Achieving Robustness in the Wild via
Adversarial Mixing with Disentangled Representations
(Supplementary Material)

A. Additional examples

Figure 9 shows additional examples of perturbations obtained on Color-MNIST by (a) mixup, (b) adversarial attacks on $\ell_\infty$-bounded perturbations of size $\epsilon = 0.1$, and (c) our method AdvMix. Figure 10 shows examples on CELEBA. The underlying classifier is the nominally trained convolutional network. We observe that the perturbations generated by AdvMix are semantically meaningful and result in plausible image variants – to the contrary of the other two methods.

Figure 9. Example of perturbations obtained by different techniques on our Color-MNIST dataset. The image on the far left is the original image. On the same row are variations of that image. Even rows show the rescaled difference between the original image and its variants.
Figure 10. Example of perturbations obtained by different techniques on CELEBA. The image on the far left is the original image. On the same row are variations of that image. Even rows show the rescaled difference between the original image and its variants.
Figure 11 shows image variants generated by AdvMix. For four out of five images, AdvMix is able to change the decision of a “smile” detector (nominally trained on CelebA). We can qualitatively observe that brighter skin-tone and rosy cheeks tends to produce images that are more easily classified as “smiling”. Our interpretation is that pictures on the second row appear to be taken using flash photography (where it is more common for people to smile). The second picture from the left (on the second row) also seem to be taken at night during an event.
B. Additional results on CELEBA

For completeness, Table 4 shows the performance of mixup, Cutout and CutMix on the CELEBA attributes used in Table 3. In addition to the evaluation of the unmodified clean test set, we also evaluate all methods by executing Algorithm 2 with \( N_r \) set to 10. In other words, for each trained classifier, we try to find a misclassified variant for each example of the test set. When a misclassified variant is found, we count the corresponding example as misclassified “under perturbation”.

We observe that while data augmentation schemes such as mixup, Cutout or CutMix sometimes improve over nominal training, they do not provide consistent improvements. With the exception of “Attribute 3” (where CutMix is particularly efficient), AdvMix achieves the highest accuracy on the clean test set. Additionally, as is expected, AdvMix systematically achieves the highest accuracy on perturbed images.

Table 4. Clean accuracy and accuracy under perturbations on different classification tasks of the CELEBA dataset.

<table>
<thead>
<tr>
<th>Method</th>
<th>Accuracy on Attribute 1</th>
<th>Accuracy on Attribute 2 (smiling)</th>
<th>Accuracy on Attribute 3</th>
<th>Accuracy on Attribute 4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Clean</td>
<td>Under perturbation</td>
<td>Clean</td>
<td>Under perturbation</td>
</tr>
<tr>
<td>Nominal</td>
<td>96.49%</td>
<td>40.35%</td>
<td>90.22%</td>
<td>18.60%</td>
</tr>
<tr>
<td>mixup (( \alpha = 0.2 ))</td>
<td>97.22%</td>
<td>50.48%</td>
<td>90.95%</td>
<td>30.49%</td>
</tr>
<tr>
<td>Cutout</td>
<td>96.92%</td>
<td>61.58%</td>
<td>90.44%</td>
<td>17.55%</td>
</tr>
<tr>
<td>CutMix</td>
<td>97.18%</td>
<td>32.86%</td>
<td>90.88%</td>
<td>15.46%</td>
</tr>
<tr>
<td>AT ( \ell_\infty ) with ( \epsilon = 4/255 )</td>
<td>95.34%</td>
<td>48.91%</td>
<td>91.11%</td>
<td>60.93%</td>
</tr>
<tr>
<td>AT ( \ell_\infty ) with ( \epsilon = 8/255 )</td>
<td>95.22%</td>
<td>45.01%</td>
<td>89.29%</td>
<td>56.19%</td>
</tr>
<tr>
<td>RandMix</td>
<td>96.70%</td>
<td>39.41%</td>
<td>90.36%</td>
<td>23.51%</td>
</tr>
<tr>
<td>AdvMix</td>
<td>97.56%</td>
<td>84.29%</td>
<td>92.29%</td>
<td>74.55%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nominal</td>
<td>83.52%</td>
<td>3.31%</td>
<td>85.16%</td>
<td>3.51%</td>
</tr>
<tr>
<td>mixup (( \alpha = 0.2 ))</td>
<td>85.16%</td>
<td>3.51%</td>
<td>84.94%</td>
<td>2.91%</td>
</tr>
<tr>
<td>Cutout</td>
<td>84.94%</td>
<td>2.91%</td>
<td>90.88%</td>
<td>1.47%</td>
</tr>
<tr>
<td>CutMix</td>
<td>85.67%</td>
<td>1.47%</td>
<td>81.43%</td>
<td>52.92%</td>
</tr>
<tr>
<td>AT ( \ell_\infty ) with ( \epsilon = 4/255 )</td>
<td>79.46%</td>
<td>62.71%</td>
<td>79.46%</td>
<td>62.71%</td>
</tr>
<tr>
<td>AT ( \ell_\infty ) with ( \epsilon = 8/255 )</td>
<td>84.49%</td>
<td>3.19%</td>
<td>84.49%</td>
<td>3.19%</td>
</tr>
<tr>
<td>RandMix</td>
<td>85.65%</td>
<td>69.55%</td>
<td>85.65%</td>
<td>69.55%</td>
</tr>
<tr>
<td>AdvMix</td>
<td>85.65%</td>
<td>69.55%</td>
<td>85.65%</td>
<td>69.55%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nominal</td>
<td>78.05%</td>
<td>0.23%</td>
<td>76.80%</td>
<td>0.03%</td>
</tr>
<tr>
<td>mixup (( \alpha = 0.2 ))</td>
<td>76.80%</td>
<td>0.03%</td>
<td>76.59%</td>
<td>0.14%</td>
</tr>
<tr>
<td>Cutout</td>
<td>76.59%</td>
<td>0.14%</td>
<td>78.50%</td>
<td>0.19%</td>
</tr>
<tr>
<td>CutMix</td>
<td>78.50%</td>
<td>0.19%</td>
<td>76.61%</td>
<td>9.74%</td>
</tr>
<tr>
<td>AT ( \ell_\infty ) with ( \epsilon = 4/255 )</td>
<td>74.39%</td>
<td>5.68%</td>
<td>74.39%</td>
<td>5.68%</td>
</tr>
<tr>
<td>AT ( \ell_\infty ) with ( \epsilon = 8/255 )</td>
<td>76.41%</td>
<td>0.42%</td>
<td>76.41%</td>
<td>0.42%</td>
</tr>
<tr>
<td>RandMix</td>
<td>79.47%</td>
<td>47.95%</td>
<td>79.47%</td>
<td>47.95%</td>
</tr>
<tr>
<td>AdvMix</td>
<td>79.47%</td>
<td>47.95%</td>
<td>79.47%</td>
<td>47.95%</td>
</tr>
</tbody>
</table>
C. Code snippets

This section shows how to implement Algorithms 1 and 2 in TensorFlow 2 [57]. Below is Algorithm 1.

```python
import numpy as np
import tensorflow as tf
import imagenet # Custom package which contains 'VGG16'.
import stylegan # Custom package which contains 'Generator'.

# Hyper-parameters.
num_steps = 2000
def learning_rate_schedule(t):
    if t < 1500:
        return .05
    return .01
mixing_level = 8

# Optimizer.
learning_rate = tf.Variable(0., trainable=False, name='learning_rate')
optimizer = tf.keras.optimizers.Adam(learning_rate)

def learn:**
    generator = stylegan.Generator()

# Create a VGG network.
# 'vgg' has the following function:
# - get_activations(x) which computes the VGG activations at its 1st block 2nd convolution,
# 3rd block 2nd convolution, 4th block 2nd convolution.
vgg = imagenet.VGG16()

def feature_loss(a, b):
    s = float(np.prod(a.shape.as_list()))
    return tf.reduce_sum(tf.square(a - b)) / s

def compute_loss(z):
    # Computes the loss.
    generated_image = generator.synthesize(z)
    activations_of_generated_image = {
        [generated_image] + vgg(tf.image.resize(generated_image, 225, 225))
    }

    # Reconstruction and perceptual loss.
    zipped_activations = zip(activations_of_generated_image, activations_of_target_image)

    loss = sum([feature_loss(g, t) for g, t in zipped_activations])

    # Generated a randomly mixed image.
    random_latents = tf.random.normal(shape=(generator.latent_size,))
    random_disentangled_latents = generator.map(random_latents)
    mixed_disentangled_latents = tf.concat([
        z[:mixing_level, :],
        random_disentangled_latents[mixing_level:, :]
    ])
```
mixed_image = generator.synthesize(mixed_disentangled_latents)

# Perceptual loss on mixed image.
activations_of_mixed_image = vgg(tf.image.resize(mixed_image, 225, 225))

for g, t in zip(activations_of_mixed_image, activations_of_target_image[1:])
    loss += .2 * feature_loss(g, t)

return loss

for step in range(num_steps):
    with tf.GradientTape() as tape:
        tape.watch(disentangled_latents)
        loss = compute_loss(disentangled_latents)

    grads = tape.gradient(loss, [disentangled_latents])
    learning_rate.assign(learning_rate_schedule(step))
    optimizer.apply(grads, [disentangled_latents])

Below is Algorithm 2.

import tensorflow as tf
import stylegan # Custom package which contains 'Generator'.

# Hyper-parameters.
mixing_level = 8
epsilon = .03
num_restarts = 5 # Or 10 for evaluation (the higher the better).
num_steps = 7 # Or 20 for evaluation.
step_size = .005

generator = stylegan.Generator()

# 'classifier' is the current classification model we are training or evaluating.
# 'classifier' has the following function:
# - get_logits(x) which returns the logits of x.
classifier = load_current_model()

# 'z' are precomputed latents obtained through Algorithm 1.
# For the FFHQ model, 'z' has shape [18, 512].
# 'y' is the class of the original image that can be synthesized through z.
z = load_disentangled_latents()
z_parallel = z[:mixing_level, :]
y = get_class_of(z)

def project(x, x0, eps):
    return x0 + tf.clip_by_value(x - x0, -eps, eps)

# Initialization.
x = generator.synthesize(z)
logits = classifier.get_logits(x)
highest_loss = tf.nn.sparse_softmax_cross_entropy_with_logits(y, logits)
adversarial_x = x

for _ in range(num_restarts):
    # Pick a random set of disentangled latents.
z = tf.random.normal(shape=(generator.latent_size,))
z_perp = z_perp_original = generator.map(z)[mixing_level:, :]

for _ in range(num_steps):
    with tf.GradientTape() as tape:
        tape.watch(z_perp)
        # Mix latents.
        mixed_z = tf.concat([z_parallel, z_perp], axis=0)
        # Generate and classify.
        x = generator.synthesize(mixed_z)
        logits = classifier.get_logits(x)
        # Cross-entropy loss.
        loss = tf.nn.sparse_softmax_cross_entropy_with_logits(y, logits)
        if loss > highest_loss:
            adversarial_x = x
            highest_loss = loss

        grad = tape.gradient(loss, [z_perp])[0]
        z_perp += step_size * tf.math.sign(grad)  # Gradient ascent.
        z_perp = project(z_perp, z_perp_original, epsilon)

    # Last step.
    mixed_z = tf.concat([z_parallel, z_perp], axis=0)
    x = generator.synthesize(mixed_z)
    logits = classifier.get_logits(x)
    loss = tf.nn.sparse_softmax_cross_entropy_with_logits(y, logits)
    if loss > highest_loss:
        adversarial_x = x
        highest_loss = loss