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

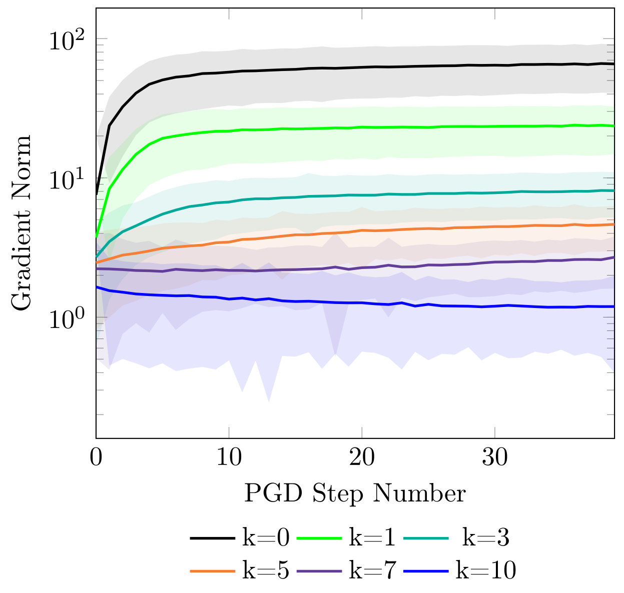


Figure 6: The y-axis (log-scale) shows the L_2 norm, and x-axis which sequential PGD step's gradient is considered. Standard deviation is shown in a lighter shaded region around each plot, for $k = 0, 1, 3, 5, 7, 10$ transforms being used in the defense. 40 EoT steps where used for $k \geq 0$.

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

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

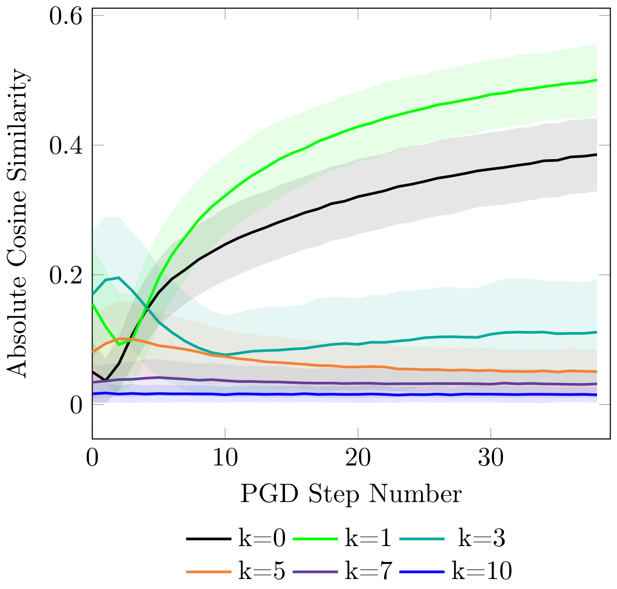


Figure 7: The y-axis shows the absolute cosine similarity, and x-axis which pair of successive PGD steps are being compared. Standard deviation is shown in a lighter shaded region around each plot, for $k = 0, 1, 3, 5, 7, 10$ transforms being used in the defense. 40 EoT steps were used for $k \geq 0$.

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 perturbation 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)
```

Figure 8: Color Precision Reduction

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.

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
```

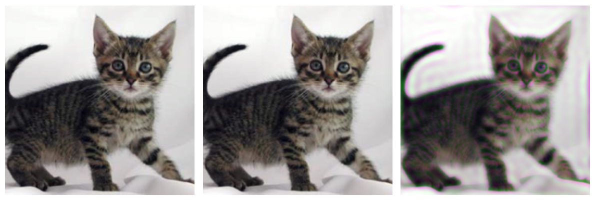


Figure 12: FFT Perturbation

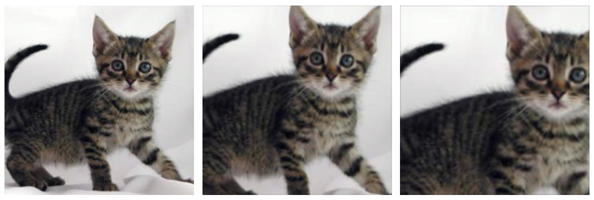


Figure 13: Random Zoom

```
    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'
```

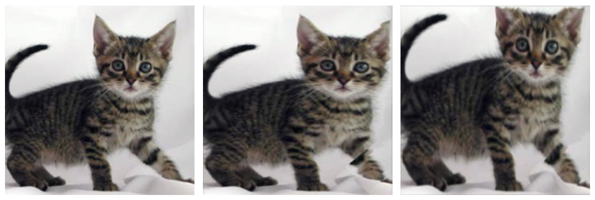


Figure 14: Seam Carving Expansion

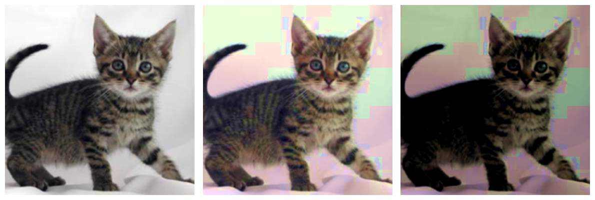


Figure 15: Alter HSV

```
    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 $x, y, z \sim \mathcal{U}[-0.25, 0.25]$.

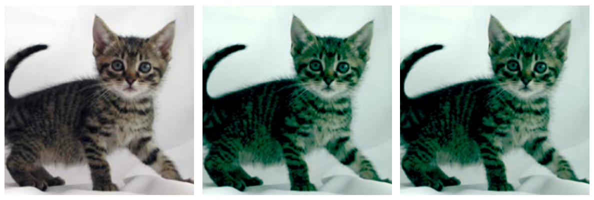


Figure 16: Alter XYZ

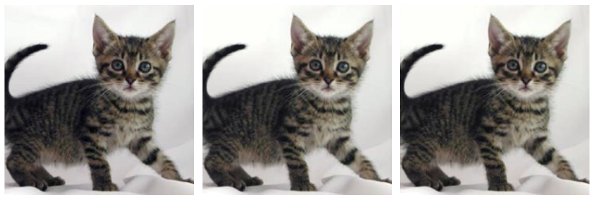


Figure 17: Alter LAB

```
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
```

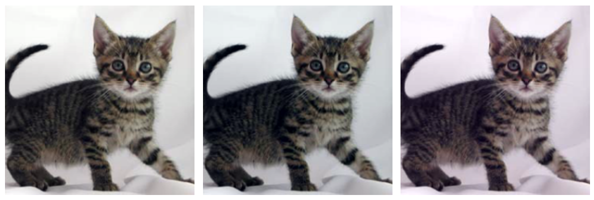


Figure 18: Alter YUV

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

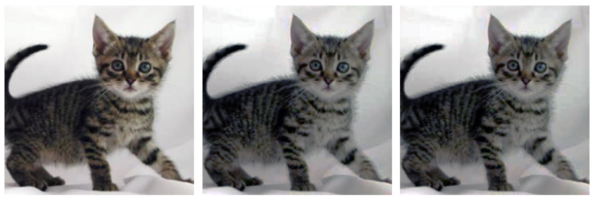


Figure 19: Histogram Equalization

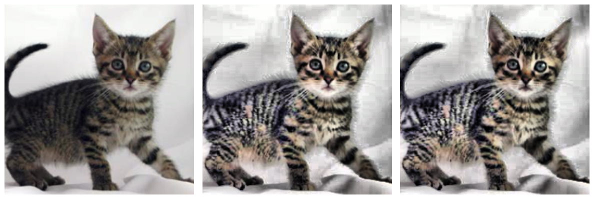


Figure 20: Adaptive Histogram Equalization

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 $max \sim \mathcal{U}[0.96, 0.99]$.

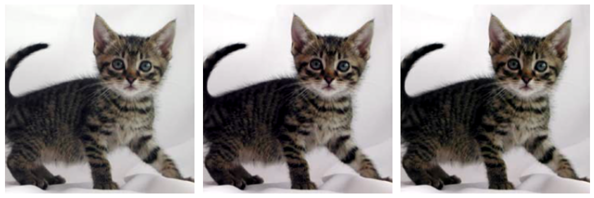


Figure 21: Contrast Stretching

```
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., $R' = p_r \cdot R + (1 - p_r) \cdot I_{\text{grey}}$).

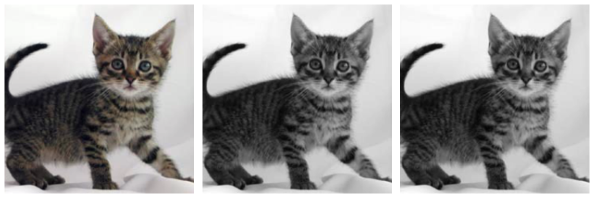


Figure 22: Grey Scale Mix

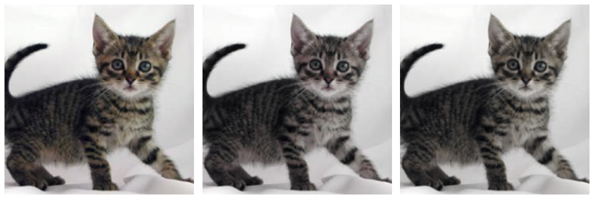


Figure 23: Grey Scale Partial Mix

```
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)
params.append( remove_channel )
```

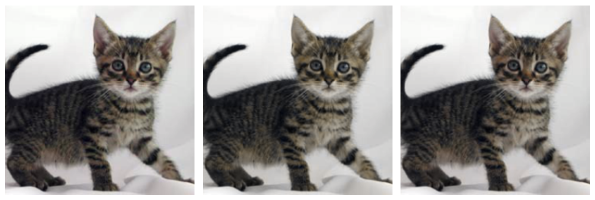


Figure 24: 2/3 Grey Scale Mix

```
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
```

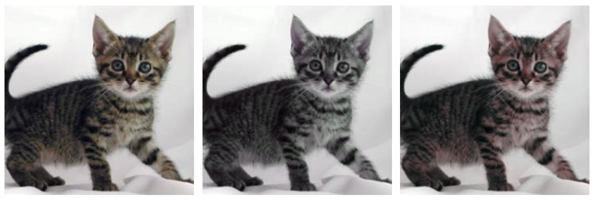


Figure 25: One Channel Partial Grey

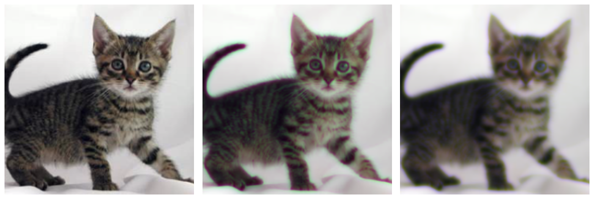


Figure 26: Gaussian Blur

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]
```

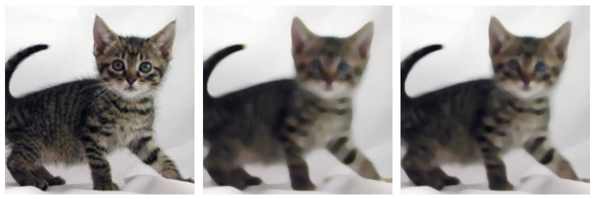


Figure 27: Median Filter

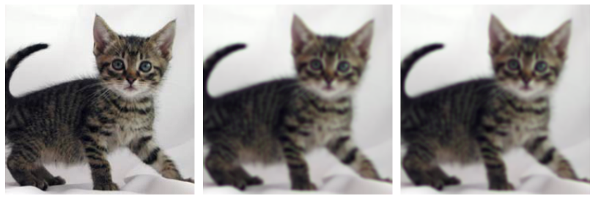


Figure 28: Mean Filter

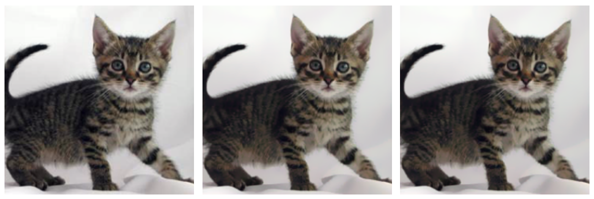


Figure 29: Mean Bilateral Filter

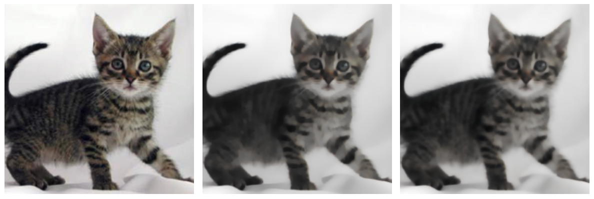


Figure 30: Chambolle Denoising

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_)
```

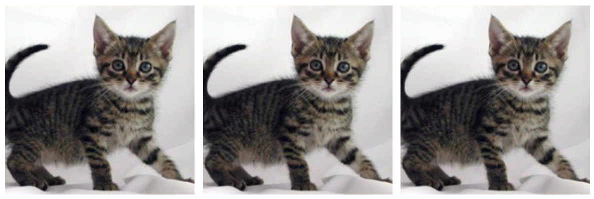


Figure 31: Wavelet Denoising

```
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 )
```

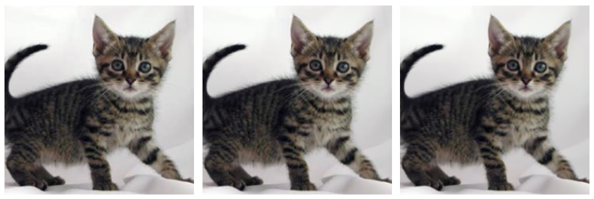


Figure 32: Non-Local Means Denoising

```
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

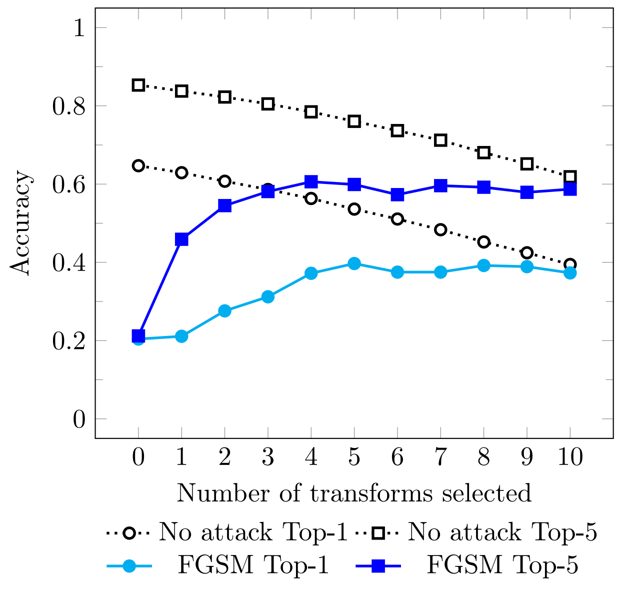
We found the FGSM attack to be significantly less effective than PGD and so concentrated our efforts on the stronger PGD attack. For completeness, we include the results from the FGSM experiments here in Figure 33.

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.

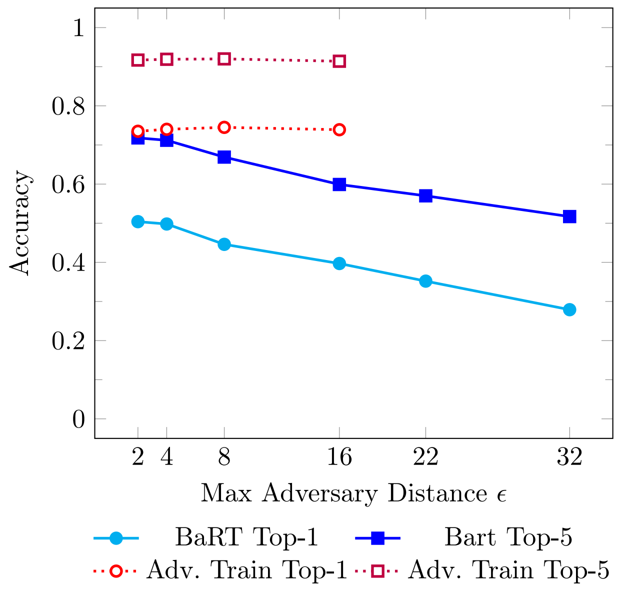
E. BaRT Ensembles

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



(a)



(b)

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.

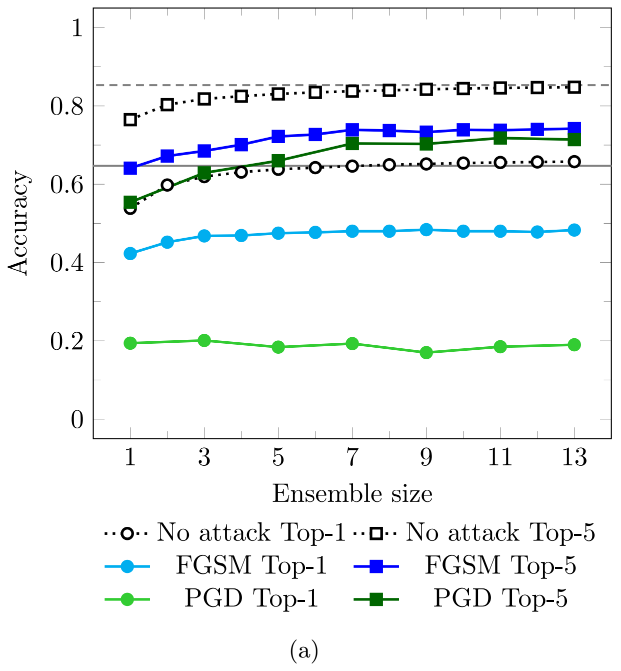
attack and obtaining correct gradient estimates. However, considering an ensemble of size Q, it would not be fair to leave the PGD attack iteration z_{PGD} the same in evaluation. For this reason, for an ensemble of size Q we will use $Q \cdot z_{\text{PGD}}$ attack iterations. We leave the number of 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



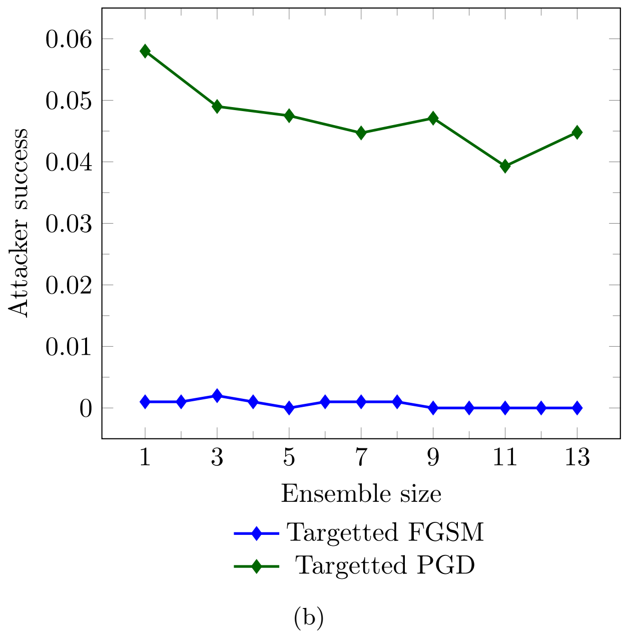


Figure 34: The effect of ensemble size on BaRT performance. In both figures, ensembles were formed by voting after the final softmax activation. (a) Accuracy of the model when varying size of ensemble for un-targeted attacks. The gray horizontal lines represent baseline model accuracy when no transforms 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.

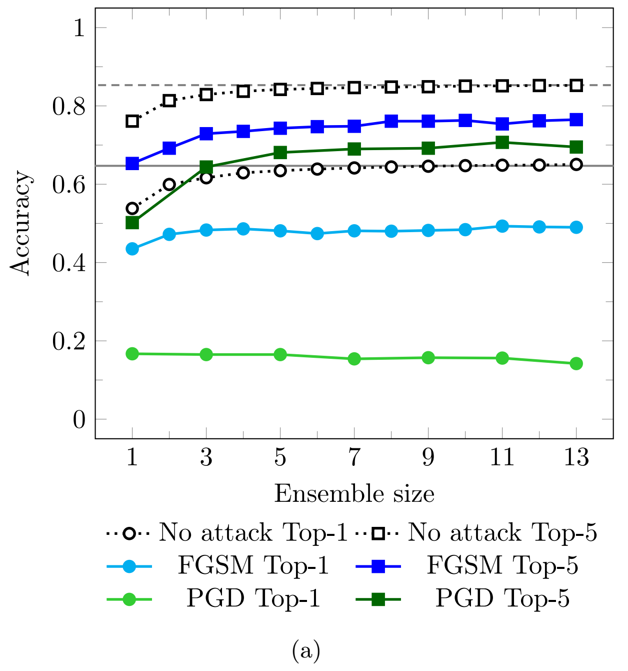
experiments across 2000 images (two from each class) instead of 1000 images as in the targeted experiments reported elsewhere in the paper.

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.



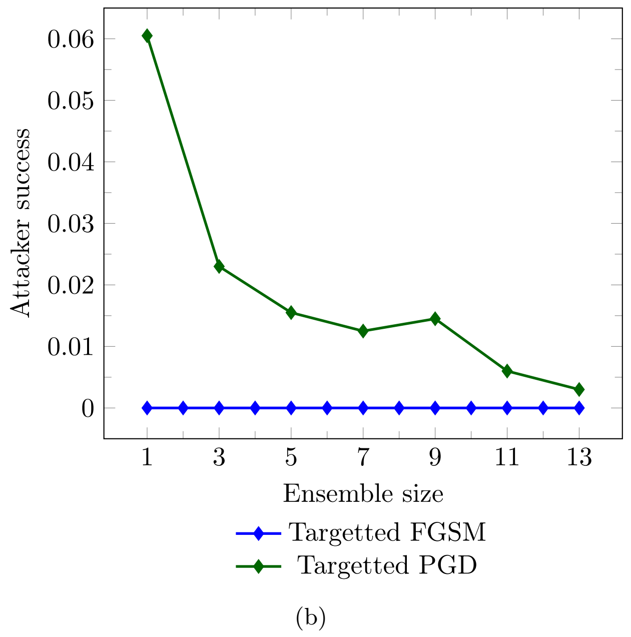


Figure 35: The effect of ensemble size on BaRT performance. In both figures, ensembles were formed by voting based on the member networks' logits, before the final softmax activation. (a) Accuracy of model when varying size of ensemble for un-targeted attackss. The gray horizontal lines represent model 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

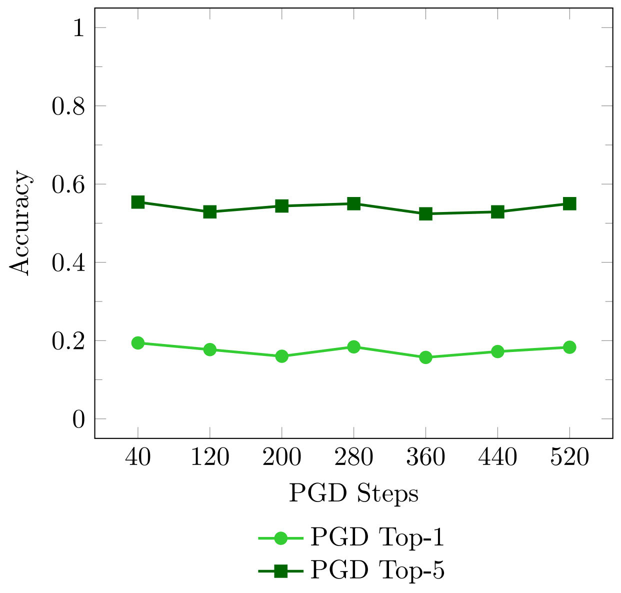


Figure 36: Accuracy of the model when under attack by PGD with a varying number of iterations in the attack.

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.

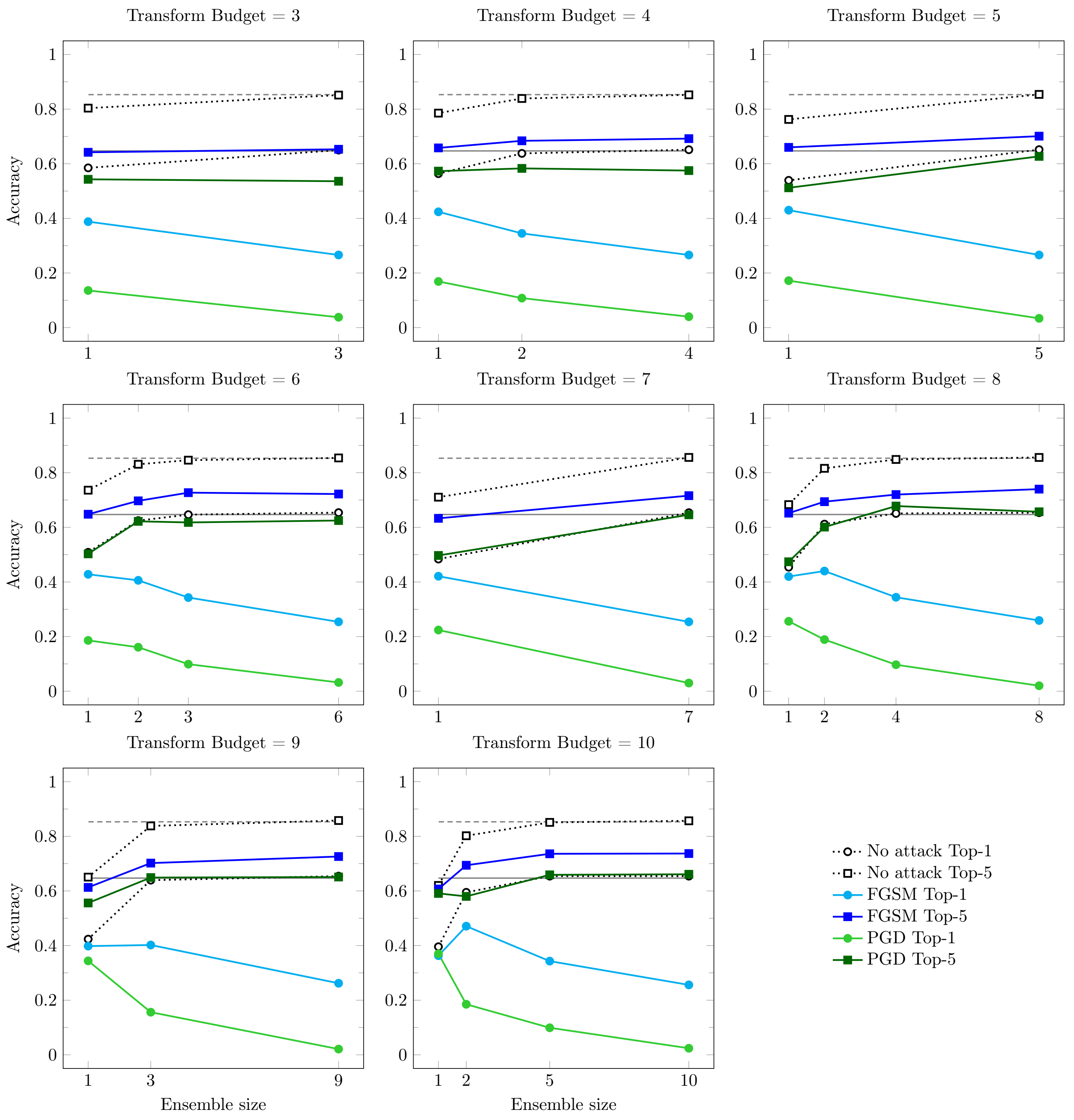


Figure 37: Accuracy of model when trading off between ensembles of many networks with fewer transforms and a single network with more transforms. For each subplot, the total number of transforms available is constant. For example, the bottom left subplot shows the three conditions in which nine transforms 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.

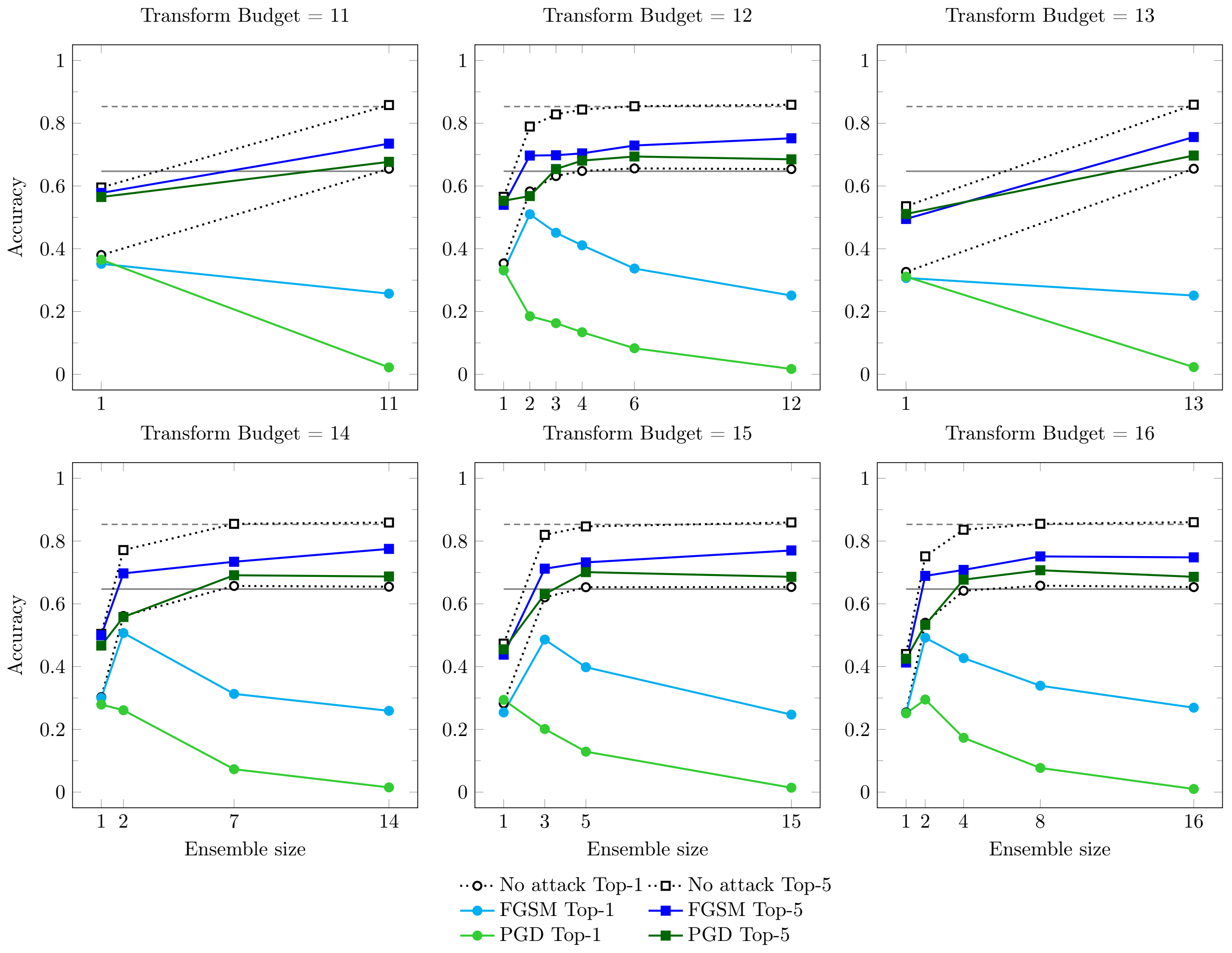


Figure 38: Accuracy of model when trading off between ensembles of many networks with fewer transforms and a single network with more transforms. For each subplot, the total number of transforms available is constant. For example, the bottom left subplot shows the three conditions in which fourteen transforms 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.

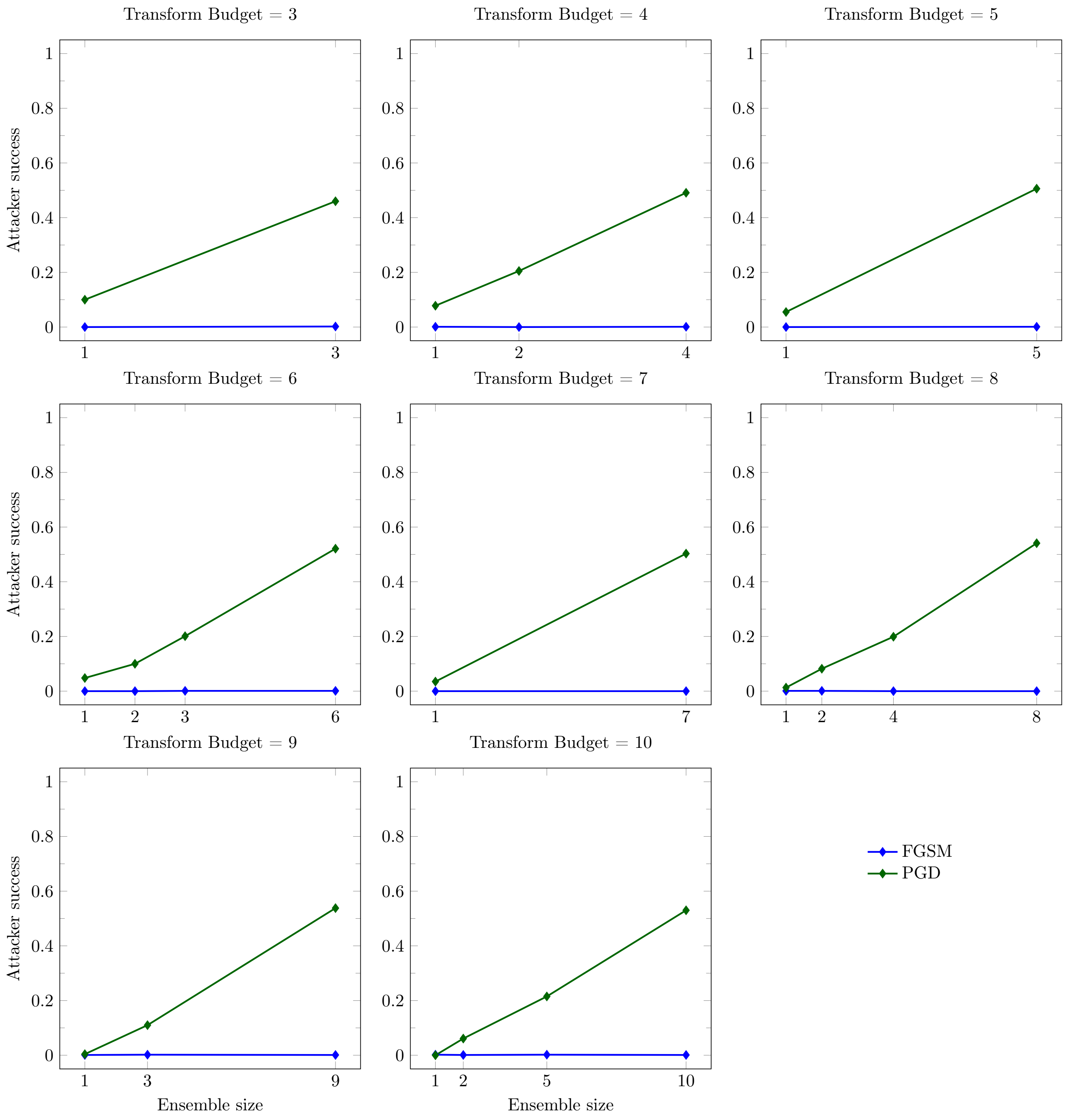


Figure 39: The attacker's success rate when trading off between ensembles of many networks with fewer transforms and a single network with more transforms. For each subplot, the total number of transforms available is constant. For example, the bottom left subplot shows the three conditions in which 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.