Basic workflow with ART for evasion attacks and defences

In this notebook we will show

- how to work with a Keras image classifier in ART
- how ART abstracts from the specific ML/DL backend
- how to apply a Projected Gradient Descent (PGD) evasion attack against that classifier
- how to deploy defences against such attacks
- how to create adversarial samples that can bypass those defences

Added by Ed Herranz (10/15/2022):

- how to protect against a white box attack agsint the defences

Install and load prerequisites

You can preinstall all prerequisites by uncommenting and running the following cell.

```
!pip install adversarial-robustness-toolbox==1.12.1
!pip install tensorflow==2.10.0
!pip install keras==2.10.0
```

Show hidden output

```


# Load basic dependencies:
import warnings
warnings.filterwarnings('ignore')

%matplotlib inline
import matplotlib.pyplot as plt
import sys
import numpy as np

# Disable TensorFlow eager execution:
import tensorflow as tf
if tf.executing_eagerly():
    tf.compat.v1.disable_eager_execution()

# Load Keras dependencies:
from tensorflow.keras.applications.resnet50 import ResNet50, preprocess_input
from tensorflow.keras.preprocessing import image

# Load ART dependencies:
from art.estimators.classification import KerasClassifier
from art.attacks.evasion import ProjectedGradientDescent
from art.defences.preprocessor import SpatialSmoothing
from art.utils import to_categorical
```

```
26
27 # Install ImageNet stubs:
28 !{sys.executable} -m pip install git+https://github.com/nottombrown/imagenet_stubs
29 import imagenet_stubs
30 from imagenet_stubs.imagenet_2012_labels import name_to_label, label_to_name
```

```
Collecting git+https://github.com/nottombrown/imagenet_stubs
  Cloning https://github.com/nottombrown/imagenet_stubs to /tmp/pip-req-build-86xs3s0v
  Running command git clone --filter=blob:none --quiet https://github.com/nottombrown/imagenet_st
  Resolved https://github.com/nottombrown/imagenet_stubs to commit 0b501276f54cbf45b8e1b67dccbdcb
  Preparing metadata (setup.py) ... done
Building wheels for collected packages: imagenet-stubs
  Building wheel for imagenet-stubs (setup.py) ... done
  Created wheel for imagenet-stubs: filename=imagenet_stubs-0.0.7-py3-none-any.whl size=794799 sh
  Stored in directory: /tmp/pip-ephem-wheel-cache-csi0o5g8/wheels/11/fe/f6/71c84cfe4ee113c44af282
Successfully built imagenet-stubs
Installing collected packages: imagenet-stubs
Successfully installed imagenet-stubs-0.0.7
```

Load images

We are going to load a set of 16 example images for illustration purposes.

```
1
2 images_list = list()
3 for i, image_path in enumerate(imagenet_stubs.get_image_paths()):
4     im = image.load_img(image_path, target_size=(224, 224))
5     im = image.img_to_array(im)
6     images_list.append(im)
7     print(image_path)
8     if 'beagle.jpg' in image_path:
9         # get gazelle index
10        gazelle_idx = i
11 images = np.array(images_list)
12
```

```
/usr/local/lib/python3.10/dist-packages/imagenet_stubs/images/beagle.jpg
/usr/local/lib/python3.10/dist-packages/imagenet_stubs/images/sleeping_bag.jpg
/usr/local/lib/python3.10/dist-packages/imagenet_stubs/images/centipede.jpg
/usr/local/lib/python3.10/dist-packages/imagenet_stubs/images/notebook_computer.jpg
/usr/local/lib/python3.10/dist-packages/imagenet_stubs/images/koala.jpg
/usr/local/lib/python3.10/dist-packages/imagenet_stubs/images/standard_poodle.jpg
/usr/local/lib/python3.10/dist-packages/imagenet_stubs/images/tractor.jpg
/usr/local/lib/python3.10/dist-packages/imagenet_stubs/images/mitten.jpg
/usr/local/lib/python3.10/dist-packages/imagenet_stubs/images/gazelle.jpg
/usr/local/lib/python3.10/dist-packages/imagenet_stubs/images/manhole_cover.jpg
/usr/local/lib/python3.10/dist-packages/imagenet_stubs/images/malamute.jpg
/usr/local/lib/python3.10/dist-packages/imagenet_stubs/images/rock_crab.jpg
/usr/local/lib/python3.10/dist-packages/imagenet_stubs/images/unicycle.jpg
/usr/local/lib/python3.10/dist-packages/imagenet_stubs/images/marmoset.jpg
/usr/local/lib/python3.10/dist-packages/imagenet_stubs/images/bagel.jpg
/usr/local/lib/python3.10/dist-packages/imagenet_stubs/images/flagpole.jpg
```

The images all have a resolution of 224 x 224 pixels, and 3 color channels (RGB).

```
print('Number of images:', images.shape[0])
print('Dimension of images:', images.shape[1], 'x', images.shape[2], 'pixels')
print('Number of color channels:', images.shape[3], '(RGB)')
```

```
Number of images: 16
Dimension of images: 224 x 224 pixels
Number of color channels: 3 (RGB)
```

As default choice, we are going to use the unicycle image for illustration purposes. But you could use any other of the 16 images in the following (just change the value of the `idx` variable).

```
images_list = list()
for i, image_path in enumerate(imagenet_stubs.get_image_paths()):
    im = image.load_img(image_path, target_size=(224, 224))
    im = image.img_to_array(im)
    images_list.append(im)
    print(image_path)
    if 'beagle.jpg' in image_path:
        # get gazelle index
        gazelle_idx = i
images = np.array(images_list)

```

```
/usr/local/lib/python3.10/dist-packages/imagenet_stubs/images/beagle.jpg
/usr/local/lib/python3.10/dist-packages/imagenet_stubs/images/sleeping_bag.jpg
/usr/local/lib/python3.10/dist-packages/imagenet_stubs/images/centipede.jpg
/usr/local/lib/python3.10/dist-packages/imagenet_stubs/images/notebook_computer.jpg
/usr/local/lib/python3.10/dist-packages/imagenet_stubs/images/koala.jpg
/usr/local/lib/python3.10/dist-packages/imagenet_stubs/images/standard_poodle.jpg
/usr/local/lib/python3.10/dist-packages/imagenet_stubs/images/tractor.jpg
/usr/local/lib/python3.10/dist-packages/imagenet_stubs/images/mitten.jpg
/usr/local/lib/python3.10/dist-packages/imagenet_stubs/images/gazelle.jpg
/usr/local/lib/python3.10/dist-packages/imagenet_stubs/images/manhole_cover.jpg
/usr/local/lib/python3.10/dist-packages/imagenet_stubs/images/malamute.jpg
/usr/local/lib/python3.10/dist-packages/imagenet_stubs/images/rock_crab.jpg
/usr/local/lib/python3.10/dist-packages/imagenet_stubs/images/unicycle.jpg
/usr/local/lib/python3.10/dist-packages/imagenet_stubs/images/marmoset.jpg
/usr/local/lib/python3.10/dist-packages/imagenet_stubs/images/bagel.jpg
/usr/local/lib/python3.10/dist-packages/imagenet_stubs/images/flagpole.jpg
```

```
idx = gazelle_idx

plt.figure(figsize=(8,8))
plt.imshow(images[idx] / 255)
plt.axis('off')
plt.show()

```

Load ResNet50 classifier

Next we are going to use a state-of-the-art classifier on those images.

```
# loads the pretrained ResNet50 model:
model = ResNet50(weights='imagenet')
```

```
WARNING:tensorflow:From /usr/local/lib/python3.10/dist-packages/keras/layers/normalization/batch_
Instructions for updating:
Colocations handled automatically by placer.
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/resnet/resnet5
102967424/102967424 [==============================] - 6s 0us/step
```

Let's look at the prediction that this model yields for the selected image:

```
# expand the input dimension and apply the preprocessing required for ResNet50:
```

```
x = np.expand_dims(images[idx].copy(), axis=0)
x = preprocess_input(x)

# apply the model, determine the predicted label and confidence:
pred = model.predict(x)
label = np.argmax(pred, axis=1)[0]
confidence = pred[:,label][0]

print('Prediction:', label_to_name(label), '- confidence {0:.2f}'.format(confidence))
```

```
Prediction: beagle - confidence 0.94
```

So the model correctly tells us that this image shows a unicycle/monocycle, which is good :-)

Next we will create an ART KerasClassifier wrapper around the model.
We need to take care of the `preprocess_input` logic that has to be applied:

- swap the order of the color channels (RGB -> BGR)
- subtract the channel means

```
from art.preprocessing.preprocessing import Preprocessor

class ResNet50Preprocessor(Preprocessor):

    def __call__(self, x, y=None):
        return preprocess_input(x.copy()), y

    def estimate_gradient(self, x, gradient):
        return gradient[..., ::-1]
```

```
# Create the ART preprocessor and classifier wrapper:
preprocessor = ResNet50Preprocessor()
classifier = KerasClassifier(clip_values=(0, 255), model=model, preprocessing=preprocessor)
```

```
WARNING:art.estimators.classification.keras:Keras model has no loss set. Classifier tries to use
```

Now we will apply the classifier object to obtain the prediction.

Note: we have to swap the color channel order (from RGB to BGR) before feeding the input to the classifier

```
# Same as for the original model, we expand the dimension of the inputs.
x_art = np.expand_dims(images[idx], axis=0)

# Then apply the model through the classifier API, determine the predicted label and confidence:
pred = classifier.predict(x_art)
label = np.argmax(pred, axis=1)[0]
confidence = pred[:,label][0]

print('Prediction:', label_to_name(label), '- confidence {0:.2f}'.format(confidence))
```

```
Prediction: beagle - confidence 0.94
```

So through the classifier API we obtain the same predictions as from the raw model, but now we have an abstraction from the actual backend (e.g. Keras).

The classifier wrapper allows us to call other functions besides predict.

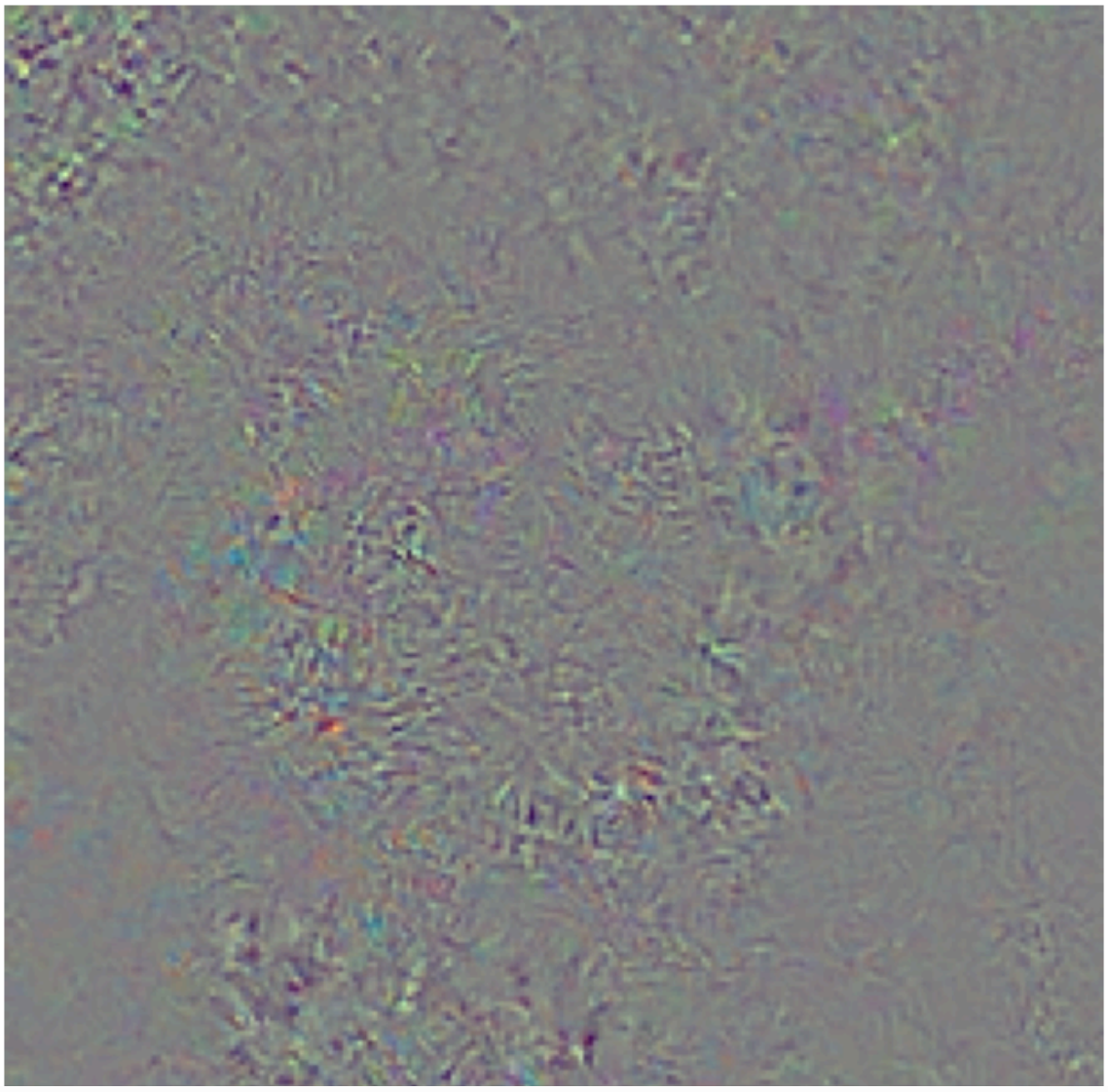
For example, we can obtain the **loss gradient** of the classifier, which is used in many of the algorithms for adversarial sample generation:

```
loss_gradient = classifier.loss_gradient(x=x_art, y=to_categorical([label], nb_classes=1000))

# plot the loss gradient.
# First, swap color channels back to RGB order:
loss_gradient_plot = loss_gradient[0]

# normalize loss gradient values to be in [0,1]:
loss_gradient_min = np.min(loss_gradient)
loss_gradient_max = np.max(loss_gradient)
loss_gradient_plot = (loss_gradient_plot - loss_gradient_min)/(loss_gradient_max - loss_gradient_m

# Show plot:
plt.figure(figsize=(8,8)); plt.imshow(loss_gradient_plot); plt.axis('off'); plt.show()
```



Create adversarial samples

Next, we are going to create an adversarial sample.
We are going to use **Projected Gradient Descent (PGD)**, which is one of the strongest existing attacks.
We will first perform an **untargeted** adversarial attack.

```
# Create the attacker:
adv = ProjectedGradientDescent(classifier, targeted=False, max_iter=10, eps_step=1, eps=5)

# Generate the adversarial sample:
x_art_adv = adv.generate(x_art)

# Plot the adversarial sample (note: we swap color channels back to RGB order):
plt.figure(figsize=(8,8)); plt.imshow(x_art_adv[0] / 255); plt.axis('off'); plt.show()

# And apply the classifier to it:
pred_adv = classifier.predict(x_art_adv)
label_adv = np.argmax(pred_adv, axis=1)[0]
```

```
confidence_adv = pred_adv[:, label_adv][0]
print('Prediction:', label_to_name(label_adv), '- confidence {0:.2f}'.format(confidence_adv))
```

PGD - Random Initializations: 100% 1/1 [00:05<00:00, 5.42s/it]

Next, we will perform a **targeted attack** where we pick the class that we want the classifier to predict on the adversarial sample.
Below is the list of labels and class names - make your pick!

```
for i in range(1000):
    print('label', i, '-', label_to_name(i))
```

```
label 951 - lemon
label 952 - fig
label 953 - pineapple, ananas
label 954 - banana
label 955 - jackfruit, jak, jack
label 956 - custard apple
label 957 - pomegranate
label 958 - hay
label 959 - carbonara
label 960 - chocolate sauce, chocolate syrup
label 961 - dough
label 962 - meat loaf, meatloaf
label 963 - pizza, pizza pie
label 964 - potpie
label 965 - burrito
label 966 - red wine
label 967 - espresso
label 968 - cup
label 969 - eggnog
label 970 - alp
label 971 - bubble
label 972 - cliff, drop, drop-off
label 973 - coral reef
label 974 - geyser
label 975 - lakeside, lakeshore
label 976 - promontory, headland, head, foreland
label 977 - sandbar, sand bar
label 978 - seashore, coast, seacoast, sea-coast
label 979 - valley, vale
label 980 - volcano
label 981 - ballplayer, baseball player
label 982 - groom, bridegroom
label 983 - scuba diver
label 984 - rapeseed
label 985 - daisy
label 986 - yellow lady's slipper, yellow lady-slipper, Cypripedium calceolus, Cypripedium pa
label 987 - corn
label 988 - acorn
label 989 - hip, rose hip, rosehip
label 990 - buckeye, horse chestnut, conker
label 991 - coral fungus
label 992 - agaric
label 993 - gyromitra
label 994 - stinkhorn, carrion fungus
label 995 - earthstar
label 996 - hen-of-the-woods, hen of the woods, Polyporus frondosus, Grifola frondosa
label 997 - bolete
label 998 - ear, spike, capitulum
label 999 - toilet tissue, toilet paper, bathroom tissue
```

As default, let's get this image misclassified as black swan (label 100)!

```
1 target_label = 100
```

Now let's perform the targeted attack:

```
# Set the configuration to a targeted attack:
adv.set_params(targeted=True)

# Generate the adversarial sample:
x_art_adv = adv.generate(x_art, y=to_categorical([target_label]))

# Plot the adversarial sample (note: we swap color channels back to RGB order):
plt.figure(figsize=(8,8)); plt.imshow(x_art_adv[0] / 255); plt.axis('off'); plt.show()

# And apply the classifier to it:
pred_adv = classifier.predict(x_art_adv)
label_adv = np.argmax(pred_adv, axis=1)[0]
confidence_adv = pred_adv[:, label_adv][0]
print('Prediction:', label_to_name(label_adv), '- confidence {0:.2f}'.format(confidence_adv))
```

PGD - Random Initializations: 100% 1/1 [00:04<00:00, 4.24s/it]

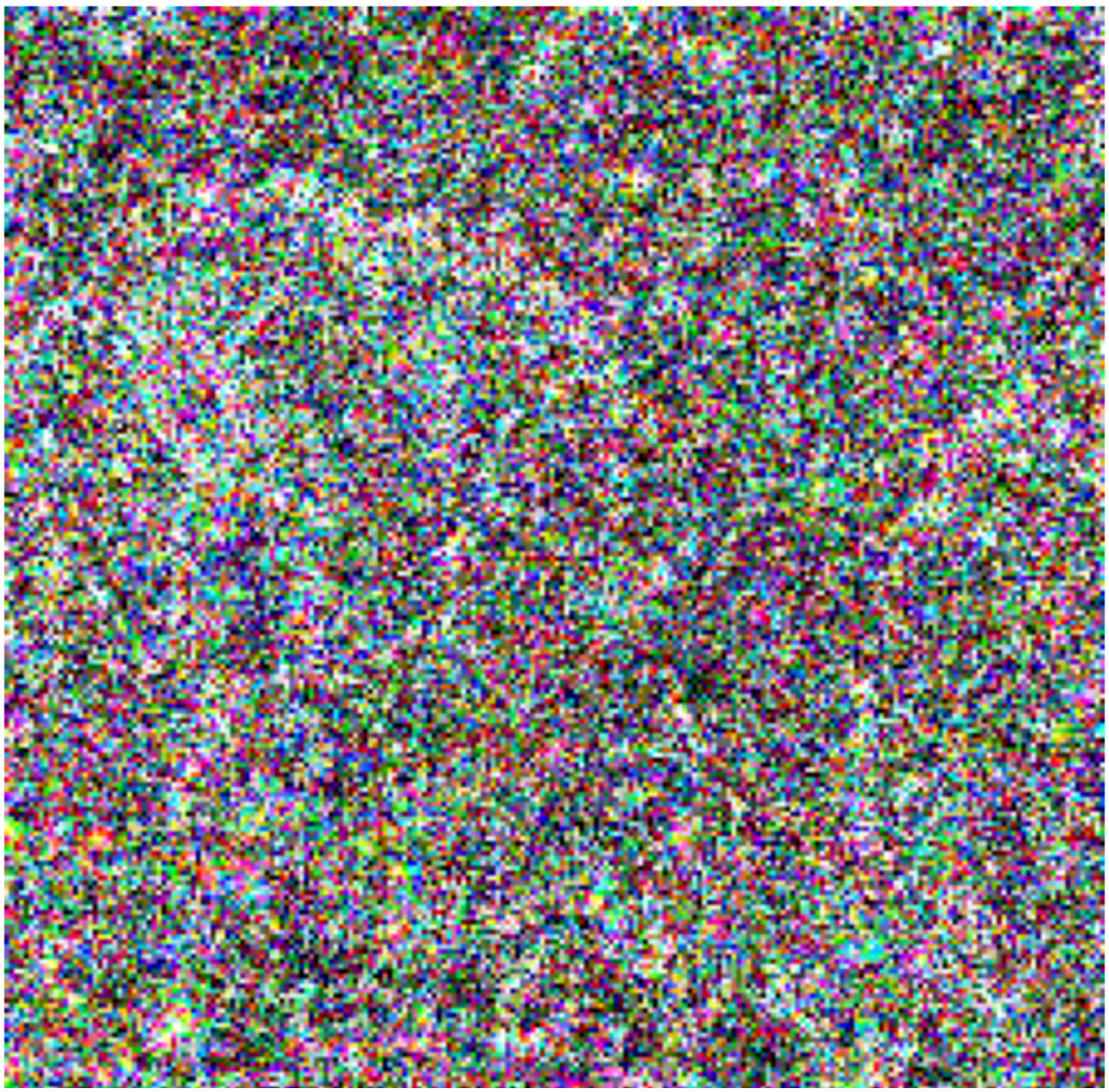
We can measure the quantity of perturbation that was added to the image using different ℓ_p norms.
Note: the PGD attack controls the ℓ_∞ norm via the `epsilon` parameter.

```
l_0 = int(99*len(np.where(np.abs(x_art[0] - x_art_adv[0])>0.5)[0]) / (224*224*3)) + 1
l_1 = int(99*np.sum(np.abs(x_art[0] - x_art_adv[0])) / np.sum(np.abs(x_art[0]))) + 1
l_2 = int(99*np.linalg.norm(x_art[0] - x_art_adv[0]) / np.linalg.norm(x_art[0])) + 1
l_inf = int(99*np.max(np.abs(x_art[0] - x_art_adv[0])) / 255) + 1

print('Perturbation l_0 norm: %d%%' % l_0)
print('Perturbation l_1 norm: %d%%' % l_1)
print('Perturbation l_2 norm: %d%%' % l_2)
print('Noise l_inf norm: %d%%' % l_inf)

# Let's also plot the absolute amount of adversarial pixel perturbations:
pert = np.abs(x_art[0] - x_art_adv[0])[..., ::-1]
pert_min = np.min(pert)
pert_max = np.max(pert)
plt.figure(figsize=(8,8)); plt.imshow((pert - pert_min) / (pert_max - pert_min)); plt.axis('off');
```

```
Perturbation l_0 norm: 77%
Perturbation l_1 norm: 2%
Perturbation l_2 norm: 3%
Noise l_inf norm: 2%
```



Apply defences

Next we are going to apply a simple input preprocessing defence: Spatial Smoothing.
Ideally, we want this defence to result in correct predictions when applied both to the original and the adversarial images.

```
from art.estimators.classification import KerasClassifier
from tensorflow.keras.applications.resnet50 import ResNet50

# Load the pre-trained ResNet50 model (or another model)
model = ResNet50(weights='imagenet')

# Wrap the model with ART's KerasClassifier
art_model = KerasClassifier(model=model, clip_values=(0, 255))

```

```
WARNING:art.estimators.classification.keras:Keras model has no loss set. Classifier tries to use
```

```
from art.attacks.evasion import ProjectedGradientDescent

# Initialize the PGD attack with the wrapped model
attack = ProjectedGradientDescent(estimator=art_model, eps=0.1, eps_step=0.01, max_iter=40)

```

```
from art.attacks.evasion import ProjectedGradientDescent

#  `art_model` is the wrapped model using KerasClassifier
attack = ProjectedGradientDescent(estimator=art_model, eps=0.1, eps_step=0.01, max_iter=40)

```

```
adversarial_examples = attack.generate(x=images)
```

```
import pandas as pd
```

```
from art.defences.preprocessor import FeatureSqueezing

# Initialize results list for Feature Squeezing
fs_results = []

# Loop over bit_depth values from 1 to 4
for bit_depth in range(1, 5):
    print(f"\nApplying Feature Squeezing with bit depth: {bit_depth}")

    # Apply Feature Squeezing defense with bit depth and clip values
    fs_defense = FeatureSqueezing(bit_depth=bit_depth, clip_values=(0, 255))

```

```
    # Apply the defense to the original and adversarial examples
    squeezed_original, _ = fs_defense(images)
    squeezed_adversarial, _ = fs_defense(adversarial_examples)

    # Get model predictions and confidence for the original and adversarial examples
    predictions_original = art_model.predict(squeezed_original)
    predictions_adversarial = art_model.predict(squeezed_adversarial)

    # Extract class and confidence (highest probability) for each prediction
    original_class = np.argmax(predictions_original, axis=1)
    original_confidence = np.max(predictions_original, axis=1)

    adversarial_class = np.argmax(predictions_adversarial, axis=1)
    adversarial_confidence = np.max(predictions_adversarial, axis=1)

    # Record results for this bit_depth
    for i in range(len(images)):  # Assuming batch processing
        fs_results.append({
            'Bit Depth': bit_depth,
            'Original Class': original_class[i],
            'Original Confidence': original_confidence[i],
            'Adversarial Class': adversarial_class[i],
            'Adversarial Confidence': adversarial_confidence[i]
        })

# Convert the results to a DataFrame for easy viewing
fs_results_df = pd.DataFrame(fs_results)

# Display the Feature Squeezing results
print("Feature Squeezing Results:")
display(fs_results_df)

```

```
Applying Feature Squeezing with bit depth: 1

Applying Feature Squeezing with bit depth: 2

Applying Feature Squeezing with bit depth: 3

Applying Feature Squeezing with bit depth: 4
Feature Squeezing Results:
```

	Bit Depth	Original Class	Original Confidence	Adversarial Class	Adversarial Confidence
0	1	641	0.805059	641	0.805059
1	1	980	0.676289	980	0.676289
2	1	556	0.886150	556	0.886150
3	1	644	0.533406	644	0.533406
4	1	310	0.437171	310	0.437171
...	...	...	...	...	...
59	4	119	0.828790	119	0.828790
60	4	671	0.900068	671	0.900068
61	4	380	0.722489	380	0.722489
62	4	931	0.878354	931	0.878354
63	4	557	0.994115	557	0.994115

Next steps: Generate code with `fs_results_df` View recommended plots New interactive sheet

```python
# Initialize the SpatialSmoothing defence.
ss = SpatialSmoothing(window_size=3)

# Apply the defence to the original input and to the adversarial sample, respectively:
x_art_def, _ = ss(x_art)
x_art_adv_def, _ = ss(x_art_adv)

# Compute the classifier predictions on the preprocessed inputs:
pred_def = classifier.predict(x_art_def)
label_def = np.argmax(pred_def, axis=1)[0]
confidence_def = pred_def[:, label_def][0]

pred_adv_def = classifier.predict(x_art_adv_def)
label_adv_def = np.argmax(pred_adv_def, axis=1)[0]
confidence_adv_def = pred_adv_def[:, label_adv_def][0]

# Print the predictions:
print('Prediction of original sample:', label_to_name(label_def), '- confidence {0:.2f}'.format(co
print('Prediction of adversarial sample:', label_to_name(label_adv_def),
      '- confidence {0:.2f}'.format(confidence_adv_def))

```

```
# Show the preprocessed adversarial sample:
plt.figure(figsize=(8,8)); plt.imshow(x_art_adv_def[0] / 255); plt.axis('off'); plt.show()
```

```
Prediction of original sample: beagle - confidence 0.59
Prediction of adversarial sample: basset, basset hound - confidence 0.29
```

```
from art.estimators.classification import KerasClassifier
from art.attacks.evasion import ProjectedGradientDescent

# Wrap the Keras model with ART's KerasClassifier
art_model = KerasClassifier(model=model, clip_values=(0, 255))

# Initialize the PGD attack with the wrapped model
attack = ProjectedGradientDescent(estimator=art_model, eps=0.1, eps_step=0.01, max_iter=40)

# Generate adversarial examples
adversarial_examples = attack.generate(x=images)

```

```
WARNING:art.estimators.classification.keras:Keras model has no loss set. Classifier tries to use
PGD - Random Initializations: 100%                                    1/1 [04:10<00:00, 250.77s/it]
```

```python
from art.defences.preprocessor import SpatialSmoothing

# Initialize Spatial Smoothing
spatial_smoothing = SpatialSmoothing(window_size=3, clip_values=(0, 255))

# Apply the defense to the original and adversarial examples
smoothing_original, _ = spatial_smoothing(images)
smoothing_adversarial, _ = spatial_smoothing(adversarial_examples)

```

```python
spatial_smoothing_results = []

# Get predictions after applying spatial smoothing
predictions_original = art_model.predict(smoothing_original)
predictions_adversarial = art_model.predict(smoothing_adversarial)

# Collect results
for i in range(len(images)):
    spatial_smoothing_results.append({
        'Image Index': i,
        'Original Prediction': np.argmax(predictions_original[i]),
        'Adversarial Prediction': np.argmax(predictions_adversarial[i]),
        'Original Confidence': np.max(predictions_original[i]),
        'Adversarial Confidence': np.max(predictions_adversarial[i])
    })

```

```python
import pandas as pd

# Assuming you have a list called 'spatial_smoothing_results'
results_df = pd.DataFrame(spatial_smoothing_results)

```

```python
# Import the libraries
import pandas as pd

# Display the DataFrame
print("Spatial Smoothing Results:")
print(results_df)

from IPython.display import display
display(results_df)

```

Spatial Smoothing Results:

	Image Index	Original Prediction	Adversarial Prediction
0	0	178	178
1	1	797	797
2	2	79	79
3	3	681	681
4	4	105	105
5	5	267	267
6	6	866	866
7	7	658	658
8	8	353	353
9	9	640	640
10	10	250	250
11	11	119	119
12	12	880	880
13	13	380	380
14	14	931	931
15	15	557	557

	Original Confidence	Adversarial Confidence
0	0.365629	0.513440
1	0.521881	0.709239
2	0.914952	0.892044
3	0.744339	0.724044
4	0.973281	0.939759
5	0.873207	0.853666
6	0.887200	0.875825
7	0.814135	0.736477
8	0.981161	0.978037
9	0.975706	0.971683
10	0.914891	0.899212
11	0.829537	0.790049
12	0.991283	0.987001
13	0.822916	0.780002
14	0.780934	0.654879
15	0.920533	0.880460

	Image Index	Original Prediction	Adversarial Prediction	Original Confidence	Adversarial Confidence
0	0	178	178	0.365629	0.513440
1	1	797	797	0.521881	0.709239
2	2	79	79	0.914952	0.892044
3	3	681	681	0.744339	0.724044
4	4	105	105	0.973281	0.939759
5	5	267	267	0.873207	0.853666
6	6	866	866	0.887200	0.875825
7	7	658	658	0.814135	0.736477
8	8	353	353	0.981161	0.978037
9	9	640	640	0.975706	0.971683
10	10	250	250	0.914891	0.899212
11	11	119	119	0.829537	0.790049

12	12	880	880	0.991283	0.987001
13	13	380	380	0.822916	0.780002
14	14	931	931	0.780934	0.654879

Next steps: Generate code with `results_df` | View recommended plots | New interactive sheet

Perform adaptive whitebox attack to defeat defences

Next we are going to mount an adaptive whitebox attack in which the attacker aims at defeating the defence that we just put into place.

First, we create a classifier which incorporates the defence:

```
classifier_def = KerasClassifier(preprocessing=preprocessor, preprocessing_defences=[ss], clip_va
                                  model=model)

# apply this classifier to the adversarial sample from before:
pred_def = classifier_def.predict(x_art_adv)
label_def = np.argmax(pred_def, axis=1)[0]
confidence_def = pred_def[:, label_def][0]

print('Prediction:', label_to_name(label_def), '- confidence {0:.2f}'.format(confidence_def))
```

```
WARNING:art.estimators.classification.keras:Keras model has no loss set. Classifier tries to use
Prediction: basset, basset hound - confidence 0.29
```

We observe that this classifier reproduces the prediction that we had obtained before by manually applying the input preprocessing defence.

Now we create an adversarial sample against the *defended* classifier.
As we are going to see, this adversarial sample is able to bypass the input preprocessing defence.

```
# Create the attacker.
# Note we use a larger number of iterations to achieve the same level of confidence in the misclas
adv_def = ProjectedGradientDescent(classifier_def, targeted=True, max_iter=40, eps_step=1, eps=5)

# Generate the adversarial sample:
x_art_adv_def = adv_def.generate(x_art, y=to_categorical([target_label]))

# Plot the adversarial sample (note: we swap color channels back to RGB order):
plt.figure(figsize=(8,8)); plt.imshow(x_art_adv_def[0] / 255); plt.axis('off'); plt.show()

# And apply the classifier to it:
pred_adv = classifier_def.predict(x_art_adv_def)
label_adv = np.argmax(pred_adv, axis=1)[0]
```

```
confidence_adv = pred_adv[:, label_adv][0]
print('Prediction:', label_to_name(label_adv), '- confidence {0:.2f}'.format(confidence_adv))
```

```
PGD - Random Initializations: 100%  1/1 [00:22<00:00, 22.54s/it]
```

Let's also look at the ℓ_p norms of that adversarial perturbation:

```
l_0 = int(99*len(np.where(np.abs(x_art[0] - x_art_adv_def[0])>0.5)[0]) / (224*224*3)) + 1
l_1 = int(99*np.sum(np.abs(x_art[0] - x_art_adv_def[0])) / np.sum(np.abs(x_art[0]))) + 1
l_2 = int(99*np.linalg.norm(x_art[0] - x_art_adv_def[0]) / np.linalg.norm(x_art[0])) + 1
l_inf = int(99*np.max(np.abs(x_art[0] - x_art_adv_def[0])) / 255) + 1

print('Perturbation l_0 norm: %d%%' % l_0)
print('Perturbation l_1 norm: %d%%' % l_1)
print('Perturbation l_2 norm: %d%%' % l_2)
print('Noise l_inf norm: %d%%' % l_inf)
```

```
Perturbation l_0 norm: 90%
Perturbation l_1 norm: 3%
Perturbation l_2 norm: 3%
```

```
    Noise l_inf norm: 2%
```

```
#This part defend against the whitebox attack to defeat defences

from art.defences.postprocessor import GaussianNoise
from art.defences.preprocessor import JpegCompression

plt.figure(figsize=(8,8)); plt.imshow(x_art_adv_def[0] / 255); plt.axis('off'); plt.show()


classifier_def1 = KerasClassifier(preprocessing=preprocessor,  postprocessing_defences=GaussianNoi
                                  model=model)

# the GaussianNoise post-processing predicts the wrong class for the  whitebox adversarial attacke
pred_adv = classifier_def1.predict(x_art_adv_def)
label_adv = np.argmax(pred_adv, axis=1)[0]
confidence_adv = pred_adv[:, label_adv][0]
print('Prediction of whitebox adversarial attacked image with GaussiaNoise Post Processing:', labe


# the GaussianNoise post-processing predicts the wrong class for the  original image
pred_adv = classifier_def1.predict(x_art)
label_adv = np.argmax(pred_adv, axis=1)[0]
confidence_adv = pred_adv[:, label_adv][0]
print('Prediction of original image with GaussianNoise Post Processing:', label_to_name(label_adv)

classifier_def2 = KerasClassifier(preprocessing=preprocessor, preprocessing_defences=JpegCompressi
                                    model=model)

# the JpegCompression pre-processing predicts the correct class for the  whitebox adversarial atta
pred_adv = classifier_def2.predict(x_art_adv_def)
label_adv = np.argmax(pred_adv, axis=1)[0]
confidence_adv = pred_adv[:, label_adv][0]
print('Prediction of whitebox adversarial attacked image with JpegCompression Pre-Processing:', la

# the JpegCompression pre-processing predicts the correct class for the original image
pred_adv = classifier_def2.predict(x_art)
label_adv = np.argmax(pred_adv, axis=1)[0]
confidence_adv = pred_adv[:, label_adv][0]
print('Prediction of original image with JpegCompression Pre-Processing', label_to_name(label_adv)
```

```
WARNING:art.estimators.classification.keras:Keras model has no loss set. Classifier tries to use
WARNING:art.estimators.classification.keras:Keras model has no loss set. Classifier tries to use
Prediction of whitebox adversarial attacked image with GaussiaNoise Post Processing: electric loc
Prediction of original image with GaussianNoise Post Processing: tray - confidence 0.01
Prediction of whitebox adversarial attacked image with JpegCompression Pre-Processing: beagle - c
Prediction of original image with JpegCompression Pre-Processing beagle - confidence 0.76
```

```
from art.defences.preprocessor import JpegCompression, GaussianAugmentation

# the pixel values of images range from 0 to 255
clip_values = (0, 255)

# Initialize results lists
jpeg_results = []
gaussian_results = []

# 'art_model' is the ART-wrapped model and 'adversarial_examples' are already generated

# Loop over JPEG compression rates
for compression_rate in range(20, 95, 5):
    print(f"Applying JPEG Compression with quality: {compression_rate}")
```

```
15
16      # Apply JPEG compression defense with clip_values
17      jpeg_defense = JpegCompression(clip_values=clip_values, quality=compression_rate)
18      compressed_original, _ = jpeg_defense(images)
19      compressed_adversarial, _ = jpeg_defense(adversarial_examples)
20
21      # Get predictions for the original and adversarial examples
22      predictions_original = art_model.predict(compressed_original)
23      predictions_adversarial = art_model.predict(compressed_adversarial)
24
25      # Extract class and confidence for each prediction
26      original_class = np.argmax(predictions_original, axis=1)
27      original_confidence = np.max(predictions_original, axis=1)
28      adversarial_class = np.argmax(predictions_adversarial, axis=1)
29      adversarial_confidence = np.max(predictions_adversarial, axis=1)
30
31      # Record results for this JPEG compression rate
32      for i in range(len(images)):  # Assuming batch processing
33          jpeg_results.append({
34              'Compression Rate': compression_rate,
```