depth. The number of resulting channels is chosen from $\mathcal{U}[8, 200]$. With 50% probability, we reduce all three color channels by an equal amount, or alter each channel independently. In the future, more advanced color quantization algorithms could be examined.

```
scales = [np.asscalar(np.random.random_integers(8,
 ↪  200)) for x in range(3)]
multi_channel = np.random.choice(2) == 0

params = [multi_channel] + [s/200.0 for s in scales]

if multi_channel:
    img = np.round(img*scales[0])/scales[0]
else:
    for i in range(3):
        img[:,:,i] = np.round(img[:,:,i]*scales[i]) /
         ↪  scales[i]

return img, params
```

[2] Some of the authors feel that a dog should have been chosen.

Figure 9: JPEG Noise

Figure 10: Swirl

C.2. JPEG Noise

In this transformation, the image is encoded at a lower JPEG quality level — chosen from $\mathcal{U}[55, 95]$ — and then re-loaded.

```
quality = np.asscalar(np.random.random_integers(55,
 ↪  95))

params = [quality/100.0]

pil_image = PIL.Image.fromarray(
 ↪  (img*255.0).astype(np.uint8) )
f = BytesIO()
pil_image.save(f, format='jpeg', quality=quality)
jpeg_image = np.asarray( PIL.Image.open(f)
 ↪  ).astype(np.float32) / 255.0

return jpeg_image, params
```

C.3. Swirl

Using the scikit-image package, each image is "swirled" by some amount to create a "whirlpool" effect. The angle of rotation, center of rotation, and radius of effect are all randomized.

```
strength = (2.0-0.01)*np.random.random(1)[0] + 0.01
c_x = np.random.random_integers(1, 256)
c_y = np.random.random_integers(1, 256)
radius = np.random.random_integers(10, 200)

params = [strength/2.0, c_x/256.0, c_y/256.0,
 ↪  radius/200.0]

img = skimage.transform.swirl(img, rotation=0,
 ↪  strength=strength, radius=radius, center=(c_x,
 ↪  c_y))

return img, params
```

Figure 11: Noise Injection

C.4. Noise Injection

In this defense, random noise is applied to each image. There is a 50% probability that the noise will be applied to all channels, and a 50% probability that different noise values will be added to each channel independently. The type of noise is chosen uniformly from six varieties implemented in scikit-image.

```
params = []

# average of color channels, different contribution
 ↪  for each channel
options = ['gaussian', 'poisson', 'salt', 'pepper',
 ↪  's&p', 'speckle']

noise_type = np.random.choice(options, 1)[0]
params.append(options.index(noise_type)/6.0)

per_channel = np.random.choice(2) == 0
params.append( per_channel )

if per_channel:
    for i in range(3):
        img[:,:,i] = skimage.util.random_noise(
         ↪  img[:,:,i], mode=noise_type )
else:
    img = skimage.util.random_noise( img,
     ↪  mode=noise_type )

return img, params
```

C.5. FFT Perturbation

```
r, c, _ = img.shape

#Everyone gets the same factor to avoid too many weird
 ↪  artifacts
point_factor = (1.02-0.98)*np.random.random((r,c)) +
 ↪  0.98

randomized_mask = [np.random.choice(2)==0 for x in
 ↪  range(3)]
keep_fraction = [(0.95-0.0)*np.random.random(1)[0] +
 ↪  0.0 for x in range(3)]

params = randomized_mask + keep_fraction

for i in range(3):
    im_fft = fftpack.fft2(img[:,:,i])

    # Set r and c to be the number of rows and columns
     ↪  of the array.
    r, c = im_fft.shape
```

Figure 12: FFT Perturbation

Figure 13: Random Zoom

```
    if randomized_mask[i]:
        mask = np.ones(im_fft.shape[:2]) > 0
        im_fft[int(r*keep_fraction[i]):
         ↪  int(r*(1-keep_fraction[i]))] = 0
        im_fft[:, int(c*keep_fraction[i]):
         ↪  int(c*(1-keep_fraction[i]))] = 0

        mask = ~mask
        #Now things to keep = 0, things to remove = 1
        mask = mask * ~(np.random.uniform(
         ↪  size=im_fft.shape[:2] ) <
         ↪  keep_fraction[i])
        #Now switch back
        mask = ~mask

        im_fft = np.multiply(im_fft, mask)
    else:
        im_fft[int(r*keep_fraction[i]):
         ↪  int(r*(1-keep_fraction[i]))] = 0
        im_fft[:, int(c*keep_fraction[i]):
         ↪  int(c*(1-keep_fraction[i]))] = 0

    #Now, lets perturb all the rest of the non-zero
     ↪  values by a relative factor
    im_fft = np.multiply(im_fft, point_factor)
    im_new = fftpack.ifft2(im_fft).real

    #FFT inverse may no longer produce exact same
     ↪  range, so clip it back
    im_new = np.clip(im_new, 0, 1)

    img[:,:,i] = im_new

return img, params
```

C.6. Zoom Group

C.6.1. Random Zoom

Guo, Rana, Cissé, *et al.* [13] considered cropping and rescaling of an image as one of their defenses, which is effectively zooming in on a portion of the image, an approach that was defeated by Athalye, Engstrom, Ilyas, *et al.* [11]. We reuse this as one of our defenses, where the distance from each edge of the image is cropped by $\mathcal{U}[10, 50]$, independently for each edge.

```
h, w, _ = img.shape

i_s = np.random.random_integers(10, 50)
i_e = np.random.random_integers(10, 50)
j_s = np.random.random_integers(10, 50)
j_e = np.random.random_integers(10, 50)

params = [i_s/50, i_e/50, j_s/50, j_e/50]

i_e = h-i_e
j_e = w-j_e

#Crop the image...
img = img[i_s:i_e,j_s:j_e,:]
#...now scale it back up
img = skimage.transform.resize(img, (h, w, 3))

return img, params
```

C.6.2. Seam Carving Expansion

Seam Carving [29] is an approach to find irregular but contiguous paths of pixels through an image, such that the pixels along the path can be removed while avoiding perturbation of the main image content. This allows for a type of fast content aware image zooming, which we use as another defense.

We randomly select some number of pixels $x, y \sim \mathcal{U}[10, 50]$ to remove from the image horizontally or vertically. With a 50% chance, we will only remove pixels from one axis of the image instead of both. Once the pixels are removed, we re-scale the image back to its original height and width.

```
h, w, _ = img.shape

both_axis = np.random.choice(2) == 0
toRemove_1 = np.random.random_integers(10, 50)
toRemove_2 = np.random.random_integers(10, 50)

params = [both_axis, toRemove_1/50, toRemove_2/50]

if both_axis:
    #First remove from vertical
    eimg = skimage.filters.sobel(
     ↪  skimage.color.rgb2gray(img) )
    img = skimage.transform.seam_carve(img, eimg,
     ↪  'vertical', toRemove_1)
    #Now from horizontal
    eimg = skimage.filters.sobel(
     ↪  skimage.color.rgb2gray(img) )
    img = skimage.transform.seam_carve(img, eimg,
     ↪  'horizontal', toRemove_2)
else:
    eimg = skimage.filters.sobel(
     ↪  skimage.color.rgb2gray(img) )
    direction = 'horizontal'
```

Figure 14: Seam Carving Expansion

Figure 15: Alter HSV

Figure 16: Alter XYZ

Figure 17: Alter LAB

```
    if toRemove_2 < 30:
        direction = 'vertical'
    img = skimage.transform.seam_carve(img, eimg,
    ↪  direction, toRemove_1)
#Now scale it back up
img = skimage.transform.resize(img, (h, w, 3))

return img, params
```

C.7. Color Space Group

C.7.1. Alter HSV

Hue is modified by a value $h \sim \mathcal{U}[-0.05, 0.05]$ and both the Saturation and Value channels are modified by a random value sampled from $s, v \sim \mathcal{U}[-0.25, 0.25]$.

```
img = color.rgb2hsv(img)

params = []

#Hue
img[:,:,0] += randUnifC(-0.05, 0.05, params=params)
#Saturation
img[:,:,1] += randUnifC(-0.25, 0.25, params=params)
#Value
img[:,:,2] += randUnifC(-0.25, 0.25, params=params)

img = np.clip(img, 0, 1.0)
img = color.hsv2rgb(img)
img = np.clip(img, 0, 1.0)

return img, params
```

C.7.2. Alter XYZ

With this transformation, the image is converted to the CIE 1931 XYZ colorspace, perturbed, and then converted back to RGB. All three color channels will be modified by a different random value, sampled as $x, y, z \sim \mathcal{U}[-0.25, 0.25]$.

```
img = color.rgb2xyz(img)

params = []
#X
img[:,:,0] += randUnifC(-0.05, 0.05, params=params)
#Y
img[:,:,1] += randUnifC(-0.05, 0.05, params=params)
#Z
img[:,:,2] += randUnifC(-0.05, 0.05, params=params)

img = np.clip(img, 0, 1.0)
img = color.xyz2rgb(img)
img = np.clip(img, 0, 1.0)

return img, params
```

C.7.3. Alter LAB

With this transformation, the image is converted to the CIELAB colorspace, perturbed, and then converted back to RGB. The L^* channel is modified by a value $l \sim \mathcal{U}[-5, 5]$, and both the a^* and b^* channels are modified by a random value sampled from $a, b \sim \mathcal{U}[-2, 2]$.

```
img = color.rgb2lab(img)

params = []
#L
img[:,:,0] += randUnifC(-5.0, 5.0, params=params)
#a
img[:,:,1] += randUnifC(-2.0, 2.0, params=params)
#b
img[:,:,2] += randUnifC(-2.0, 2.0, params=params)

# L ∈ [0,100] so clip it; a & b channels can have
↪  negative values.
img[:,:,0] = np.clip(img[:,:,0], 0, 100.0)

img = color.lab2rgb(img)
img = np.clip(img, 0, 1.0)

return img, params
```

Figure 18: Alter YUV

Figure 19: Histogram Equalization

Figure 20: Adaptive Histogram Equalization

C.7.4. Alter YUV

Both the U and V channels are modified by a random value sampled from $u, v \sim \mathcal{U}[-0.02, 0.02]$, and Y is modified by a value $y \sim \mathcal{U}[-0.05, 0.05]$.

```
img = color.rgb2yuv(img)

params = []
#Y
img[:,:,0] += randUnifC(-0.05, 0.05, params=params)
#U
img[:,:,1] += randUnifC(-0.02, 0.02, params=params)
#V
img[:,:,2] += randUnifC(-0.02, 0.02, params=params)

# U & V channels can have negative values; clip only Y
img[:,:,0] = np.clip(img[:,:,0], 0, 1.0)

img = color.yuv2rgb(img)
img = np.clip(img, 0, 1.0)

return img, params
```

C.8. Contrast Group

C.8.1. Histogram Equalization

The first transformation in this group performs the simplest of histogram equalizations, applied separately over each channel. All channels use the same number of bins for the histogram, which is chosen from $bins \sim \mathcal{U}[40, 256]$.

```
nbins = np.random.random_integers(40, 256)

params = [ nbins/256.0 ]

for i in range(3):
    img[:,:,i] = skimage.exposure.equalize_hist(
     ↪  img[:,:,i], nbins=nbins)

return img, params
```

C.8.2. Adaptive Histogram Equalization

For the adaptive case we use the Contrast Limited Adaptive Histogram Equalization (CLAHE) algorithm [30]. With a 50% probability, the adaptive histogram equalization is applied on either the whole image or on a channel-by-channel basis. For every application of the process, the kernel width and heights are selected as $k_w, k_h \sim \mathcal{U}[22, 37]$. A clip limit parameter is chosen as $c \sim \mathcal{U}[0.01, 0.04]$.

```
min_size = min(img.shape[0], img.shape[1])/10
max_size = min(img.shape[0], img.shape[1])/6
per_channel = np.random.choice(2) == 0

params = [ per_channel ]

kernel_h = [ randUnifI(min_size, max_size,
 ↪  params=params) for x in range(3)]
kernel_w = [ randUnifI(min_size, max_size,
 ↪  params=params) for x in range(3)]

clip_lim = [randUnifC(0.01, 0.04, params=params)  for
 ↪  x in range(3)]

if per_channel:
    for i in range(3):
        kern = (kernel_w[i], kernel_h[i])
        img[:,:,i] =
         ↪  skimage.exposure.equalize_adapthist(
         ↪  img[:,:,i], kernel_size=kern,
         ↪  clip_limit=clip_lim[i])
else:
    kern = (kernel_w[0], kernel_h[0])
    img = skimage.exposure.equalize_adapthist( img,
     ↪  kernel_size=kern, clip_limit=clip_lim[0])

return img, params
```

C.8.3. Contrast Stretching

The last approach we consider in this group performs a simple re-scaling of all values within a channel to "stretch" to a specified minimum and maximum value. With a 50% probability this will be done on the whole image at once with a single value range, or on a channel-by-channel basis with a different min and max value for each channel. The minimum will be selected from $min \sim \mathcal{U}[0.01, 0.04]$, and the maximum from $max \sim \mathcal{U}[0.96, 0.99]$.

Figure 21: Contrast Stretching

```
per_channel = np.random.choice(2) == 0

params = [ per_channel ]

low_precentile = [ randUnifC(0.01, 0.04, params=params)
 ↪  for x in range(3)]
hi_precentile = [ randUnifC(0.96, 0.99, params=params)
 ↪  for x in range(3)]
if per_channel:
    for i in range(3):
        p2, p98 = np.percentile(img[:,:,i],
         ↪  (low_precentile[i]*100,
         ↪  hi_precentile[i]*100))
        img[:,:,i] =
         ↪  skimage.exposure.rescale_intensity(
         ↪  img[:,:,i], in_range=(p2, p98))
else:
    p2, p98 = np.percentile(img, (low_precentile[0] *
     ↪  100, hi_precentile[0]*100))
    img = skimage.exposure.rescale_intensity( img,
     ↪  in_range=(p2, p98) )

return img, params
```

C.8.4. Grey Scale Mix

Each channel is given a random weight sampled as $w_r, w_g, w_b \sim \mathcal{U}[0,1]$, and then all channels are set to the same weighted average of the channels (i.e., $I_{\text{grey}} = (w_r \cdot R + w_g \cdot G + w_b \cdot B)/(w_r + w_g + w_b)$. Then each channel is set to this value ($R' = G' = B' = I_{\text{grey}}$) to create a grey scale image. This is an alteration of a crude form of grey scale transformation where each channel contributes equally.

```
# average of color channels, different contribution
 ↪  for each channel
ratios = np.random.rand(3)
ratios /= ratios.sum()

params = [x for x in ratios]

img_g = img[:,:,0] * ratios[0] + img[:,:,1] *
 ↪  ratios[1] + img[:,:,2] * ratios[2]

for i in range(3):
    img[:,:,i] = img_g

return img, params
```

C.8.5. Grey Scale Partial Mix

This is the same as the *Grey Scale Mix* transform, in that we first compute a random weighted average grey scale image I_{grey}. Then, instead of simply setting each channel to the new grey value, we randomly interpolate between the original channel's value and the grey scale target. So we sample $p_r, p_g, p_b \sim \mathcal{U}[0,1]$ and set each channel as the interpolated value (e.g., $R' = p_r \cdot R + (1 - p_r) \cdot I_{\text{grey}}$).

Figure 22: Grey Scale Mix

Figure 23: Grey Scale Partial Mix

```
ratios = np.random.rand(3)
ratios/=ratios.sum()

prop_ratios = np.random.rand(3)

params = [x for x in ratios] + [x for x in
 ↪  prop_ratios]

img_g = img[:,:,0] * ratios[0] + img[:,:,1] *
 ↪  ratios[1] + img[:,:,2] * ratios[2]

for i in range(3):
    p = max(prop_ratios[i], 0.2)
    img[:,:,i] = img[:,:,i]*p + img_g*(1.0-p)

return img, params
```

C.8.6. 2/3 Grey Scale Mix

This technique is also similar to *Grey Scale Mix*, but here we randomly select one of the three channels to exclude. The remaining two channels will have a randomly weighted average computed, and the same two channels will be set to this new grey image. The randomly selected channel will be left as is, so only two of three channels will have been altered.

```
params = []

# Pick a channel that will be left alone and remove it
 ↪  from the ones to be averaged
channels = [0, 1, 2]
remove_channel = np.random.choice(3)
channels.remove( remove_channel)
```

Figure 24: 2/3 Grey Scale Mix

```
params.append( remove_channel )

ratios = np.random.rand(2)
ratios/=ratios.sum()
params.append(ratios[0]) #They sum to one, so first
 ↪  item fully specifies the group

img_g = img[:,:,channels[0]] * ratios[0] +
 ↪  img[:,:,channels[1]] * ratios[1]

for i in channels:
    img[:,:,i] = img_g

return img, params
```

C.8.7. One Channel Partial Grey

In this case, we randomly pick one channel to convert to the grey scale value, and leave the other two channels alone. The value used will be a random weighted average from the other two channels (each weight sampled from $\mathcal{U}[0,1]$), and similar to *Grey Scale Partial Mix*, we will interpolate the grey scale value I_{grey} with the randomly selected channel using a ratio chosen from $\mathcal{U}[0.1, 0.9]$.

```
params = []

# Pick a channel that will be altered and remove it
 ↪  from the ones to be averaged
channels = [0, 1, 2]
to_alter = np.random.choice(3)
channels.remove(to_alter)
params.append(to_alter)

ratios = np.random.rand(2)
ratios/=ratios.sum()
params.append(ratios[0]) #They sum to one, so first
 ↪  item fully specifies the group

img_g = img[:,:,channels[0]] * ratios[0] +
 ↪  img[:,:,channels[1]] * ratios[1]

# Lets mix it back in with the original channel
p = (0.9-0.1)*np.random.random(1)[0] + 0.1
params.append( p )

img[:,:,to_alter] = img_g*p + img[:,:,to_alter]
 ↪  *(1.0-p)

return img, params
```

Figure 25: One Channel Partial Grey

Figure 26: Gaussian Blur

C.9. Denoising Group

C.9.1. Gaussian Blur

The first technique in this group is a simple Gaussian blur. To add randomness, each channel will have a different blur strength chosen from $\sigma \sim \mathcal{U}[0.1, 3]$. With a 50% probability, all channels will be set to use the same σ.

```
if randUnifC(0, 1) > 0.5:
    sigma = [randUnifC(0.1, 3)]*3
else:
    sigma = [randUnifC(0.1, 3), randUnifC(0.1, 3),
     ↪  randUnifC(0.1, 3)]
img[:,:,0] = skimage.filters.gaussian(img[:,:,0],
 ↪  sigma=sigma[0])
img[:,:,1] = skimage.filters.gaussian(img[:,:,1],
 ↪  sigma=sigma[1])
img[:,:,2] = skimage.filters.gaussian(img[:,:,2],
 ↪  sigma=sigma[2])

return img, [x/3.0 for x in sigma]
```

C.9.2. Median Filter

Next we use a simple median filter, and follow the same approach as the Guassian blur. The radius of the blur kernel is chosen from $r \sim \mathcal{U}[2, 5]$, with a 50% chance all channels are forced to use the same radius.

```
if randUnifC(0, 1) > 0.5:
    radius = [randUnifI(2, 5)]*3
else:
    radius = [randUnifI(2, 5), randUnifI(2, 5),
     ↪  randUnifI(2, 5)]

# median blur - different sigma for each channel
for i in range(3):
    mask = skimage.morphology.disk(radius[i])
    img[:,:,i] = skimage.filters.rank.median(
     ↪  img[:,:,i], mask) / 255.0
return img, [x/5.0 for x in radius]
```

Figure 27: Median Filter

Figure 28: Mean Filter

Figure 29: Mean Bilateral Filter

Figure 30: Chambolle Denoising

C.9.3. Mean Filter

This is the same as the median filter described above, but using a mean. It was used as an attempted defensive technique by Li and Li [12]. We choose the radius randomly from $r \sim \mathcal{U}[2, 3]$ instead of using a fixed radius.

```
if randUnifC(0, 1) > 0.5:
    radius = [randUnifI(2, 3)]*3
else:
    radius = [randUnifI(2, 3), randUnifI(2, 3),
     ↪  randUnifI(2, 3)]

# mean blur w/ different sigma for each channel
for i in range(3):
    mask = skimage.morphology.disk(radius[i])
    img[:,:,i] = skimage.filters.rank.mean(img[:,:,i],
     ↪  mask)/255.0

return img, [x/3.0 for x in radius]
```

C.9.4. Mean Bilateral Filter

Next we use mean bilateral filtering, which is an edge preserving filter [31]. We apply it on a channel-wise basis. The implementation we use has 3 parameters, including the radius, that we set using values sampled from $\mathcal{U}[5, 20]$.

```
params = []
radius = []
ss = []

for i in range(3):
    radius.append( randUnifI(2, 20, params=params) )
    ss.append( randUnifI(5, 20, params=params) )
    ss.append( randUnifI(5, 20, params=params) )

for i in range(3):
    mask = skimage.morphology.disk(radius[i])
    img[:,:,i] = skimage.filters.rank.mean_bilateral(
     ↪  img[:,:,i], mask, s0=ss[i], s1=ss[3+i])/255.0

return img, params
```

C.9.5. Chambolle Denoising

We apply Chambolle's total variation denoising algorithm [38] as a potential defense. We choose the algorithm's weight parameter from $w \sim \mathcal{U}[0.05, 0.25]$ and with a 50% chance apply it on either the whole image holistically, or on a channel-by-channel basis.

```
params = []

weight = (0.25-0.05)*np.random.random(1)[0] + 0.05
params.append( weight )

multi_channel = np.random.choice(2) == 0
params.append( multi_channel )

img = skimage.restoration.denoise_tv_chambolle( img,
 ↪  weight=weight, multichannel=multi_channel)

return img, params
```

C.9.6. Wavelet Denoising

We apply wavelet denoising [32] using the Daubechies 1 wavelet as another defense. Wavelets where used by Prakash, Moran, Garber, *et al.* [15] but easily defeated with BPDA [17]. For the randomized parameters, we use a 50% chance to first convert to the YCbCr color space first (the scikit-image documentation recommends this to improve results), a 50:50 chance to select between soft and hard thresholding of the filter, and a 50:50 chance to use either 0 or 1 levels of the wavelets.

```
convert2ycbcr = np.random.choice(2) == 0
wavelet = np.random.choice(self.wavelets)
mode_ = np.random.choice(["soft", "hard"])
denoise_kwargs = dict(multichannel=True,
 ↪  convert2ycbcr=convert2ycbcr, wavelet=wavelet,
 ↪  mode=mode_)
```

Figure 31: Wavelet Denoising

Figure 32: Non-Local Means Denoising

```
max_shifts = np.random.choice([0, 1])

params = [convert2ycbcr, self.wavelets.index(wavelet)/
 ↪  float(len(self.wavelets)), max_shifts/5.0,
 ↪  (mode_=="soft")]

img = skimage.restoration.cycle_spin(img,
 ↪  func=skimage.restoration.denoise_wavelet,
 ↪  max_shifts=max_shifts, func_kw=denoise_kwargs,
 ↪  multichannel=True, num_workers=1)

return img, params
```

Initially we wanted to use a wider spectrum of possible wavelets and levels for this filter, but found them to be too computationally demanding.

C.9.7. Non-Local Means Denoising

The last denoising approach we apply is a fast Non-Local Means denoising [39], which is edge preserving and beneficial when repeated patterns are present. This same transform was used by Xu, Evans, and Qi [27]. With a 50% chance we will apply this denoising to either the image holistically, or on a channel-by-channel basis. The patch-size parameter is chosen from $\mathcal{U}[5, 7]$, and the patch-distance from $\mathcal{U}[6, 11]$. A third random value is used to perturb the estimation of the variance σ, and is better understood through the code in the appendix.

```
h_1 = randUnifC(0, 1)

params = [h_1]

sigma_est = np.mean(
 ↪  skimage.restoration.estimate_sigma(img,
 ↪  multichannel=True) )
h = (1.15-0.6)*sigma_est*h_1 + 0.6*sigma_est
#If false, it assumes some weird 3D stuff
multi_channel = np.random.choice(2) == 0
params.append( multi_channel )
#Takes too long to run without fast mode.
fast_mode = True
patch_size = np.random.random_integers(5, 7)
params.append(patch_size)
patch_distance = np.random.random_integers(6, 11)
params.append(patch_distance)

if multi_channel:
    img = skimage.restoration.denoise_nl_means( img,
     ↪  h=h, patch_size=patch_size,
     ↪  patch_distance=patch_distance,
     ↪  fast_mode=fast_mode )
else:
    for i in range(3):
        sigma_est = np.mean(
         ↪  skimage.restoration.estimate_sigma(
         ↪  img[:,:,i],  multichannel=True ) )
        h = (1.15-0.6)*sigma_est*params[i] +
         ↪  0.6*sigma_est
        img[:,:,i] =
         ↪  skimage.restoration.denoise_nl_means(
         ↪  img[:,:,i], h=h, patch_size=patch_size,
         ↪  patch_distance=patch_distance,
         ↪  fast_mode=fast_mode )

return img, params
```

D. FGSM Figures

We found the FGSM attack to be significantly less effective than PGD and so concentrated our efforts on the stronger PGD attack. For completeness, we include the results from the FGSM experiments here in Figure 33.

Note that in Figure 33b, Adversarial Training outperforms BaRT, but Kurakin, Goodfellow, and Bengio specifically trained their models to withstand the FGSM attack. They report that providing a similar defense against PGD was nearly computationally intractable and provided no benefits over training with FGSM.

E. BaRT Ensembles

BaRT is premised on the amalgamation of multiple weak defenses into one stronger system. We can further extend BaRT as a "self-ensembling" technique, by averaging the predictions of BaRT over multiple realizations of $t(\cdot)$. Because $t(\cdot)$ represents the randomized process of selecting multiple transformations with random parameterizations, each invocation of $t(\cdot)$ will produce a different image, and thus our model will produce a different result. In this way the BaRT technique can be used as an ensemble approach without any additional work, as the intrinsic nature of $t(\cdot)$ provides the diversity of outputs that is a necessary condition for an ensemble to improve on the performance of its members [40].

To attack our ensembled BaRT defense, we note that since we are only averaging the results of the same model $f(\cdot)$, no changes are necessary for running our

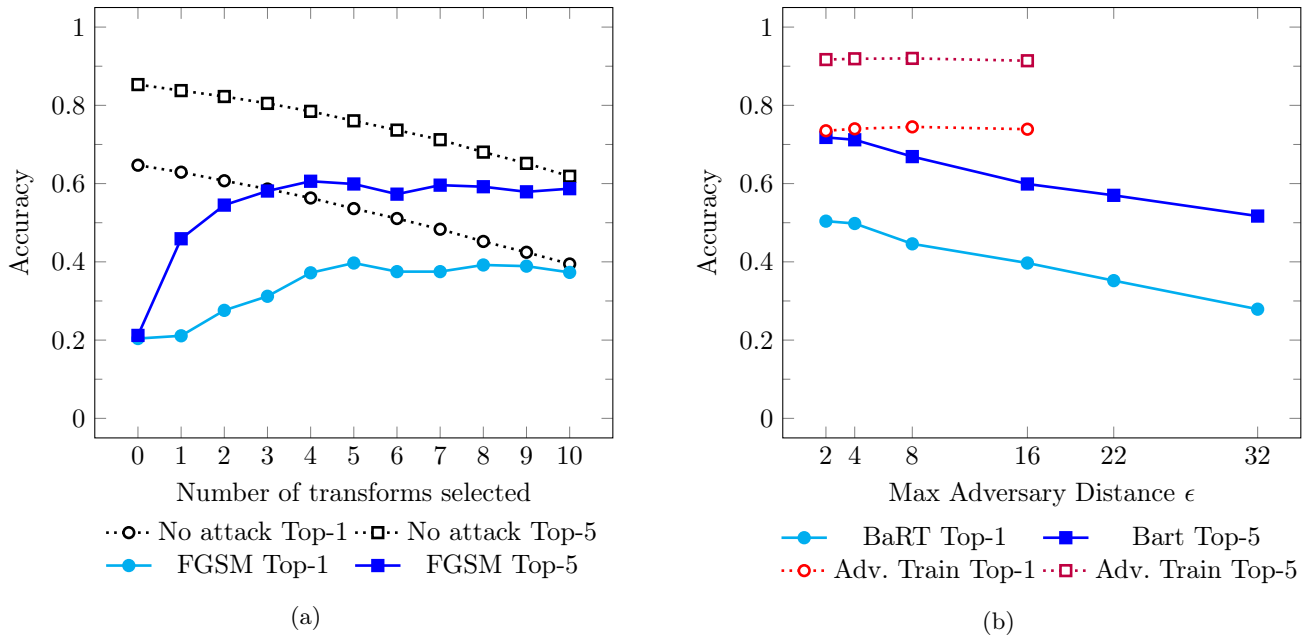

Figure 33: Results of FGSM attacks. (a) Accuracy of BaRT for a varying number of transforms, when not under attack and when being attacked by FGSM. (b) Accuracy of BaRT and the an Adversarially Trained model when under attack by FGSM for varying adversarial distances.

attack and obtaining correct gradient estimates. However, considering an ensemble of size Q, it would not be fair to leave the PGD attack iteration z_{PGD} the same in evaluation. For this reason, for an ensemble of size Q we will use $Q \cdot z_{\text{PGD}}$ attack iterations. We leave the number of EoT iterations $z_{\text{EoT}} = 10$, and will only consider an maximal adversarial attack distance of $\epsilon = 8$. While we would prefer to vary z_{EoT} and ϵ as well, we do not have the computational resources to run all of these experiments. We choose z_{PGD} as the parameter to increase because it allows us to simultaneously perform a larger attack against the standard BaRT model (without any ensembling), which is an indepdently valuable experiment to determine the robustness of the BaRT method. (See Appendix F.)

The results on un-targeted attacks can be seen in Figure 34a. For all of these experiments, we used $k = 5$ random transformations per ensemble member, 10 EoT steps for the adversary, and a maximum adversarial distance of $\epsilon = 8$.

One of the drawbacks of BaRT is a decrease in accuracy when the model is not being attacked by an adversary. This downside can be completely eliminated by ensembling: both Top-1 and Top-5 accuracy can be improved to the level of the baseline ResNet model that had no prepossessing transforms applied.

We also observe that Top-5 accuracy when under attack by PGD improves significantly (from 55.4% with a single member to 71.4% with a thirteen member ensemble), although the Top-1 accuracy does not show an improvement. We suspect two factors are a play that result in this phenomena. 1) The variance introduced by our transforms $t(\cdot)$ can impede the model's ability to get the correct class as the Top-1 prediction, especially when multiple classes are related and small details become necessary to make distinctions. Because of the correlated classes, we expect the variance to be greatest in the Top-1 and Top-2 predictions, and for the variance to decrease with the prediction rank. 2) The variance reduction obtained by ensembling is of the same order of magnitude as the variance introduced to the Top-1 and Top-2 predictions, causing the effects to "cancel out" and result in the same accuracy. The Top-5 predictions would then have a lower initial variance, which more easily averaged out by voting, resulting in an improved accuracy. This would explain why the accuracy at Top-1 remains relatively stable, but has a more significant improvement in the Top-5 regime.

The effects of ensembling on targeted attacks are shown in Figure 34b. The attack parameters were the same as those used on the un-targeted attacks above. The FGSM attack was too weak to draw any conclusions. We see some evidence that ensembling improves robustness against targeted PGD attacks, although there was too much variability in outcome to make definitive conclusions. In order to reduce variation, we ran these

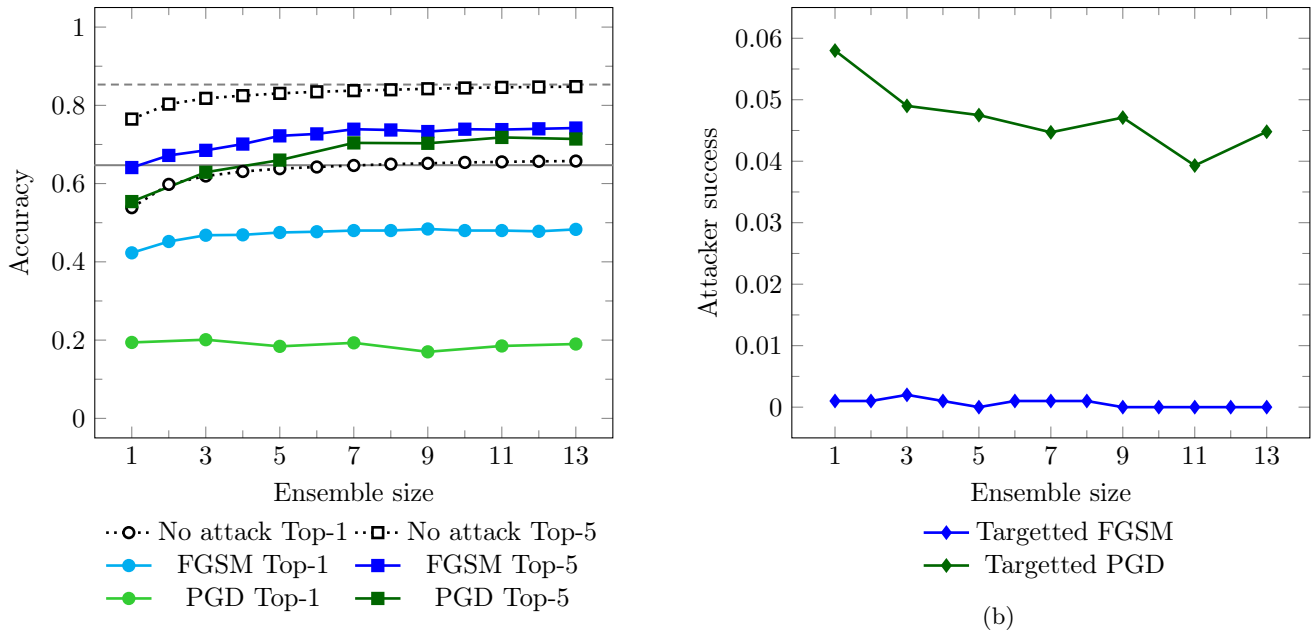

Figure 34: The effect of ensemble size on BaRT performance. In both figures, ensembles were formed by voting after the final softmax activation. (a) Accuracy of the model when varying size of ensemble for un-targeted attacks. The gray horizontal lines represent baseline model accuracy when no transforms or attacks are applied (solid: Top-1 accuracy; dashed: Top-5 accuracy). (b) Success of the adversary of when varying size of ensemble for targeted attacks.

experiments across 2000 images (two from each class) instead of 1000 images as in the targeted experiments reported elsewhere in the paper.

In Figure 35 we report the same results, but with ensembles where the final decision was the result of adding the logits of the members, i.e. before the final softmax activation was performed. This is less reasonable than performing the aggregation after the softmax, but because the base learners are so similar (indeed, they are identical except for choice of random prepossessing steps) their logits are in the same range and can be effectively averaged. We find no significant difference between combining ensemble members before and after softmax activation for untargetted attacks, which accords with the results on ensembling similar base networks reported by Ju, Bibaut, and Laan [41]. Interestingly, the performance against targeted attacks appears considerably better when aggregating logits. (Compare Figure 34b and Figure 35b.) As noted above these results are particularly noisy and since we do not have a hypothesis about why averaging logits would improve defense in this situation, we do not wish to read too much into this improvement.

As a final set of experiments, we consider the trade-off between the "width" and "depth" of a defensive ensemble. "Width" refers to the number of ensemble members, and "depth" to the number of transforms applied to the input of each member. We did this in order to answer the following question: If you had a limited transformation budget, would it be better to apply more of them in series to fewer networks, or to apply fewer transformations in parallel to more networks?

Figure 37 shows the results for total transformation budgets ranging from three to ten. For each one of the the subplots, the left side shows the results of having a single network with n transforms applied in series to the input, while the right side shows the results of an ensemble of n different networks each using only a single transform. In between are intermediate sized ensembles. For example, the bottom left ("Transform Budget 9") shows the effect of having a single network with nine transforms applied in series, three networks with three transforms each, and nine networks with one transform each.

Across the different transform budgets, having a single network with the maximum number of transforms is best if you want to maximize the Top-1 accuracy when under attack, but having a larger ensemble with fewer transforms applied to each member is better for Top-5 accuracy or when the system is not under attack.

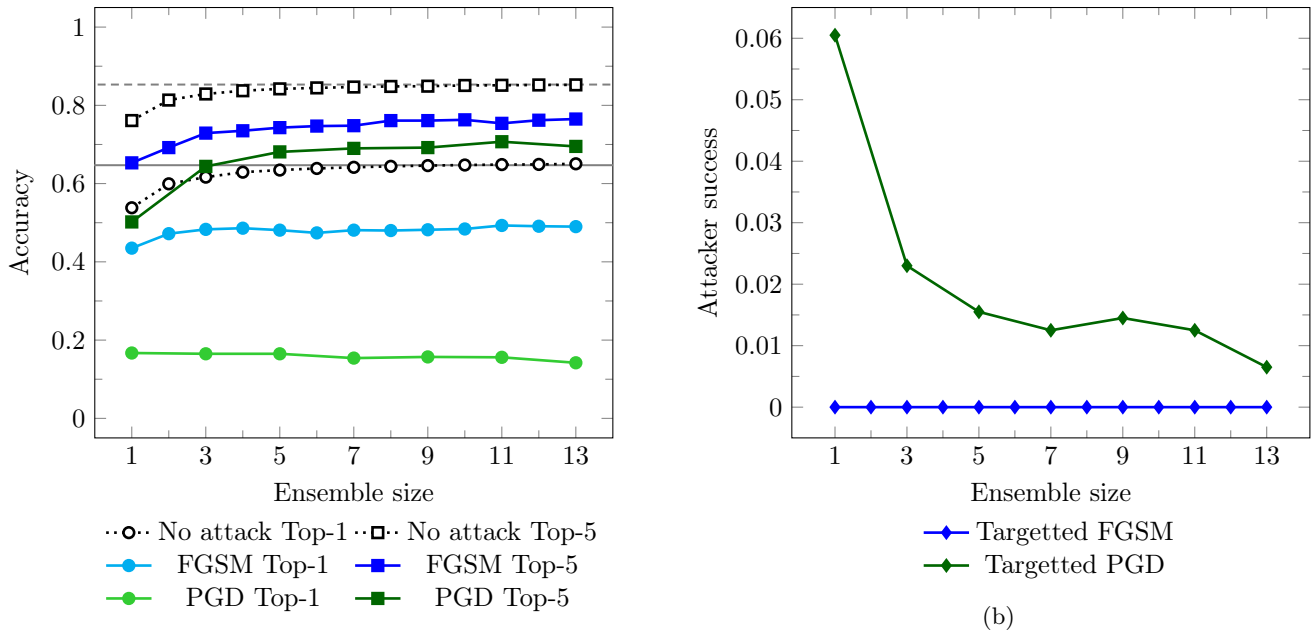

Figure 35: The effect of ensemble size on BaRT performance. In both figures, ensembles were formed by voting based on the member networks' logits, before the final softmax activation. (a) Accuracy of model when varying size of ensemble for un-targeted attackss. The gray horizontal lines represent model accuracy when no transforms or attacks are applied (solid: Top-1 accuracy; dashed: Top-5 accuracy). (b) Success of the adversary of when varying size of ensemble for targeted attacks.

These results indicate that a defensive actor may be able to manipulate width-vs-height as a meta-parameter in order to respond to their particular context. However, more experiments should be run on larger transform budgets before drawing strong conclusions. Applying only one or two transformations before doing inference does not take full advantage of the compounding nature of applying randomized transformations serially. Having only one or two members of an ensemble does not take full advantage of the the ensemble's ability to trade higher variance for lower bias. Of all of the results shown in Figure 37, only one experimental set-up has an ensemble size greater than two and uses more than two transforms per ensemble member: the middle condition of Transform Budget 9, with an ensemble of three networks with three transforms each.

In order to address this limitation, we re-ran the experiments with transform budgets up to 16. (See Figure 38.) This required a slight change to the way BaRT ensembles and the BPA models were constructed, since our total set of transforms was grouped into ten categories. Previously, selections were made without replacement, but this constraint needed to be dropped in order to support using more than ten pre-processing transformations in series on a given input. The results are qualitatively similar to those for transform budgets up to ten, although there is some indication that ensembles of size two and three may be useful for improving Top-1 accuracy against FGSM, and in the Transform Budget = 16 case, against PGD as well.

We also ran experiments comparing width and depth on targeted attacks (Figure 39). The FGSM attack was not strong enough to produce a success rate above 0.2% in any condition. The PGD attack always achieved better success rates as ensemble size increased (i.e. as the number of transformed applied in serial decreased). The results were similar when we tested transform transform budgets up to 16.

F. Increasing PGD Strength

By scaling the strength of the PGD attack as the ensemble size increased we were also able to judge the effect increasing the attack iterations have on a single (non-ensembled) network. As can be seen in Figure 36, increasing the number of attack iterations from 40 to 520 had a negligible effect on accuracy. Top-1 accuracy decreased from 19.40% to 18.30% and Top-5 accuracy decreased from 55.40% to 55.00%. While it is somewhat surprising that increased attack strength does not have a more dramatic impact on accuracy, we feel that the

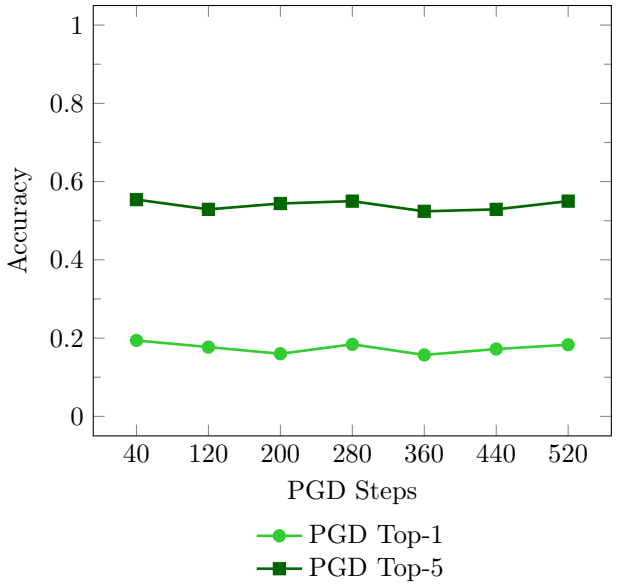

Figure 36: Accuracy of the model when under attack by PGD with a varying number of iterations in the attack.

results of Appendix A and especially Appendix B offer some explanation. To wit, when a sufficient number of transformations are applied to the input image consecutive gradient updates are nearly orthogonal to each other. Because more iterations do not lead the adversary any closer to an image which successfully fools the model, increasing the number of adversarial iterations does not result in a lower accuracy.

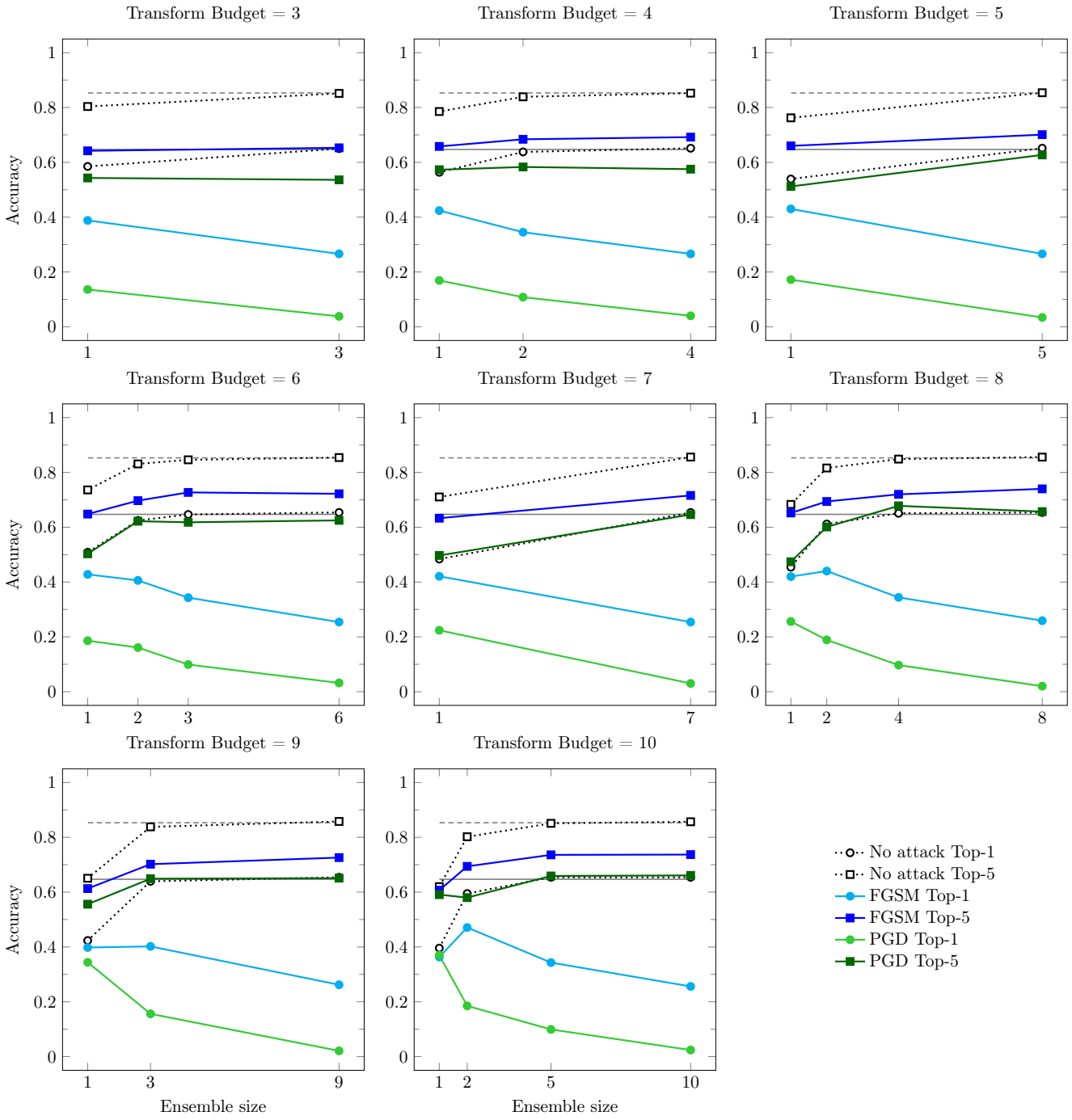

Figure 37: Accuracy of model when trading off between ensembles of many networks with fewer transforms and a single network with more transforms. For each subplot, the total number of transforms available is constant. For example, the bottom left subplot shows the three conditions in which nine transforms can be applied: an "ensemble" of size one with nine transforms applied to its input (on the left of the x-axis), an ensemble of three networks, each with using three transforms (in the middle), or an ensemble of nine networks, each with a single transform (on the right of the x-axis). The number of preprocessing transforms is therefore given by the transform budget divided by the ensemble size. The gray horizontal lines represent model accuracy when no transforms or attacks are applied (solid: Top-1 accuracy; dashed: Top-5 accuracy). Larger transform budgets are shown in Figure 38.

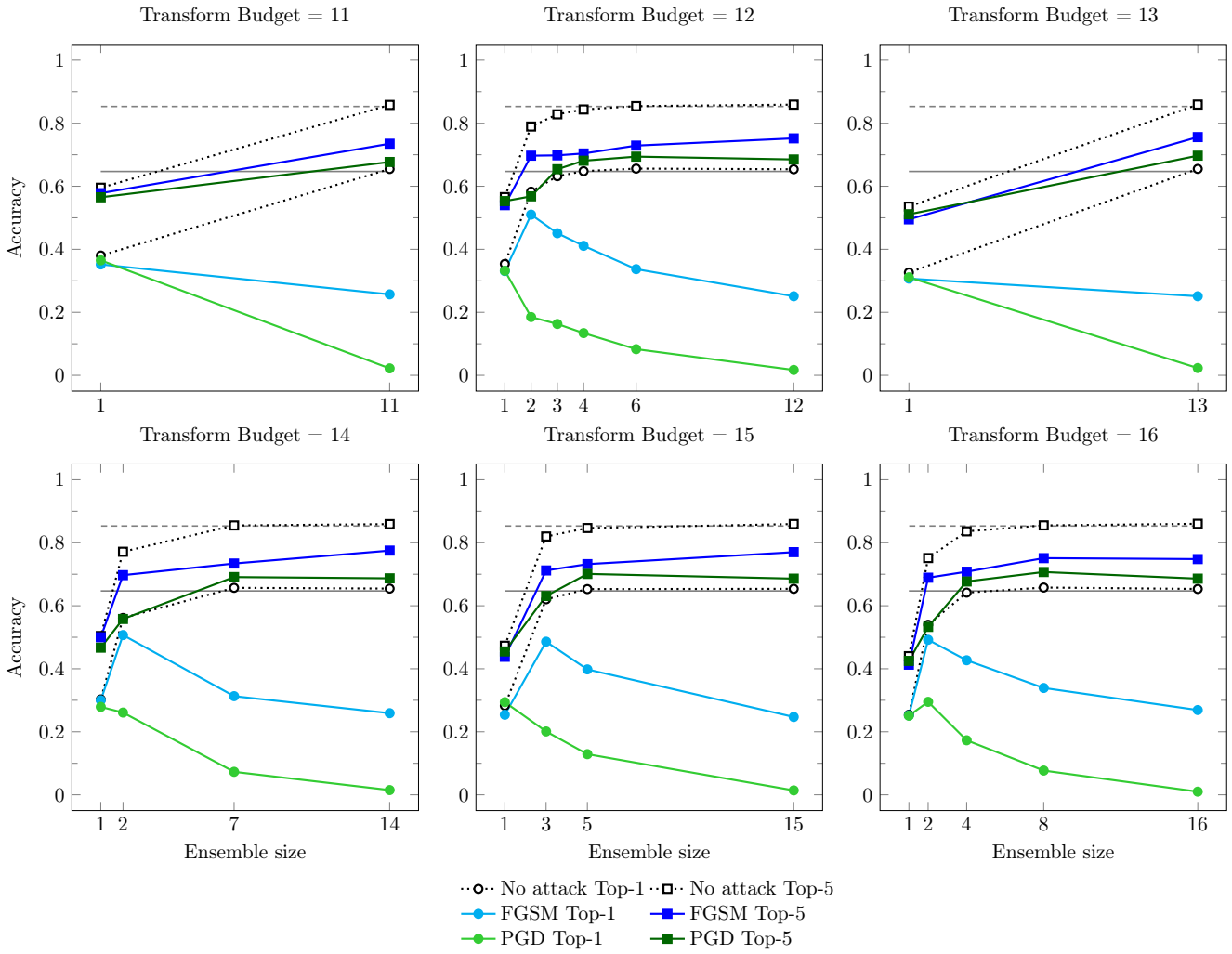

Figure 38: Accuracy of model when trading off between ensembles of many networks with fewer transforms and a single network with more transforms. For each subplot, the total number of transforms available is constant. For example, the bottom left subplot shows the three conditions in which fourteen transforms can be applied: an "ensemble" of size one with fourteen transforms applied to its input (on the left end of the x-axis), an ensemble of two networks, each with using seven transforms (just to the right), an ensemble of seven networks each using two transforms (in the middle of the x-axis), or an ensemble of fourteen networks, each with a single transform (on the right of the x-axis). The number of preprocessing transforms is therefore given by the transform budget divided by the ensemble size. The gray horizontal lines represent model accuracy when no transforms or attacks are applied (solid: Top-1 accuracy; dashed: Top-5 accuracy). Transform budgets between three and ten are shown in Figure 37.

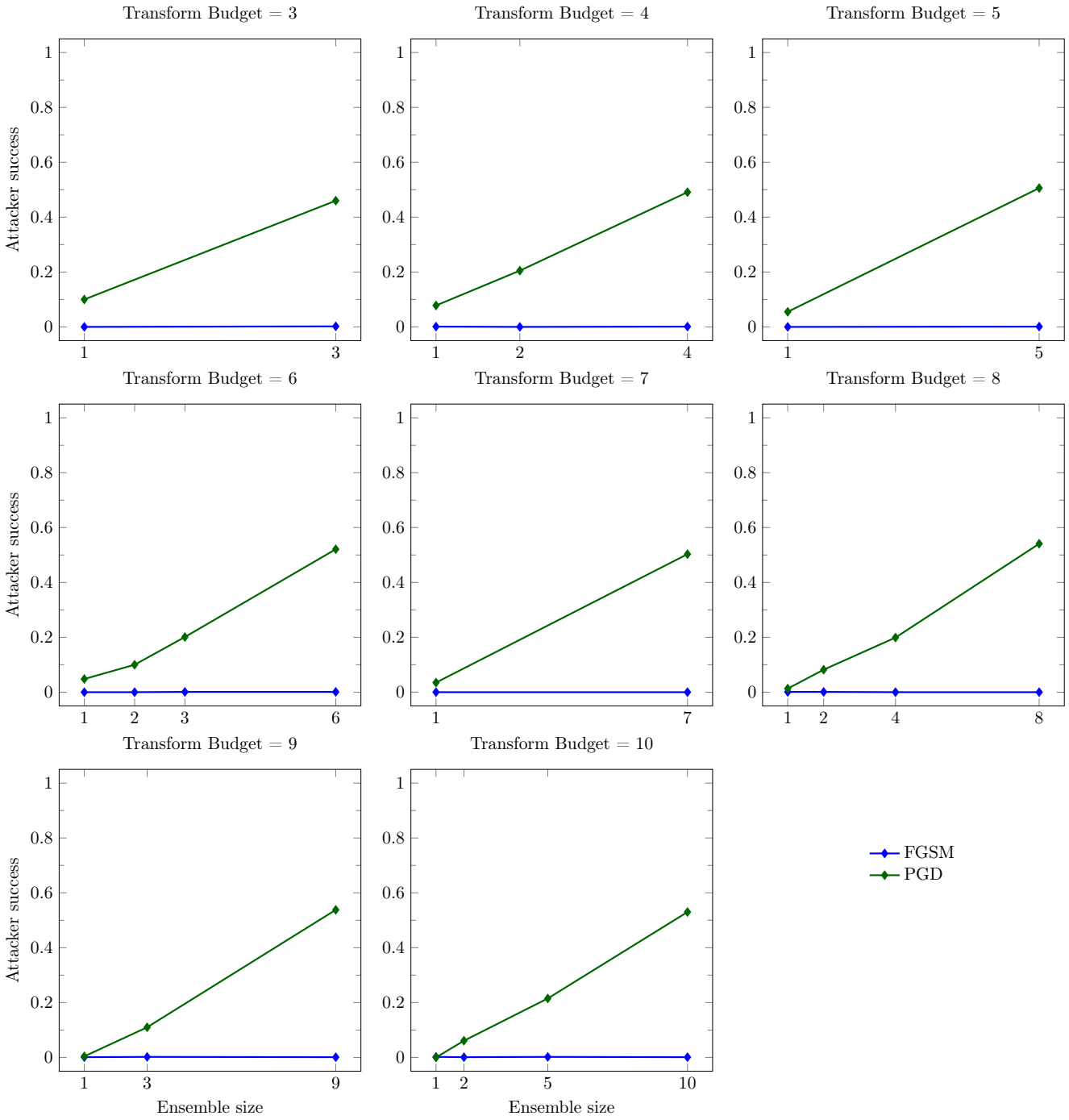

Figure 39: The attacker's success rate when trading off between ensembles of many networks with fewer transforms and a single network with more transforms. For each subplot, the total number of transforms available is constant. For example, the bottom left subplot shows the three conditions in which nine transforms can be applied: an "ensemble" of size one with nine transforms applied to its input (on the left of the x-axis), an ensemble of three networks, each with using three transforms (in the middle), or an ensemble of nine networks, each with a single transform (on the right of the x-axis). The number of preprocessing transforms is therefore given by the transform budget divided by the ensemble size.