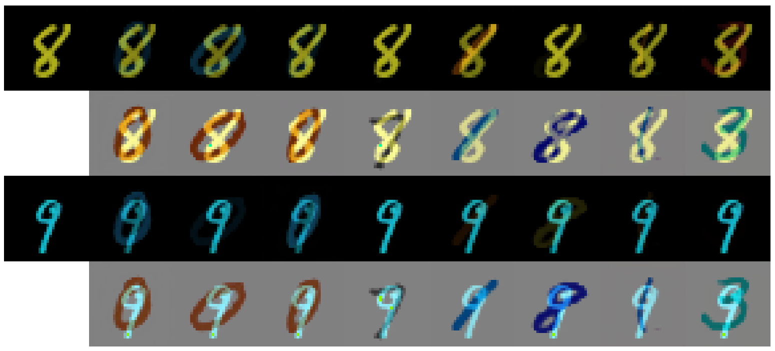
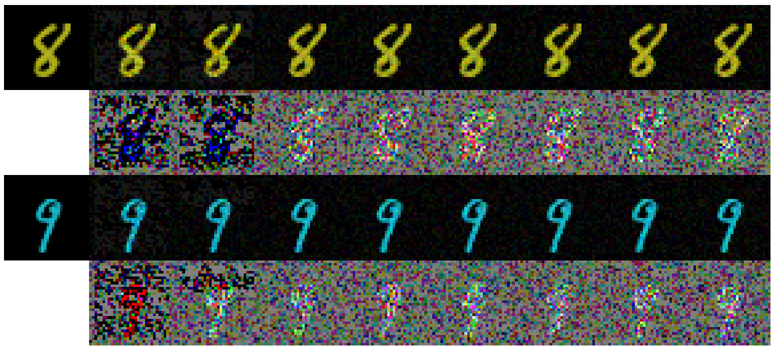
Achieving Robustness in the Wild via Adversarial Mixing with Disentangled Representations (Supplementary Material)

A. Additional examples

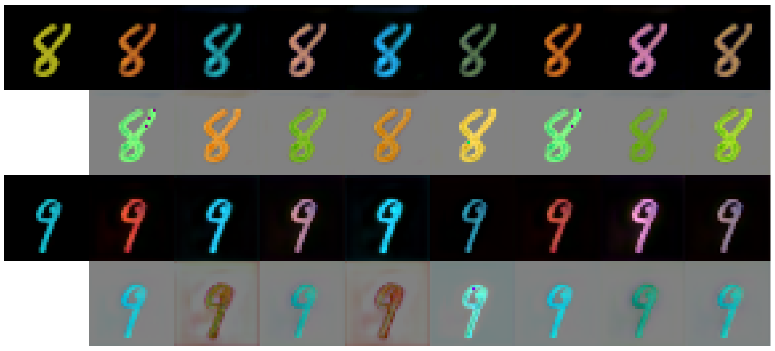
Figure 9 shows additional examples of perturbations obtained on Color-MNIST by *(a) mixup*, *(b)* adversarial attacks on ℓ_∞-bounded perturbations of size $\epsilon = 0.1$, and *(c)* our method *AdvMix*. Figure 10 shows examples on CELEBA. The underlying classifier is the nominally trained convolutional 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.



(a) *mixup*

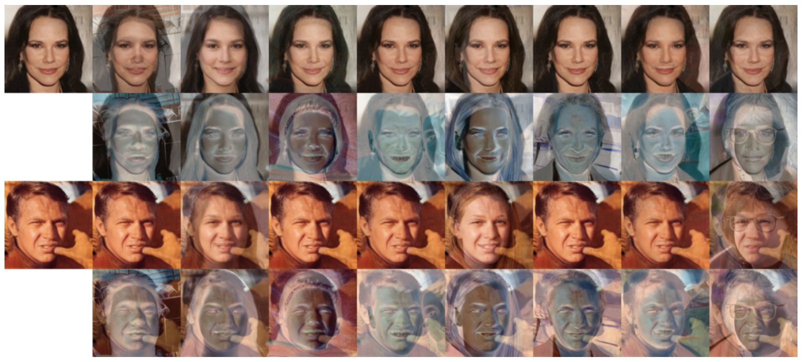


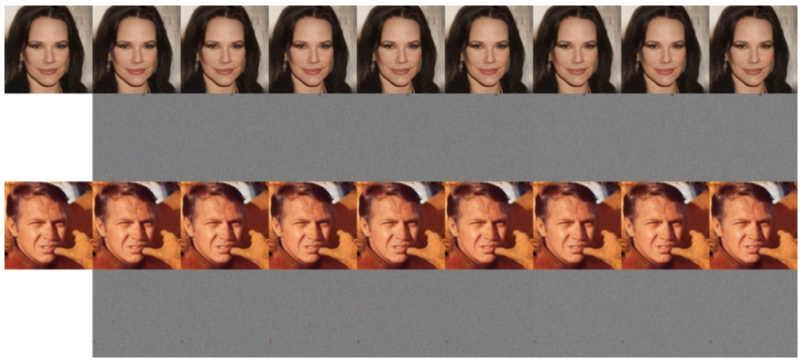
(b) Adversarial Training ($\epsilon = 0.1$)



(c) *AdvMix* or *RandMix*

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.


(a) *mixup*


(b) Adversarial Training ($\epsilon = 8/255$)

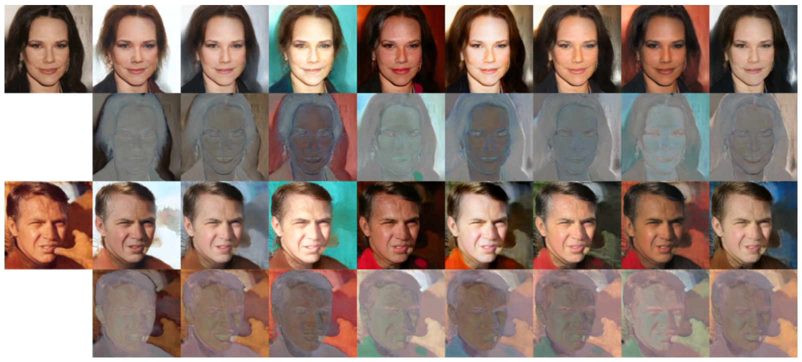

(c) *AdvMix* or *RandMix*

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.

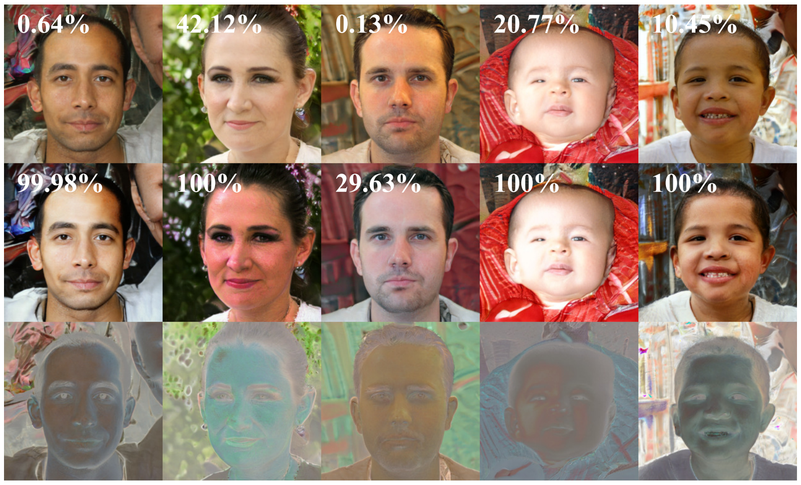


Figure 11. Example of perturbations obtained by *AdvMix* on randomly generated images. The top row consists of images generated by a *StyleGAN* model – all these images are classified as "not smiling" by the nominal classifier (the numbers indicate the classifier output probability for "smiling"). The second row consists of adversarial perturbations obtained by *AdvMix*. The last row shows the rescaled differences between the original images and their variant.

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.

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

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)

# Create the StyleGAN generator.
# `generator` has a the following properties:
# - latent_size: Number of latent coordinates (for the FFHQ model, this is equal to 512)
# - average_disentangled_latents: Average disentangled latents.
# It also has the following functions:
# - map(z) which runs the mapping operation that transforms latents into disentangled latents.
# - synthesize(z_disentangled) which generates an image from disentangled latents.
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()

# Target image.
target_image = load_target_image()
activations_of_target_image = [target_image] + vgg(tf.image.resize(target_image, 225, 225))

# `disentangled_latents` will be optimized to match `target_image`.
# For the FFHQ model, `disentangled_latents` has shape [18, 512].
disentangled_latents = tf.Variable(average_disentangled_latents, name='disentangled_latents')


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:, :]
    ], axis=0)
    mixed_image = generator.synthesize(mixed_disentangled_latents)

    # Perceptual loss on mixed image.
    activations_of_mixed_image = vgg(tf.image.resize(mixed_image, 225, 225))
    zipped_activations = zip(activations_of_mixed_image, activations_of_target_image[1:])
    loss += sum([.2 * feature_loss(g, t) for g, t in zipped_activations])

    return loss


# Run the optimization procedure.
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.

```python
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

# Create the StyleGAN generator.
# `generator` has a the following properties:
# - latent_size: Number of latent coordinates (for the FFHQ model, this is equal to 512)
# - average_disentangled_latents: Average disentangled latents.
# It also has the following functions:
# - map(z) which runs the mapping operation that transforms latents into disentangled latents.
# - synthesize(z_disentangled) which generates an image from disentangled latents.
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
```