Foolbox Documentation

Release 1.8.0

Jonas Rauber & Wieland Brendel

Jun 29, 2019

User Guide

Foolbox is a Python toolbox to create adversarial examples that fool neural networks.

It comes with support for many frameworks to build models including

- TensorFlow
- PyTorch
- Theano
- Keras
- Lasagne
- MXNet

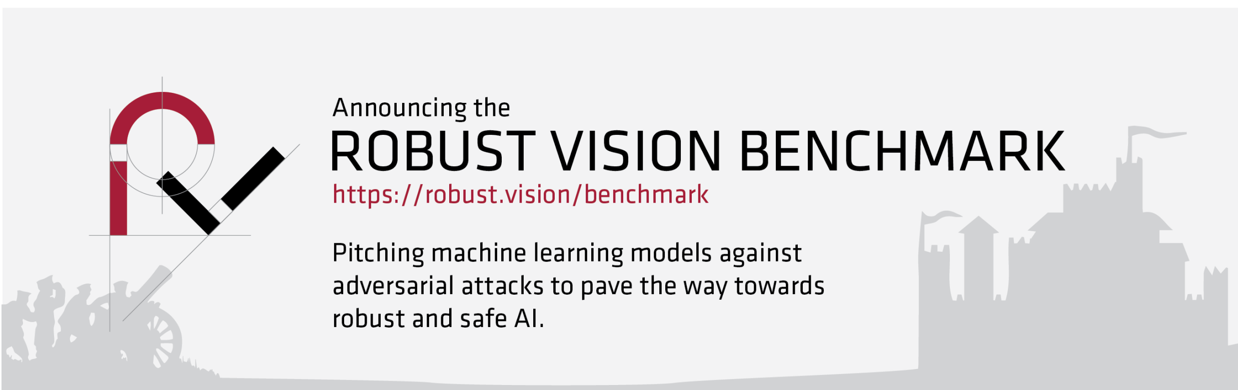
and it is easy to extend to other frameworks.

In addition, it comes with a **large collection of adversarial attacks**, both gradient-based attacks as well as black-box attacks. See *foolbox.attacks* for details.

The source code and a minimal working example can be found on GitHub.

CHAPTER 1

Robust Vision Benchmark



You might want to have a look at our recently announced Robust Vision Benchmark, a benchmark for adversarial attacks and the robustness of machine learning models.

1.1 Installation

Foolbox is a Python package to create adversarial examples. We test using Python 2.7, 3.5 and 3.6, but other versions of Python might work as well. **We recommend using Python 3!**.

1.1.1 Stable release

You can install the latest stable release of Foolbox from PyPI using *pip*:

```
pip install foolbox
```

Make sure that *pip* installs packages for Python 3, otherwise you might need to use *pip3* instead of *pip*.

1.1.2 Development version

Alternatively, you can install the latest development version of Foolbox from GitHub. We try to keep the master branch stable, so this version should usually work fine. Feel free to open an issue on GitHub if you encounter any problems.

```
pip install https://github.com/bethgelab/foolbox/archive/master.zip
```

1.1.3 Contributing to Foolbox

If you would like to contribute the development of Foolbox, install it in editable mode:

```
git clone https://github.com/bethgelab/foolbox.git
cd foolbox
pip install --editable .
```

To contribute your changes, you will need to fork the Foolbox repository on GitHub. You can than add it as a remote:

```
git remote rename origin upstream
git remote add origin https://github.com/<your-github-name>/foolbox.git
```

You can now commit your changes, push them to your fork and create a pull-request to contribute them to Foolbox.

1.2 Tutorial

This tutorial will show you how an adversarial attack can be used to find adversarial examples for a model.

1.2.1 Creating a model

For the tutorial, we will target *VGG19* implemented in *TensorFlow*, but it is straight forward to apply the same to other models or other frameworks such as *Theano* or *PyTorch*.

```
import tensorflow as tf

images = tf.placeholder(tf.float32, (None, 224, 224, 3))
preprocessed = vgg_preprocessing(images)
logits = vgg19(preprocessed)
```

To turn a model represented as a standard TensorFlow graph into a model that can be attacked by the Adversarial Toolbox, all we have to do is to create a new *TensorFlowModel* instance:

```
from foolbox.models import TensorFlowModel

model = TensorFlowModel(images, logits, bounds=(0, 255))
```

1.2.2 Specifying the criterion

To run an adversarial attack, we need to specify the type of adversarial we are looking for. This can be done using the `Criterion` class.

```
from foolbox.criteria import TargetClassProbability

target_class = 22
criterion = TargetClassProbability(target_class, p=0.99)
```

1.2.3 Running the attack

Finally, we can create and apply the attack:

```
from foolbox.attacks import LBFGSAttack

attack = LBFGSAttack(model, criterion)

image = np.asarray(Image.open('example.jpg'))
label = np.argmax(model.predictions(image))

adversarial = attack(image, label=label)
```

1.2.4 Visualizing the adversarial examples

To plot the adversarial example we can use *matplotlib*:

```
import matplotlib.pyplot as plt

plt.subplot(1, 3, 1)
plt.imshow(image)

plt.subplot(1, 3, 2)
plt.imshow(adversarial)

plt.subplot(1, 3, 3)
plt.imshow(adversarial - image)
```

1.3 Examples

Here you can find a collection of examples how Foolbox models can be created using different deep learning frameworks and some full-blown attack examples at the end.

1.3.1 Creating a model

Keras: ResNet50

```
import keras
import numpy as np
import foolbox

keras.backend.set_learning_phase(0)
kmodel = keras.applications.resnet50.ResNet50(weights='imagenet')
preprocessing = (np.array([104, 116, 123]), 1)
```

(continues on next page)

(continued from previous page)

```
model = foolbox.models.KerasModel(kmodel, bounds=(0, 255),␣
↪preprocessing=preprocessing)

image, label = foolbox.utils.imagenet_example()
# ::-1 reverses the color channels, because Keras ResNet50 expects BGR instead of RGB
print(np.argmax(model.predictions(image[:, :, ::-1])), label)
```

PyTorch: ResNet18

You might be interested in checking out the full PyTorch example at the end of this document.

```
import torchvision.models as models
import numpy as np
import foolbox

# instantiate the model
resnet18 = models.resnet18(pretrained=True).cuda().eval()  # for CPU, remove cuda()
mean = np.array([0.485, 0.456, 0.406]).reshape((3, 1, 1))
std = np.array([0.229, 0.224, 0.225]).reshape((3, 1, 1))
model = foolbox.models.PyTorchModel(resnet18, bounds=(0, 1), num_classes=1000,␣
↪preprocessing=(mean, std))

image, label = foolbox.utils.imagenet_example(data_format='channels_first')
image = image / 255
print(np.argmax(model.predictions(image)), label)
```

TensorFlow: VGG19

First, create the model in TensorFlow.

```
import tensorflow as tf
from tensorflow.contrib.slim.nets import vgg
import numpy as np
import foolbox

images = tf.placeholder(tf.float32, shape=(None, 224, 224, 3))
preprocessed = images - [123.68, 116.78, 103.94]
logits, _ = vgg.vgg_19(preprocessed, is_training=False)
restorer = tf.train.Saver(tf.trainable_variables())

image, _ = foolbox.utils.imagenet_example()
```

Then transform it into a Foolbox model using one of these four options:

Option 1

This option is recommended if you want to keep the code as short as possible. It makes use of the TensorFlow session created by Foolbox internally if no default session is set.

```
with foolbox.models.TensorFlowModel(images, logits, (0, 255)) as model:
    restorer.restore(model.session, '/path/to/vgg_19.ckpt')
    print(np.argmax(model.predictions(image)))
```

Option 2

This option is recommended if you want to create the TensorFlow session yourself.

```
with tf.Session() as session:
    restorer.restore(session, '/path/to/vgg_19.ckpt')
    model = foolbox.models.TensorFlowModel(images, logits, (0, 255))
    print(np.argmax(model.predictions(image)))
```

Option 3

This option is recommended if you want to avoid nesting context managers, e.g. during interactive development.

```
session = tf.InteractiveSession()
restorer.restore(session, '/path/to/vgg_19.ckpt')
model = foolbox.models.TensorFlowModel(images, logits, (0, 255))
print(np.argmax(model.predictions(image)))
session.close()
```

Option 4

This is possible, but usually one of the other options should be preferred.

```
session = tf.Session()
with session.as_default():
    restorer.restore(session, '/path/to/vgg_19.ckpt')
    model = foolbox.models.TensorFlowModel(images, logits, (0, 255))
    print(np.argmax(model.predictions(image)))
session.close()
```

1.3.2 Applying an attack

Once you created a Foolbox model (see the previous section), you can apply an attack.

FGSM (GradientSignAttack)

```
# create a model (see previous section)
fmodel = ...

# get source image and label
image, label = foolbox.utils.imagenet_example()

# apply attack on source image
attack   = foolbox.attacks.FGSM(fmodel)
adversarial = attack(image[:,:,::-1], label)
```

1.3.3 Creating an untargeted adversarial for a PyTorch model

```
import foolbox
import torch
import torchvision.models as models
import numpy as np

# instantiate the model
resnet18 = models.resnet18(pretrained=True).eval()
if torch.cuda.is_available():
    resnet18 = resnet18.cuda()
mean = np.array([0.485, 0.456, 0.406]).reshape((3, 1, 1))
std = np.array([0.229, 0.224, 0.225]).reshape((3, 1, 1))
fmodel = foolbox.models.PyTorchModel(
    resnet18, bounds=(0, 1), num_classes=1000, preprocessing=(mean, std))

# get source image and label
image, label = foolbox.utils.imagenet_example(data_format='channels_first')
image = image / 255.  # because our model expects values in [0, 1]

print('label', label)
print('predicted class', np.argmax(fmodel.predictions(image)))

# apply attack on source image
attack = foolbox.attacks.FGSM(fmodel)
adversarial = attack(image, label)

print('adversarial class', np.argmax(fmodel.predictions(adversarial)))
```

outputs

```
label 282
predicted class 282
adversarial class 281
```

To plot image and adversarial, don't forget to move the channel axis to the end before passing them to matplotlib's imshow, e.g. using `np.transpose(image, (1, 2, 0))`.

1.3.4 Creating a targeted adversarial for the Keras ResNet model

```
import foolbox
from foolbox.models import KerasModel
from foolbox.attacks import LBFGSAttack
from foolbox.criteria import TargetClassProbability
import numpy as np
import keras
from keras.applications.resnet50 import ResNet50
from keras.applications.resnet50 import preprocess_input
from keras.applications.resnet50 import decode_predictions

keras.backend.set_learning_phase(0)
kmodel = ResNet50(weights='imagenet')
preprocessing = (np.array([104, 116, 123]), 1)
fmodel = KerasModel(kmodel, bounds=(0, 255), preprocessing=preprocessing)

image, label = foolbox.utils.imagenet_example()
```

(continues on next page)

(continued from previous page)

```
# run the attack
attack = LBFGSAttack(model=fmodel, criterion=TargetClassProbability(781, p=.5))
adversarial = attack(image[:, :, ::-1], label)

# show results
print(np.argmax(fmodel.predictions(adversarial)))
print(foolbox.utils.softmax(fmodel.predictions(adversarial))[781])
adversarial_rgb = adversarial[np.newaxis, :, :, ::-1]
preds = kmodel.predict(preprocess_input(adversarial_rgb.copy()))
print("Top 5 predictions (adversarial: ", decode_predictions(preds, top=5))
```

outputs

```
781
0.832095
Top 5 predictions (adversarial:  [[('n04149813', 'scoreboard', 0.83013469), (
↪'n03196217', 'digital_clock', 0.030192226), ('n04152593', 'screen', 0.016133979), (
↪'n04141975', 'scale', 0.011708578), ('n03782006', 'monitor', 0.0091574294)]]
```

1.4 Advanced

The `Adversarial` class provides an advanced way to specify the adversarial example that should be found by an attack and provides detailed information about the created adversarial. In addition, it provides a way to improve a previously found adversarial example by re-running an attack.

1.4.1 Implicit

```
model = TensorFlowModel(images, logits, bounds=(0, 255))
criterion = TargetClassProbability('ostrich', p=0.99)
attack = LBFGSAttack(model, criterion)
```

Running the attack by passing image and label will implicitly create an `Adversarial` instance. By passing *unpack=False* we tell the attack to return the `Adversarial` instance rather than the actual image.

```
adversarial = attack(image, label=label, unpack=False)
```

We can then get the actual image using the `image` attribute:

```
adversarial_image = adversarial.image
```

1.4.2 Explicit

```
model = TensorFlowModel(images, logits, bounds=(0, 255))
criterion = TargetClassProbability('ostrich', p=0.99)
attack = LBFGSAttack()
```

We can also create the `Adversarial` instance ourselves and then pass it to the attack.

```
adversarial = Adversarial(model, criterion, image, label)
attack(adversarial)
```

Again, we can get the image using the `image` attribute:

```
adversarial_image = adversarial.image
```

This approach gives us more flexibility and allows us to specify a different distance measure:

```
distance = MeanAbsoluteDistance
adversarial = Adversarial(model, criterion, image, label, distance=distance)
```

1.5 Model Zoo

This tutorial will show you how the model zoo can be used to run your attack against a robust model.

1.5.1 Downloading a model

For this tutorial, we will download the *Madry et al. CIFAR10 challenge* robust model implemented in *TensorFlow* and run a *FGSM (GradienSignAttack)* against it.

```
from foolbox import zoo

# download the model
model = zoo.get_model(url="https://github.com/bethgelab/cifar10_challenge.git")

# read image and label
image = ...
label = ...

# apply attack on source image
attack  = foolbox.attacks.FGSM(model)
adversarial = attack(image[:,:,::-1], label)
```

1.6 Development

To install Foolbox in editable mode, see the installation instructions under *Contributing to Foolbox*.

1.6.1 Running Tests

pytest

To run the tests, you need to have pytest and pytest-cov installed. Afterwards, you can simply run `pytest` in the root folder of the project. Some tests will require TensorFlow, PyTorch and the other frameworks, so to run all tests, you need to have all of them installed.

flake8

Foolbox follows the PEP 8 style guide for Python code. To check for violations, we use flake8 and run it like this:

```
flake8 --ignore E402,E741 .
```

1.6.2 New Adversarial Attacks

Foolbox makes it easy to develop new adversarial attacks that can be applied to arbitrary models.

To implement an attack, simply subclass the `Attack` class, implement the `__call__()` method and decorate it with the **:decorator:'call_decorator'**. The **:decorator:'call_decorator'** will make sure that your `__call__()` implementation will be called with an instance of the `Adversarial` class. You can use this instance to ask for model predictions and gradients, get the original image and its label and more. In addition, the `Adversarial` instance automatically keeps track of the best adversarial amongst all the images tested by the attack. That way, the implementation of the attack can focus on the attack logic.

1.7 FAQ

How does Foolbox handle inputs that are misclassified without any perturbation? The attacks will not be run and instead the unperturbed input is returned as an *adversarial* with distance 0 to the clean input.

What happens if an attack fails? The attack will return *None* and the distance will be *np.inf*.

Why is the returned adversarial not misclassified by my model? Most likely you have a discrepancy between how you evaluate your model and how you told Foolbox to evaluate it. For example, you might not be using the same preprocessing. Compare the output of the *predictions* method of the Foolbox model instance with your model's output (logits). This problem can also be caused by non-deterministic models. Make sure that your model is not stochastic and always returns the same output when given the same input. In rare cases it can also be that a seemlingly deterministic model becomes numerically stochastic around the decision boundary (e.g. because of non-deterministic floating point *reduce_sum* operations). You can always check *adversarial.output* and *adversarial.adversarial_class* to see the output Foolbox got from your model when deciding that this was an adversarial.

Why are the gradients multiplied by the bounds (*max_ - min_*)? This scaling is meant to make hyperparameters such as the *epsilon* for FGSM independent of the bounds. *epsilon = 0.1* thus means that you perturb the image by 10% relative to the *max - max* range (which could for example go from 0 to 1 or from 0 to 255).

1.8 `foolbox.models`

Provides classes to wrap existing models in different framworks so that they provide a unified API to the attacks.

1.8.1 Models

`Model`	Base class to provide attacks with a unified interface to models.
`DifferentiableModel`	Base class for differentiable models that provide gradients.

Continued on next page

Table 1 – continued from previous page

TensorFlowModel	Creates a *Model* instance from existing *TensorFlow* tensors.
TensorFlowEagerModel	Creates a *Model* instance from a *TensorFlow* model using eager execution.
PyTorchModel	Creates a *Model* instance from a *PyTorch* module.
KerasModel	Creates a *Model* instance from a *Keras* model.
TheanoModel	Creates a *Model* instance from existing *Theano* tensors.
LasagneModel	Creates a *Model* instance from a *Lasagne* network.
MXNetModel	Creates a *Model* instance from existing *MXNet* symbols and weights.
MXNetGluonModel	Creates a *Model* instance from an existing *MXNet Gluon* Block.

1.8.2 Wrappers

ModelWrapper	Base class for models that wrap other models.
DifferentiableModelWrapper	Base class for models that wrap other models and provide gradient methods.
ModelWithoutGradients	Turns a model into a model without gradients.
ModelWithEstimatedGradients	Turns a model into a model with gradients estimated by the given gradient estimator.
CompositeModel	Combines predictions of a (black-box) model with the gradient of a (substitute) model.

1.8.3 Detailed description

class foolbox.models.**Model**(*bounds*, *channel_axis*, *preprocessing=(0, 1)*)

Base class to provide attacks with a unified interface to models.

The *Model* class represents a model and provides a unified interface to its predictions. Subclasses must implement batch_predictions and num_classes.

Model instances can be used as context managers and subclasses can require this to allocate and release resources.

Parameters

bounds [tuple] Tuple of lower and upper bound for the pixel values, usually (0, 1) or (0, 255).

channel_axis [int] The index of the axis that represents color channels.

preprocessing: 2-element tuple with floats or numpy arrays Elementwises preprocessing of input; we first subtract the first element of preprocessing from the input and then divide the input by the second element.

batch_predictions(*self*, *images*)

Calculates predictions for a batch of images.

Parameters

images [*numpy.ndarray*] Batch of images with shape (batch size, height, width, channels).

Returns

numpy.ndarray Predictions (logits, i.e. before the softmax) with shape (batch size, number of classes).

See also:

`predictions()`

num_classes (*self*)
Determines the number of classes.

Returns

int The number of classes for which the model creates predictions.

predictions (*self*, *image*)
Convenience method that calculates predictions for a single image.

Parameters

image [*numpy.ndarray*] Image with shape (height, width, channels).

Returns

numpy.ndarray Vector of predictions (logits, i.e. before the softmax) with shape (number of classes,).

See also:

`batch_predictions()`

class foolbox.models.**DifferentiableModel** (*bounds*, *channel_axis*, *preprocessing=(0, 1)*)
Base class for differentiable models that provide gradients.

The `DifferentiableModel` class can be used as a base class for models that provide gradients. Subclasses must implement predictions_and_gradient.

A model should be considered differentiable based on whether it provides a `predictions_and_gradient()` method and a `gradient()` method, not based on whether it subclasses `DifferentiableModel`.

A differentiable model does not necessarily provide reasonable values for the gradients, the gradient can be wrong. It only guarantees that the relevant methods can be called.

backward (*self*, *gradient*, *image*)
Backpropagates the gradient of some loss w.r.t. the logits through the network and returns the gradient of that loss w.r.t to the input image.

Parameters

gradient [*numpy.ndarray*] Gradient of some loss w.r.t. the logits.

image [*numpy.ndarray*] Image with shape (height, width, channels).

Returns

gradient [*numpy.ndarray*] The gradient w.r.t the image.

See also:

`gradient()`

gradient (*self*, *image*, *label*)
Calculates the gradient of the cross-entropy loss w.r.t. the image.

The default implementation calls predictions_and_gradient. Subclasses can provide more efficient implementations that only calculate the gradient.

Parameters

image [*numpy.ndarray*] Image with shape (height, width, channels).

label [int] Reference label used to calculate the gradient.

Returns

gradient [*numpy.ndarray*] The gradient of the cross-entropy loss w.r.t. the image. Will have the same shape as the image.

See also:

gradient()

predictions_and_gradient (*self*, *image*, *label*)
Calculates predictions for an image and the gradient of the cross-entropy loss w.r.t. the image.

Parameters

image [*numpy.ndarray*] Image with shape (height, width, channels).

label [int] Reference label used to calculate the gradient.

Returns

predictions [*numpy.ndarray*] Vector of predictions (logits, i.e. before the softmax) with shape (number of classes,).

gradient [*numpy.ndarray*] The gradient of the cross-entropy loss w.r.t. the image. Will have the same shape as the image.

See also:

gradient()

class foolbox.models.**TensorFlowModel** (*images*, *logits*, *bounds*, *channel_axis=3*, *preprocessing=(0, 1)*)
Creates a *Model* instance from existing *TensorFlow* tensors.

Parameters

images [*tensorflow.Tensor*] The input to the model, usually a *tensorflow.placeholder*.

logits [*tensorflow.Tensor*] The predictions of the model, before the softmax.

bounds [tuple] Tuple of lower and upper bound for the pixel values, usually (0, 1) or (0, 255).

channel_axis [int] The index of the axis that represents color channels.

preprocessing: 2-element tuple with floats or numpy arrays Elementwises preprocessing of input; we first subtract the first element of preprocessing from the input and then divide the input by the second element.

backward (*self*, *gradient*, *image*)
Backpropagates the gradient of some loss w.r.t. the logits through the network and returns the gradient of that loss w.r.t to the input image.

Parameters

gradient [*numpy.ndarray*] Gradient of some loss w.r.t. the logits.

image [*numpy.ndarray*] Image with shape (height, width, channels).

Returns

gradient [*numpy.ndarray*] The gradient w.r.t the image.

See also:

`gradient()`

`batch_predictions` (*self*, *images*)

Calculates predictions for a batch of images.

Parameters

images [*numpy.ndarray*] Batch of images with shape (batch size, height, width, channels).

Returns

numpy.ndarray Predictions (logits, i.e. before the softmax) with shape (batch size, number of classes).

See also:

`predictions()`

`classmethod from_keras` (*model*, *bounds*, *input_shape=None*, *channel_axis=3*, *preprocessing=(0, 1)*)

Alternative constructor for a TensorFlowModel that accepts a *tf.keras.Model* instance.

Parameters

model [*tensorflow.keras.Model*] A *tensorflow.keras.Model* that accepts a single input tensor and returns a single output tensor representing logits.

bounds [tuple] Tuple of lower and upper bound for the pixel values, usually (0, 1) or (0, 255).

input_shape [tuple] The shape of a single input, e.g. (28, 28, 1) for MNIST. If None, tries to get the the shape from the model's input_shape attribute.

channel_axis [int] The index of the axis that represents color channels.

preprocessing: 2-element tuple with floats or numpy arrays Elementwises preprocessing of input; we first subtract the first element of preprocessing from the input and then divide the input by the second element.

`gradient` (*self*, *image*, *label*)

Calculates the gradient of the cross-entropy loss w.r.t. the image.

The default implementation calls predictions_and_gradient. Subclasses can provide more efficient implementations that only calculate the gradient.

Parameters

image [*numpy.ndarray*] Image with shape (height, width, channels).

label [int] Reference label used to calculate the gradient.

Returns

gradient [*numpy.ndarray*] The gradient of the cross-entropy loss w.r.t. the image. Will have the same shape as the image.

See also:

gradient()

num_classes(*self*)
Determines the number of classes.

Returns

int The number of classes for which the model creates predictions.

predictions_and_gradient(*self*, *image*, *label*)
Calculates predictions for an image and the gradient of the cross-entropy loss w.r.t. the image.

Parameters

image [*numpy.ndarray*] Image with shape (height, width, channels).

label [int] Reference label used to calculate the gradient.

Returns

predictions [*numpy.ndarray*] Vector of predictions (logits, i.e. before the softmax) with shape (number of classes,).

gradient [*numpy.ndarray*] The gradient of the cross-entropy loss w.r.t. the image. Will have the same shape as the image.

See also:

gradient()

class foolbox.models.**TensorFlowEagerModel**(*model*, *bounds*, *num_classes=None*, *channel_axis=3*, *preprocessing=(0, 1)*)
Creates a *Model* instance from a *TensorFlow* model using eager execution.

Parameters

model [a TensorFlow eager model] The TensorFlow eager model that should be attacked. It will be called with input tensors and should return logits.

bounds [tuple] Tuple of lower and upper bound for the pixel values, usually (0, 1) or (0, 255).

num_classes [int] If None, will try to infer it from the model's output shape.

channel_axis [int] The index of the axis that represents color channels.

preprocessing: 2-element tuple with floats or numpy arrays Elementwises preprocessing of input; we first subtract the first element of preprocessing from the input and then divide the input by the second element.

backward(*self*, *gradient*, *image*)
Backpropagates the gradient of some loss w.r.t. the logits through the network and returns the gradient of that loss w.r.t to the input image.

Parameters

gradient [*numpy.ndarray*] Gradient of some loss w.r.t. the logits.

image [*numpy.ndarray*] Image with shape (height, width, channels).

Returns

gradient [*numpy.ndarray*] The gradient w.r.t the image.

See also:

gradient()

batch_predictions(*self*, *images*)
Calculates predictions for a batch of images.

Parameters

images [*numpy.ndarray*] Batch of images with shape (batch size, height, width, channels).

Returns

numpy.ndarray Predictions (logits, i.e. before the softmax) with shape (batch size, number of classes).

See also:

predictions()

num_classes(*self*)
Determines the number of classes.

Returns

int The number of classes for which the model creates predictions.

predictions_and_gradient(*self*, *image*, *label*)
Calculates predictions for an image and the gradient of the cross-entropy loss w.r.t. the image.

Parameters

image [*numpy.ndarray*] Image with shape (height, width, channels).

label [int] Reference label used to calculate the gradient.

Returns

predictions [*numpy.ndarray*] Vector of predictions (logits, i.e. before the softmax) with shape (number of classes,).

gradient [*numpy.ndarray*] The gradient of the cross-entropy loss w.r.t. the image. Will have the same shape as the image.

See also:

gradient()

class foolbox.models.**PyTorchModel**(*model*, *bounds*, *num_classes*, *channel_axis=1*, *device=None*, *preprocessing=(0, 1)*)
Creates a `Model` instance from a *PyTorch* module.

Parameters

model [*torch.nn.Module*] The PyTorch model that should be attacked.

bounds [tuple] Tuple of lower and upper bound for the pixel values, usually (0, 1) or (0, 255).

num_classes [int] Number of classes for which the model will output predictions.

channel_axis [int] The index of the axis that represents color channels.

device [string] A string specifying the device to do computation on. If None, will default to "cuda:0" if torch.cuda.is_available() or "cpu" if not.

preprocessing: 2-element tuple with floats or numpy arrays Elementwises preprocessing of input; we first subtract the first element of preprocessing from the input and then divide the input by the second element.

backward(*self*, *gradient*, *image*)
Backpropagates the gradient of some loss w.r.t. the logits through the network and returns the gradient of that loss w.r.t to the input image.

Parameters

gradient [*numpy.ndarray*] Gradient of some loss w.r.t. the logits.

image [*numpy.ndarray*] Image with shape (height, width, channels).

Returns

gradient [*numpy.ndarray*] The gradient w.r.t the image.

See also:

`gradient()`

batch_predictions(*self*, *images*)
Calculates predictions for a batch of images.

Parameters

images [*numpy.ndarray*] Batch of images with shape (batch size, height, width, channels).

Returns

numpy.ndarray Predictions (logits, i.e. before the softmax) with shape (batch size, number of classes).

See also:

`predictions()`

num_classes(*self*)
Determines the number of classes.

Returns

int The number of classes for which the model creates predictions.

predictions_and_gradient(*self*, *image*, *label*)
Calculates predictions for an image and the gradient of the cross-entropy loss w.r.t. the image.

Parameters

image [*numpy.ndarray*] Image with shape (height, width, channels).

label [int] Reference label used to calculate the gradient.

Returns

predictions [*numpy.ndarray*] Vector of predictions (logits, i.e. before the softmax) with shape (number of classes,).

gradient [*numpy.ndarray*] The gradient of the cross-entropy loss w.r.t. the image. Will have the same shape as the image.

See also:

`gradient()`

class foolbox.models.**KerasModel**(*model*, *bounds*, *channel_axis=3*, *preprocessing=(0, 1)*, *predicts='probabilities'*)

Creates a *Model* instance from a *Keras* model.

Parameters

model [*keras.models.Model*] The *Keras* model that should be attacked.

bounds [tuple] Tuple of lower and upper bound for the pixel values, usually (0, 1) or (0, 255).

channel_axis [int] The index of the axis that represents color channels.

preprocessing: 2-element tuple with floats or numpy arrays Elementwises preprocessing of input; we first subtract the first element of preprocessing from the input and then divide the input by the second element.

predicts [str] Specifies whether the *Keras* model predicts logits or probabilities. Logits are preferred, but probabilities are the default.

backward(*self*, *gradient*, *image*)

Backpropagates the gradient of some loss w.r.t. the logits through the network and returns the gradient of that loss w.r.t to the input image.

Parameters

gradient [*numpy.ndarray*] Gradient of some loss w.r.t. the logits.

image [*numpy.ndarray*] Image with shape (height, width, channels).

Returns

gradient [*numpy.ndarray*] The gradient w.r.t the image.

See also:

gradient()

batch_predictions(*self*, *images*)

Calculates predictions for a batch of images.

Parameters

images [*numpy.ndarray*] Batch of images with shape (batch size, height, width, channels).

Returns

numpy.ndarray Predictions (logits, i.e. before the softmax) with shape (batch size, number of classes).

See also:

predictions()

num_classes(*self*)

Determines the number of classes.

Returns

int The number of classes for which the model creates predictions.

predictions_and_gradient(*self*, *image*, *label*)

Calculates predictions for an image and the gradient of the cross-entropy loss w.r.t. the image.

Parameters

image [*numpy.ndarray*] Image with shape (height, width, channels).

label [int] Reference label used to calculate the gradient.

Returns

predictions [*numpy.ndarray*] Vector of predictions (logits, i.e. before the softmax) with shape (number of classes,).

gradient [*numpy.ndarray*] The gradient of the cross-entropy loss w.r.t. the image. Will have the same shape as the image.

See also:

`gradient()`

class `foolbox.models.TheanoModel`(*images, logits, bounds, num_classes, channel_axis=1, preprocessing=[0, 1]*)

Creates a `Model` instance from existing *Theano* tensors.

Parameters

images [*theano.tensor*] The input to the model.

logits [*theano.tensor*] The predictions of the model, before the softmax.

bounds [tuple] Tuple of lower and upper bound for the pixel values, usually (0, 1) or (0, 255).

num_classes [int] Number of classes for which the model will output predictions.

channel_axis [int] The index of the axis that represents color channels.

preprocessing: 2-element tuple with floats or numpy arrays Elementwises preprocessing of input; we first subtract the first element of preprocessing from the input and then divide the input by the second element.

`backward`(*self*, *gradient*, *image*)

Backpropagates the gradient of some loss w.r.t. the logits through the network and returns the gradient of that loss w.r.t to the input image.

Parameters

gradient [*numpy.ndarray*] Gradient of some loss w.r.t. the logits.

image [*numpy.ndarray*] Image with shape (height, width, channels).

Returns

gradient [*numpy.ndarray*] The gradient w.r.t the image.

See also:

`gradient()`

`batch_predictions`(*self*, *images*)

Calculates predictions for a batch of images.

Parameters

images [*numpy.ndarray*] Batch of images with shape (batch size, height, width, channels).

Returns

numpy.ndarray Predictions (logits, i.e. before the softmax) with shape (batch size, number of classes).

See also:

predictions()

gradient(*self*, *image*, *label*)

Calculates the gradient of the cross-entropy loss w.r.t. the image.

The default implementation calls predictions_and_gradient. Subclasses can provide more efficient implementations that only calculate the gradient.

Parameters

image [*numpy.ndarray*] Image with shape (height, width, channels).

label [int] Reference label used to calculate the gradient.

Returns

gradient [*numpy.ndarray*] The gradient of the cross-entropy loss w.r.t. the image. Will have the same shape as the image.

See also:

gradient()

num_classes(*self*)

Determines the number of classes.

Returns

int The number of classes for which the model creates predictions.

predictions_and_gradient(*self*, *image*, *label*)

Calculates predictions for an image and the gradient of the cross-entropy loss w.r.t. the image.

Parameters

image [*numpy.ndarray*] Image with shape (height, width, channels).

label [int] Reference label used to calculate the gradient.

Returns

predictions [*numpy.ndarray*] Vector of predictions (logits, i.e. before the softmax) with shape (number of classes,).

gradient [*numpy.ndarray*] The gradient of the cross-entropy loss w.r.t. the image. Will have the same shape as the image.

See also:

gradient()

class foolbox.models.**LasagneModel**(*input_layer*, *logits_layer*, *bounds*, *channel_axis=1*, *preprocessing=(0, 1)*)

Creates a *Model* instance from a *Lasagne* network.

Parameters

input_layer [*lasagne.layers.Layer*] The input to the model.

logits_layer [*lasagne.layers.Layer*] The output of the model, before the softmax.

bounds [tuple] Tuple of lower and upper bound for the pixel values, usually (0, 1) or (0, 255).

channel_axis [int] The index of the axis that represents color channels.

preprocessing: 2-element tuple with floats or numpy arrays Elementwises preprocessing of input; we first subtract the first element of preprocessing from the input and then divide the input by the second element.

backward(*self*, *gradient*, *image*)
Backpropagates the gradient of some loss w.r.t. the logits through the network and returns the gradient of that loss w.r.t to the input image.

Parameters

gradient [*numpy.ndarray*] Gradient of some loss w.r.t. the logits.

image [*numpy.ndarray*] Image with shape (height, width, channels).

Returns

gradient [*numpy.ndarray*] The gradient w.r.t the image.

See also:

gradient()

batch_predictions(*self*, *images*)
Calculates predictions for a batch of images.

Parameters

images [*numpy.ndarray*] Batch of images with shape (batch size, height, width, channels).

Returns

numpy.ndarray Predictions (logits, i.e. before the softmax) with shape (batch size, number of classes).

See also:

predictions()

gradient(*self*, *image*, *label*)
Calculates the gradient of the cross-entropy loss w.r.t. the image.

The default implementation calls predictions_and_gradient. Subclasses can provide more efficient implementations that only calculate the gradient.

Parameters

image [*numpy.ndarray*] Image with shape (height, width, channels).

label [int] Reference label used to calculate the gradient.

Returns

gradient [*numpy.ndarray*] The gradient of the cross-entropy loss w.r.t. the image. Will have the same shape as the image.

See also:

gradient()

num_classes(*self*)
Determines the number of classes.

Returns

int The number of classes for which the model creates predictions.

predictions_and_gradient(*self*, *image*, *label*)
Calculates predictions for an image and the gradient of the cross-entropy loss w.r.t. the image.

Parameters

image [*numpy.ndarray*] Image with shape (height, width, channels).

label [int] Reference label used to calculate the gradient.

Returns

predictions [*numpy.ndarray*] Vector of predictions (logits, i.e. before the softmax) with shape (number of classes,).

gradient [*numpy.ndarray*] The gradient of the cross-entropy loss w.r.t. the image. Will have the same shape as the image.

See also:

gradient()

class foolbox.models.**MXNetModel**(*data*, *logits*, *args*, *ctx*, *num_classes*, *bounds*, *channel_axis=1*, *aux_states=None*, *preprocessing=(0, 1)*)
Creates a *Model* instance from existing *MXNet* symbols and weights.

Parameters

data [*mxnet.symbol.Variable*] The input to the model.

logits [*mxnet.symbol.Symbol*] The predictions of the model, before the softmax.

args [*dictionary mapping str to mxnet.nd.array*] The parameters of the model.

ctx [*mxnet.context.Context*] The device, e.g. mxnet.cpu() or mxnet.gpu().

num_classes [int] The number of classes.

bounds [tuple] Tuple of lower and upper bound for the pixel values, usually (0, 1) or (0, 255).

channel_axis [int] The index of the axis that represents color channels.

aux_states [*dictionary mapping str to mxnet.nd.array*] The states of auxiliary parameters of the model.

preprocessing: 2-element tuple with floats or numpy arrays Elementwises preprocessing of input; we first subtract the first element of preprocessing from the input and then divide the input by the second element.

backward(*self*, *gradient*, *image*)
Backpropagates the gradient of some loss w.r.t. the logits through the network and returns the gradient of that loss w.r.t to the input image.

Parameters

gradient [*numpy.ndarray*] Gradient of some loss w.r.t. the logits.

image [*numpy.ndarray*] Image with shape (height, width, channels).

Returns

gradient [*numpy.ndarray*] The gradient w.r.t the image.

See also:

gradient()

batch_predictions (*self*, *images*)
Calculates predictions for a batch of images.

Parameters

images [*numpy.ndarray*] Batch of images with shape (batch size, height, width, channels).

Returns

numpy.ndarray Predictions (logits, i.e. before the softmax) with shape (batch size, number of classes).

See also:

predictions()

num_classes (*self*)
Determines the number of classes.

Returns

int The number of classes for which the model creates predictions.

predictions_and_gradient (*self*, *image*, *label*)
Calculates predictions for an image and the gradient of the cross-entropy loss w.r.t. the image.

Parameters

image [*numpy.ndarray*] Image with shape (height, width, channels).

label [int] Reference label used to calculate the gradient.

Returns

predictions [*numpy.ndarray*] Vector of predictions (logits, i.e. before the softmax) with shape (number of classes,).

gradient [*numpy.ndarray*] The gradient of the cross-entropy loss w.r.t. the image. Will have the same shape as the image.

See also:

gradient()

class foolbox.models.**MXNetGluonModel** (*block*, *bounds*, *num_classes*, *ctx=None*, *channel_axis=1*, *preprocessing=(0, 1)*)
Creates a `Model` instance from an existing *MXNet Gluon* Block.

Parameters

block [*mxnet.gluon.Block*] The Gluon Block representing the model to be run.

ctx [*mxnet.context.Context*] The device, e.g. mxnet.cpu() or mxnet.gpu().

num_classes [int] The number of classes.

bounds [tuple] Tuple of lower and upper bound for the pixel values, usually (0, 1) or (0, 255).

channel_axis [int] The index of the axis that represents color channels.

preprocessing: 2-element tuple with floats or numpy arrays Elementwises preprocessing of input; we first subtract the first element of preprocessing from the input and then divide the input by the second element.

backward(*self*, *gradient*, *image*)
Backpropagates the gradient of some loss w.r.t. the logits through the network and returns the gradient of that loss w.r.t to the input image.

Parameters

gradient [*numpy.ndarray*] Gradient of some loss w.r.t. the logits.

image [*numpy.ndarray*] Image with shape (height, width, channels).

Returns

gradient [*numpy.ndarray*] The gradient w.r.t the image.

See also:

gradient()

batch_predictions(*self*, *images*)
Calculates predictions for a batch of images.

Parameters

images [*numpy.ndarray*] Batch of images with shape (batch size, height, width, channels).

Returns

numpy.ndarray Predictions (logits, i.e. before the softmax) with shape (batch size, number of classes).

See also:

predictions()

num_classes(*self*)
Determines the number of classes.

Returns

int The number of classes for which the model creates predictions.

predictions_and_gradient(*self*, *image*, *label*)
Calculates predictions for an image and the gradient of the cross-entropy loss w.r.t. the image.

Parameters

image [*numpy.ndarray*] Image with shape (height, width, channels).

label [int] Reference label used to calculate the gradient.

Returns

predictions [*numpy.ndarray*] Vector of predictions (logits, i.e. before the softmax) with shape (number of classes,).

gradient [*numpy.ndarray*] The gradient of the cross-entropy loss w.r.t. the image. Will have the same shape as the image.

See also:

gradient()

class foolbox.models.**ModelWrapper**(*model*)

Base class for models that wrap other models.

This base class can be used to implement model wrappers that turn models into new models, for example by preprocessing the input or modifying the gradient.

Parameters

model [*Model*] The model that is wrapped.

batch_predictions(*self*, *images*)

Calculates predictions for a batch of images.

Parameters

images [*numpy.ndarray*] Batch of images with shape (batch size, height, width, channels).

Returns

numpy.ndarray Predictions (logits, i.e. before the softmax) with shape (batch size, number of classes).

See also:

predictions()

num_classes(*self*)

Determines the number of classes.

Returns

int The number of classes for which the model creates predictions.

predictions(*self*, *image*)

Convenience method that calculates predictions for a single image.

Parameters

image [*numpy.ndarray*] Image with shape (height, width, channels).

Returns

numpy.ndarray Vector of predictions (logits, i.e. before the softmax) with shape (number of classes,).

See also:

batch_predictions()

class foolbox.models.**DifferentiableModelWrapper**(*model*)

Base class for models that wrap other models and provide gradient methods.

This base class can be used to implement model wrappers that turn models into new models, for example by preprocessing the input or modifying the gradient.

Parameters

model [*Model*] The model that is wrapped.

class foolbox.models.**ModelWithoutGradients**(*model*)

Turns a model into a model without gradients.

class foolbox.models.**ModelWithEstimatedGradients**(*model*, *gradient_estimator*)

Turns a model into a model with gradients estimated by the given gradient estimator.

Parameters

model [*Model*] The model that is wrapped.

gradient_estimator [*callable*] Callable taking three arguments (pred_fn, image, label) and returning the estimated gradients. pred_fn will be the batch_predictions method of the wrapped model.

class `foolbox.models.`**`CompositeModel`**(*forward_model*, *backward_model*)

Combines predictions of a (black-box) model with the gradient of a (substitute) model.

Parameters

forward_model [*Model*] The model that should be fooled and will be used for predictions.

backward_model [*Model*] The model that provides the gradients.

`backward`(*self*, *gradient*, *image*)

Backpropagates the gradient of some loss w.r.t. the logits through the network and returns the gradient of that loss w.r.t to the input image.

Parameters

gradient [*numpy.ndarray*] Gradient of some loss w.r.t. the logits.

image [*numpy.ndarray*] Image with shape (height, width, channels).

Returns

gradient [*numpy.ndarray*] The gradient w.r.t the image.

See also:

`gradient()`

`batch_predictions`(*self*, *images*)

Calculates predictions for a batch of images.

Parameters

images [*numpy.ndarray*] Batch of images with shape (batch size, height, width, channels).

Returns

numpy.ndarray Predictions (logits, i.e. before the softmax) with shape (batch size, number of classes).

See also:

`predictions()`

`gradient`(*self*, *image*, *label*)

Calculates the gradient of the cross-entropy loss w.r.t. the image.

The default implementation calls predictions_and_gradient. Subclasses can provide more efficient implementations that only calculate the gradient.

Parameters

image [*numpy.ndarray*] Image with shape (height, width, channels).

label [int] Reference label used to calculate the gradient.

Returns

gradient [*numpy.ndarray*] The gradient of the cross-entropy loss w.r.t. the image. Will have the same shape as the image.

See also:

`gradient()`

`num_classes` (*self*)
Determines the number of classes.

Returns

int The number of classes for which the model creates predictions.

`predictions_and_gradient` (*self*, *image*, *label*)
Calculates predictions for an image and the gradient of the cross-entropy loss w.r.t. the image.

Parameters

image [*numpy.ndarray*] Image with shape (height, width, channels).

label [int] Reference label used to calculate the gradient.

Returns

predictions [*numpy.ndarray*] Vector of predictions (logits, i.e. before the softmax) with shape (number of classes,).

gradient [*numpy.ndarray*] The gradient of the cross-entropy loss w.r.t. the image. Will have the same shape as the image.

See also:

`gradient()`

1.9 `foolbox.criteria`

Provides classes that define what is adversarial.

1.9.1 Criteria

We provide criteria for untargeted and targeted adversarial attacks.

`Misclassification`	Defines adversarials as images for which the predicted class is not the original class.
`TopKMisclassification`	Defines adversarials as images for which the original class is not one of the top k predicted classes.
`OriginalClassProbability`	Defines adversarials as images for which the probability of the original class is below a given threshold.
`ConfidentMisclassification`	Defines adversarials as images for which the probability of any class other than the original is above a given threshold.

TargetClass	Defines adversarials as images for which the predicted class is the given target class.
TargetClassProbability	Defines adversarials as images for which the probability of a given target class is above a given threshold.

1.9.2 Examples

Untargeted criteria:

```
>>> from foolbox.criteria import Misclassification
>>> criterion1 = Misclassification()
```

```
>>> from foolbox.criteria import TopKMisclassification
>>> criterion2 = TopKMisclassification(k=5)
```

Targeted criteria:

```
>>> from foolbox.criteria import TargetClass
>>> criterion3 = TargetClass(22)
```

```
>>> from foolbox.criteria import TargetClassProbability
>>> criterion4 = TargetClassProbability(22, p=0.99)
```

Criteria can be combined to create a new criterion:

```
>>> criterion5 = criterion2 & criterion3
```

1.9.3 Detailed description

class foolbox.criteria.**Criterion**

Base class for criteria that define what is adversarial.

The *Criterion* class represents a criterion used to determine if predictions for an image are adversarial given a reference label. It should be subclassed when implementing new criteria. Subclasses must implement is_adversarial.

is_adversarial(*self*, *predictions*, *label*)

Decides if predictions for an image are adversarial given a reference label.

Parameters

predictions [numpy.ndarray] A vector with the pre-softmax predictions for some image.

label [int] The label of the unperturbed reference image.

Returns

bool True if an image with the given predictions is an adversarial example when the ground-truth class is given by label, False otherwise.

name(*self*)

Returns a human readable name that uniquely identifies the criterion with its hyperparameters.

Returns

str Human readable name that uniquely identifies the criterion with its hyperparameters.

Notes

Defaults to the class name but subclasses can provide more descriptive names and must take hyperparameters into account.

class foolbox.criteria.**Misclassification**

Defines adversarials as images for which the predicted class is not the original class.

See also:

TopKMisclassification

Notes

Uses *numpy.argmax* to break ties.

is_adversarial(*self*, *predictions*, *label*)

Decides if predictions for an image are adversarial given a reference label.

Parameters

predictions [numpy.ndarray] A vector with the pre-softmax predictions for some image.

label [int] The label of the unperturbed reference image.

Returns

bool True if an image with the given predictions is an adversarial example when the ground-truth class is given by label, False otherwise.

name(*self*)

Returns a human readable name that uniquely identifies the criterion with its hyperparameters.

Returns

str Human readable name that uniquely identifies the criterion with its hyperparameters.

Notes

Defaults to the class name but subclasses can provide more descriptive names and must take hyperparameters into account.

class foolbox.criteria.**ConfidentMisclassification**(*p*)

Defines adversarials as images for which the probability of any class other than the original is above a given threshold.

Parameters

p [float] The threshold probability. If the probability of any class other than the original is at least p, the image is considered an adversarial. It must satisfy 0 <= p <= 1.

is_adversarial(*self*, *predictions*, *label*)

Decides if predictions for an image are adversarial given a reference label.

Parameters

predictions [numpy.ndarray] A vector with the pre-softmax predictions for some image.

label [int] The label of the unperturbed reference image.

Returns

bool True if an image with the given predictions is an adversarial example when the ground-truth class is given by label, False otherwise.

name (*self*)
Returns a human readable name that uniquely identifies the criterion with its hyperparameters.

Returns

str Human readable name that uniquely identifies the criterion with its hyperparameters.

Notes

Defaults to the class name but subclasses can provide more descriptive names and must take hyperparameters into account.

class foolbox.criteria.**TopKMisclassification** (*k*)
Defines adversarials as images for which the original class is not one of the top k predicted classes.

For k = 1, the *Misclassification* class provides a more efficient implementation.

Parameters

k [int] Number of top predictions to which the reference label is compared to.

See also:

Misclassification Provides a more effcient implementation for k = 1.

Notes

Uses *numpy.argsort* to break ties.

is_adversarial (*self*, *predictions*, *label*)
Decides if predictions for an image are adversarial given a reference label.

Parameters

predictions [numpy.ndarray] A vector with the pre-softmax predictions for some image.

label [int] The label of the unperturbed reference image.

Returns

bool True if an image with the given predictions is an adversarial example when the ground-truth class is given by label, False otherwise.

name (*self*)
Returns a human readable name that uniquely identifies the criterion with its hyperparameters.

Returns

str Human readable name that uniquely identifies the criterion with its hyperparameters.

Notes

Defaults to the class name but subclasses can provide more descriptive names and must take hyperparameters into account.

class foolbox.criteria.**TargetClass**(*target_class*)

Defines adversarials as images for which the predicted class is the given target class.

Parameters

target_class [int] The target class that needs to be predicted for an image to be considered an adversarial.

Notes

Uses *numpy.argmax* to break ties.

is_adversarial(*self*, *predictions*, *label*)

Decides if predictions for an image are adversarial given a reference label.

Parameters

predictions [numpy.ndarray] A vector with the pre-softmax predictions for some image.

label [int] The label of the unperturbed reference image.

Returns

bool True if an image with the given predictions is an adversarial example when the ground-truth class is given by label, False otherwise.

name(*self*)

Returns a human readable name that uniquely identifies the criterion with its hyperparameters.

Returns

str Human readable name that uniquely identifies the criterion with its hyperparameters.

Notes

Defaults to the class name but subclasses can provide more descriptive names and must take hyperparameters into account.

class foolbox.criteria.**OriginalClassProbability**(*p*)

Defines adversarials as images for which the probability of the original class is below a given threshold.

This criterion alone does not guarantee that the class predicted for the adversarial image is not the original class (unless p < 1 / number of classes). Therefore, it should usually be combined with a classifcation criterion.

Parameters

p [float] The threshold probability. If the probability of the original class is below this threshold, the image is considered an adversarial. It must satisfy 0 <= p <= 1.

is_adversarial(*self*, *predictions*, *label*)

Decides if predictions for an image are adversarial given a reference label.

Parameters

predictions [numpy.ndarray] A vector with the pre-softmax predictions for some image.

label [int] The label of the unperturbed reference image.

Returns

bool True if an image with the given predictions is an adversarial example when the ground-truth class is given by label, False otherwise.

name (*self*)
Returns a human readable name that uniquely identifies the criterion with its hyperparameters.

Returns

str Human readable name that uniquely identifies the criterion with its hyperparameters.

Notes

Defaults to the class name but subclasses can provide more descriptive names and must take hyperparameters into account.

class foolbox.criteria.**TargetClassProbability** (*target_class*, *p*)
Defines adversarials as images for which the probability of a given target class is above a given threshold.

If the threshold is below 0.5, this criterion does not guarantee that the class predicted for the adversarial image is not the original class. In that case, it should usually be combined with a classification criterion.

Parameters

target_class [int] The target class for which the predicted probability must be above the threshold probability p, otherwise the image is not considered an adversarial.

p [float] The threshold probability. If the probability of the target class is above this threshold, the image is considered an adversarial. It must satisfy 0 <= p <= 1.

is_adversarial (*self*, *predictions*, *label*)
Decides if predictions for an image are adversarial given a reference label.

Parameters

predictions [`numpy.ndarray`] A vector with the pre-softmax predictions for some image.

label [int] The label of the unperturbed reference image.

Returns

bool True if an image with the given predictions is an adversarial example when the ground-truth class is given by label, False otherwise.

name (*self*)
Returns a human readable name that uniquely identifies the criterion with its hyperparameters.

Returns

str Human readable name that uniquely identifies the criterion with its hyperparameters.

Notes

Defaults to the class name but subclasses can provide more descriptive names and must take hyperparameters into account.

1.10 foolbox.zoo

1.10.1 Get Model

foolbox.zoo.**get_model**(*url*)

Provides utilities to download foolbox-compatible robust models to easily test attacks against them by simply providing a git-URL.

Examples

Instantiate a model:

```
>>> from foolbox import zoo
>>> url = "https://github.com/bveliqi/foolbox-zoo-dummy.git"
>>> model = zoo.get_model(url)  # doctest: +SKIP
```

Only works with a foolbox-zoo compatible repository. I.e. models need to have a *foolbox_model.py* file with a *create()*-function, which returns a foolbox-wrapped model.

Example repositories:

- https://github.com/bethgelab/mnist_challenge
- https://github.com/bethgelab/cifar10_challenge
- https://github.com/bethgelab/convex_adversarial

Parameters **url** – URL to the git repository

Returns a foolbox-wrapped model instance

1.10.2 Fetch Weights

foolbox.zoo.**fetch_weights**(*weights_uri*, *unzip=False*)

Provides utilities to download and extract packages containing model weights when creating foolbox-zoo compatible repositories, if the weights are not part of the repository itself.

Examples

Download and unzip weights:

```
>>> from foolbox import zoo
>>> url = 'https://github.com/MadryLab/mnist_challenge_models/raw/master/secret.
↪zip'  # noqa F501
>>> weights_path = zoo.fetch_weights(url, unzip=True)
```

Parameters

- **weights_uri** – the URI to fetch the weights from
- **unzip** – should be *True* if the file to be downloaded is a zipped package

Returns local path where the weights have been downloaded and potentially unzipped to

1.11 foolbox.distances

Provides classes to measure the distance between images.

1.11.1 Distances

MeanSquaredDistance	Calculates the mean squared error between two images.
MeanAbsoluteDistance	Calculates the mean absolute error between two images.
Linfinity	Calculates the L-infinity norm of the difference between two images.
L0	Calculates the L0 norm of the difference between two images.

1.11.2 Aliases

MSE	alias of *foolbox.distances.MeanSquaredDistance*
MAE	alias of *foolbox.distances.MeanAbsoluteDistance*
Linf	alias of *foolbox.distances.Linfinity*

1.11.3 Base class

To implement a new distance, simply subclass the *Distance* class and implement the `_calculate()` method.

Distance	Base class for distances.

1.11.4 Detailed description

class foolbox.distances.**Distance**(*reference=None*, *other=None*, *bounds=None*, *value=None*)

Base class for distances.

This class should be subclassed when implementing new distances. Subclasses must implement _calculate.

class foolbox.distances.**MeanSquaredDistance**(*reference=None*, *other=None*, *bounds=None*, *value=None*)

Calculates the mean squared error between two images.

class foolbox.distances.**MeanAbsoluteDistance**(*reference=None*, *other=None*, *bounds=None*, *value=None*)

Calculates the mean absolute error between two images.

class foolbox.distances.**Linfinity**(*reference=None*, *other=None*, *bounds=None*, *value=None*)

Calculates the L-infinity norm of the difference between two images.

class foolbox.distances.**L0**(*reference=None*, *other=None*, *bounds=None*, *value=None*)

Calculates the L0 norm of the difference between two images.

foolbox.distances.**MSE**

alias of *foolbox.distances.MeanSquaredDistance*

foolbox.distances.**MAE**
alias of *foolbox.distances.MeanAbsoluteDistance*

foolbox.distances.**Linf**
alias of *foolbox.distances.Linfinity*

1.12 foolbox.attacks

1.12.1 Gradient-based attacks

class foolbox.attacks.**GradientAttack**(*model=None*, *criterion=<foolbox.criteria.Misclassification object>*, *distance=<class 'foolbox.distances.MeanSquaredDistance'>*, *threshold=None*)

Perturbs the image with the gradient of the loss w.r.t. the image, gradually increasing the magnitude until the image is misclassified.

Does not do anything if the model does not have a gradient.

__call__(*self*, *input_or_adv*, *label=None*, *unpack=True*, *epsilons=1000*, *max_epsilon=1*)

Perturbs the image with the gradient of the loss w.r.t. the image, gradually increasing the magnitude until the image is misclassified.

Parameters

input_or_adv [*numpy.ndarray* or `Adversarial`] The original, unperturbed input as a *numpy.ndarray* or an `Adversarial` instance.

label [int] The reference label of the original input. Must be passed if *a* is a *numpy.ndarray*, must not be passed if *a* is an `Adversarial` instance.

unpack [bool] If true, returns the adversarial input, otherwise returns the Adversarial object.

epsilons [int or Iterable[float]] Either Iterable of step sizes in the gradient direction or number of step sizes between 0 and max_epsilon that should be tried.

max_epsilon [float] Largest step size if epsilons is not an iterable.

class foolbox.attacks.**GradientSignAttack**(*model=None*, *criterion=<foolbox.criteria.Misclassification object>*, *distance=<class 'foolbox.distances.MeanSquaredDistance'>*, *threshold=None*)

Adds the sign of the gradient to the image, gradually increasing the magnitude until the image is misclassified. This attack is often referred to as Fast Gradient Sign Method and was introduced in [R20d0064ee4c9-1].

Does not do anything if the model does not have a gradient.

References

[R20d0064ee4c9-1]

__call__(*self*, *input_or_adv*, *label=None*, *unpack=True*, *epsilons=1000*, *max_epsilon=1*)

Adds the sign of the gradient to the image, gradually increasing the magnitude until the image is misclassified.

Parameters

input_or_adv [*numpy.ndarray* or `Adversarial`] The original, unperturbed input as a *numpy.ndarray* or an `Adversarial` instance.

label [int] The reference label of the original input. Must be passed if *a* is a *numpy.ndarray*, must not be passed if *a* is an `Adversarial` instance.

unpack [bool] If true, returns the adversarial input, otherwise returns the Adversarial object.

epsilons [int or Iterable[float]] Either Iterable of step sizes in the direction of the sign of the gradient or number of step sizes between 0 and max_epsilon that should be tried.

max_epsilon [float] Largest step size if epsilons is not an iterable.

`foolbox.attacks.`**`FGSM`**
alias of `foolbox.attacks.gradient.GradientSignAttack`

`class` `foolbox.attacks.`**`LinfinityBasicIterativeAttack`**(*model=None*, *criterion=<foolbox.criteria.Misclassification object>*, *distance=<class 'foolbox.distances.MeanSquaredDistance'>*, *threshold=None*)

The Basic Iterative Method introduced in [R37dbc8f24aee-1].

This attack is also known as Projected Gradient Descent (PGD) (without random start) or FGMS^k.

References

See also:

`ProjectedGradientDescentAttack`

[R37dbc8f24aee-1]

`__call__`(*self*, *input_or_adv*, *label=None*, *unpack=True*, *binary_search=True*, *epsilon=0.3*, *stepsize=0.05*, *iterations=10*, *random_start=False*, *return_early=True*)
Simple iterative gradient-based attack known as Basic Iterative Method, Projected Gradient Descent or FGSM^k.

Parameters

input_or_adv [*numpy.ndarray* or `Adversarial`] The original, unperturbed input as a *numpy.ndarray* or an `Adversarial` instance.

label [int] The reference label of the original input. Must be passed if *a* is a *numpy.ndarray*, must not be passed if *a* is an `Adversarial` instance.

unpack [bool] If true, returns the adversarial input, otherwise returns the Adversarial object.

binary_search [bool or int] Whether to perform a binary search over epsilon and stepsize, keeping their ratio constant and using their values to start the search. If False, hyperparameters are not optimized. Can also be an integer, specifying the number of binary search steps (default 20).

epsilon [float] Limit on the perturbation size; if binary_search is True, this value is only for initialization and automatically adapted.

stepsize [float] Step size for gradient descent; if binary_search is True, this value is only for initialization and automatically adapted.

iterations [int] Number of iterations for each gradient descent run.

random_start [bool] Start the attack from a random point rather than from the original input.

return_early [bool] Whether an individual gradient descent run should stop as soon as an adversarial is found.

`foolbox.attacks.`**`BasicIterativeMethod`**
alias of `foolbox.attacks.iterative_projected_gradient.LinfinityBasicIterativeAttack`

`foolbox.attacks.`**`BIM`**
alias of `foolbox.attacks.iterative_projected_gradient.LinfinityBasicIterativeAttack`

`class` `foolbox.attacks.`**`L1BasicIterativeAttack`**(*model=None*, *criterion=<foolbox.criteria.Misclassification object>*, *distance=<class 'foolbox.distances.MeanSquaredDistance'>*, *threshold=None*)

Modified version of the Basic Iterative Method that minimizes the L1 distance.

See also:

LinfinityBasicIterativeAttack

`__call__`(*self*, *input_or_adv*, *label=None*, *unpack=True*, *binary_search=True*, *epsilon=0.3*, *stepsize=0.05*, *iterations=10*, *random_start=False*, *return_early=True*)

Simple iterative gradient-based attack known as Basic Iterative Method, Projected Gradient Descent or FGSM^k.

Parameters

input_or_adv [*numpy.ndarray* or `Adversarial`] The original, unperturbed input as a *numpy.ndarray* or an `Adversarial` instance.

label [int] The reference label of the original input. Must be passed if *a* is a *numpy.ndarray*, must not be passed if *a* is an `Adversarial` instance.

unpack [bool] If true, returns the adversarial input, otherwise returns the Adversarial object.

binary_search [bool or int] Whether to perform a binary search over epsilon and stepsize, keeping their ratio constant and using their values to start the search. If False, hyperparameters are not optimized. Can also be an integer, specifying the number of binary search steps (default 20).

epsilon [float] Limit on the perturbation size; if binary_search is True, this value is only for initialization and automatically adapted.

stepsize [float] Step size for gradient descent; if binary_search is True, this value is only for initialization and automatically adapted.

iterations [int] Number of iterations for each gradient descent run.

random_start [bool] Start the attack from a random point rather than from the original input.

return_early [bool] Whether an individual gradient descent run should stop as soon as an adversarial is found.

`class` `foolbox.attacks.`**`L2BasicIterativeAttack`**(*model=None*, *criterion=<foolbox.criteria.Misclassification object>*, *distance=<class 'foolbox.distances.MeanSquaredDistance'>*, *threshold=None*)

Modified version of the Basic Iterative Method that minimizes the L2 distance.

See also:

LinfinityBasicIterativeAttack

__call__(*self*, *input_or_adv*, *label=None*, *unpack=True*, *binary_search=True*, *epsilon=0.3*, *stepsize=0.05*, *iterations=10*, *random_start=False*, *return_early=True*)

Simple iterative gradient-based attack known as Basic Iterative Method, Projected Gradient Descent or FGSM^k.

Parameters

input_or_adv [*numpy.ndarray* or `Adversarial`] The original, unperturbed input as a *numpy.ndarray* or an `Adversarial` instance.

label [int] The reference label of the original input. Must be passed if *a* is a *numpy.ndarray*, must not be passed if *a* is an `Adversarial` instance.

unpack [bool] If true, returns the adversarial input, otherwise returns the Adversarial object.

binary_search [bool or int] Whether to perform a binary search over epsilon and stepsize, keeping their ratio constant and using their values to start the search. If False, hyperparameters are not optimized. Can also be an integer, specifying the number of binary search steps (default 20).

epsilon [float] Limit on the perturbation size; if binary_search is True, this value is only for initialization and automatically adapted.

stepsize [float] Step size for gradient descent; if binary_search is True, this value is only for initialization and automatically adapted.

iterations [int] Number of iterations for each gradient descent run.

random_start [bool] Start the attack from a random point rather than from the original input.

return_early [bool] Whether an individual gradient descent run should stop as soon as an adversarial is found.

class foolbox.attacks.**ProjectedGradientDescentAttack**(*model=None*, *criterion=<foolbox.criteria.Misclassification object>*, *distance=<class 'foolbox.distances.MeanSquaredDistance'>*, *threshold=None*)

The Projected Gradient Descent Attack introduced in [R367e8e10528a-1] without random start.

When used without a random start, this attack is also known as Basic Iterative Method (BIM) or FGSM^k.

References

See also:

LinfinityBasicIterativeAttack and *RandomStartProjectedGradientDescentAttack*

[R367e8e10528a-1]

__call__(*self*, *input_or_adv*, *label=None*, *unpack=True*, *binary_search=True*, *epsilon=0.3*, *stepsize=0.01*, *iterations=40*, *random_start=False*, *return_early=True*)

Simple iterative gradient-based attack known as Basic Iterative Method, Projected Gradient Descent or FGSM^k.

Parameters

input_or_adv [*numpy.ndarray* or `Adversarial`] The original, unperturbed input as a *numpy.ndarray* or an `Adversarial` instance.

label [int] The reference label of the original input. Must be passed if *a* is a *numpy.ndarray*, must not be passed if *a* is an `Adversarial` instance.

unpack [bool] If true, returns the adversarial input, otherwise returns the Adversarial object.

binary_search [bool or int] Whether to perform a binary search over epsilon and stepsize, keeping their ratio constant and using their values to start the search. If False, hyperparameters are not optimized. Can also be an integer, specifying the number of binary search steps (default 20).

epsilon [float] Limit on the perturbation size; if binary_search is True, this value is only for initialization and automatically adapted.

stepsize [float] Step size for gradient descent; if binary_search is True, this value is only for initialization and automatically adapted.

iterations [int] Number of iterations for each gradient descent run.

random_start [bool] Start the attack from a random point rather than from the original input.

return_early [bool] Whether an individual gradient descent run should stop as soon as an adversarial is found.

`foolbox.attacks.`**`ProjectedGradientDescent`**

alias of `foolbox.attacks.iterative_projected_gradient.ProjectedGradientDescentAttack`

class `foolbox.attacks.`**`RandomStartProjectedGradientDescentAttack`**(*model=None, criterion=<foolbox.criteria.Misclassification object>, distance=<class 'foolbox.distances.MeanSquaredDistance'>, threshold=None*)

The Projected Gradient Descent Attack introduced in [Re6066bc39e14-1] with random start.

References

See also:

`ProjectedGradientDescentAttack`

[Re6066bc39e14-1]

`__call__`(*self, input_or_adv, label=None, unpack=True, binary_search=True, epsilon=0.3, stepsize=0.01, iterations=40, random_start=True, return_early=True*)

Simple iterative gradient-based attack known as Basic Iterative Method, Projected Gradient Descent or FGSM^k.

Parameters

input_or_adv [*numpy.ndarray* or `Adversarial`] The original, unperturbed input as a *numpy.ndarray* or an `Adversarial` instance.

label [int] The reference label of the original input. Must be passed if *a* is a *numpy.ndarray*, must not be passed if *a* is an `Adversarial` instance.

unpack [bool] If true, returns the adversarial input, otherwise returns the Adversarial object.

binary_search [bool or int] Whether to perform a binary search over epsilon and stepsize, keeping their ratio constant and using their values to start the search. If False, hyperparameters are not optimized. Can also be an integer, specifying the number of binary search steps (default 20).

epsilon [float] Limit on the perturbation size; if binary_search is True, this value is only for initialization and automatically adapted.

stepsize [float] Step size for gradient descent; if binary_search is True, this value is only for initialization and automatically adapted.

iterations [int] Number of iterations for each gradient descent run.

random_start [bool] Start the attack from a random point rather than from the original input.

return_early [bool] Whether an individual gradient descent run should stop as soon as an adversarial is found.

`foolbox.attacks.`**`RandomProjectedGradientDescent`**

alias of `foolbox.attacks.iterative_projected_gradient.RandomStartProjectedGradientDescentAttack`

`foolbox.attacks.`**`RandomPGD`**

alias of `foolbox.attacks.iterative_projected_gradient.RandomStartProjectedGradientDescentAttack`

`class` `foolbox.attacks.`**`MomentumIterativeAttack`**(*model=None, criterion=<foolbox.criteria.Misclassification object>, distance=<class 'foolbox.distances.MeanSquaredDistance'>, threshold=None*)

The Momentum Iterative Method attack introduced in [R86d363e1fb2f-1]. It's like the Basic Iterative Method or Projected Gradient Descent except that it uses momentum.

References

[R86d363e1fb2f-1]

`__call__`(*self, input_or_adv, label=None, unpack=True, binary_search=True, epsilon=0.3, stepsize=0.06, iterations=10, decay_factor=1.0, random_start=False, return_early=True*)

Momentum-based iterative gradient attack known as Momentum Iterative Method.

Parameters

input_or_adv [*numpy.ndarray* or `Adversarial`] The original, unperturbed input as a *numpy.ndarray* or an `Adversarial` instance.

label [int] The reference label of the original input. Must be passed if *a* is a *numpy.ndarray*, must not be passed if *a* is an `Adversarial` instance.

unpack [bool] If true, returns the adversarial input, otherwise returns the Adversarial object.

binary_search [bool] Whether to perform a binary search over epsilon and stepsize, keeping their ratio constant and using their values to start the search. If False, hyperparameters are not optimized. Can also be an integer, specifying the number of binary search steps (default 20).

epsilon [float] Limit on the perturbation size; if binary_search is True, this value is only for initialization and automatically adapted.

stepsize [float] Step size for gradient descent; if binary_search is True, this value is only for initialization and automatically adapted.

iterations [int] Number of iterations for each gradient descent run.

decay_factor [float] Decay factor used by the momentum term.

random_start [bool] Start the attack from a random point rather than from the original input.

return_early [bool] Whether an individual gradient descent run should stop as soon as an adversarial is found.

`foolbox.attacks.`**`MomentumIterativeMethod`**

alias of `foolbox.attacks.iterative_projected_gradient.MomentumIterativeAttack`

class `foolbox.attacks.`**`LBFGSAttack`**(**args*, ***kwargs*)

Uses L-BFGS-B to minimize the distance between the image and the adversarial as well as the cross-entropy between the predictions for the adversarial and the the one-hot encoded target class.

If the criterion does not have a target class, a random class is chosen from the set of all classes except the original one.

Notes

This implementation generalizes algorithm 1 in [Rf3ff9c7ff5d3-1] to support other targeted criteria and other distance measures.

References

[Rf3ff9c7ff5d3-1]

`__call__`(*self*, *input_or_adv*, *label=None*, *unpack=True*, *epsilon=1e-05*, *num_random_targets=0*, *maxiter=150*)

Uses L-BFGS-B to minimize the distance between the image and the adversarial as well as the cross-entropy between the predictions for the adversarial and the the one-hot encoded target class.

Parameters

input_or_adv [*numpy.ndarray* or `Adversarial`] The original, unperturbed input as a *numpy.ndarray* or an `Adversarial` instance.

label [int] The reference label of the original input. Must be passed if *a* is a *numpy.ndarray*, must not be passed if *a* is an `Adversarial` instance.

unpack [bool] If true, returns the adversarial input, otherwise returns the Adversarial object.

epsilon [float] Epsilon of the binary search.

num_random_targets [int] Number of random target classes if no target class is given by the criterion.

maxiter [int] Maximum number of iterations for L-BFGS-B.

`__init__`(*self*, **args*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

`name`(*self*)

Returns a human readable name that uniquely identifies the attack with its hyperparameters.

Returns

str Human readable name that uniquely identifies the attack with its hyperparameters.

Notes

Defaults to the class name but subclasses can provide more descriptive names and must take hyperparameters into account.

class foolbox.attacks.**DeepFoolAttack**(*model=None*, *criterion=<foolbox.criteria.Misclassification object>*, *distance=<class 'foolbox.distances.MeanSquaredDistance'>*, *threshold=None*)

Simple and close to optimal gradient-based adversarial attack.

Implementes DeepFool introduced in [Rb4dd02640756-1].

References

[Rb4dd02640756-1]

__call__(*self*, *input_or_adv*, *label=None*, *unpack=True*, *steps=100*, *subsample=10*, *p=None*)

Simple and close to optimal gradient-based adversarial attack.

Parameters

input_or_adv [*numpy.ndarray* or `Adversarial`] The original, unperturbed input as a *numpy.ndarray* or an `Adversarial` instance.

label [int] The reference label of the original input. Must be passed if *a* is a *numpy.ndarray*, must not be passed if *a* is an `Adversarial` instance.

unpack [bool] If true, returns the adversarial input, otherwise returns the Adversarial object.

steps [int] Maximum number of steps to perform.

subsample [int] Limit on the number of the most likely classes that should be considered. A small value is usually sufficient and much faster.

p [int or float] Lp-norm that should be minimzed, must be 2 or np.inf.

class foolbox.attacks.**NewtonFoolAttack**(*model=None*, *criterion=<foolbox.criteria.Misclassification object>*, *distance=<class 'foolbox.distances.MeanSquaredDistance'>*, *threshold=None*)

Implements the NewtonFool Attack.

The attack was introduced in [R6a972939b320-1].

References

[R6a972939b320-1]

__call__(*self*, *input_or_adv*, *label=None*, *unpack=True*, *max_iter=100*, *eta=0.01*)

Parameters

input_or_adv [*numpy.ndarray* or `Adversarial`] The original, unperturbed input as a *numpy.ndarray* or an `Adversarial` instance.

label [int] The reference label of the original input. Must be passed if *a* is a *numpy.ndarray*, must not be passed if *a* is an `Adversarial` instance.

unpack [bool] If true, returns the adversarial input, otherwise returns the Adversarial object.

max_iter [int] The maximum number of iterations.

eta [float] the eta coefficient

class foolbox.attacks.**DeepFoolL2Attack**(*model=None*, *criterion=<foolbox.criteria.Misclassification object>*, *distance=<class 'foolbox.distances.MeanSquaredDistance'>*, *threshold=None*)

__call__(*self*, *input_or_adv*, *label=None*, *unpack=True*, *steps=100*, *subsample=10*)
Simple and close to optimal gradient-based adversarial attack.

Parameters

input_or_adv [*numpy.ndarray* or `Adversarial`] The original, unperturbed input as a *numpy.ndarray* or an `Adversarial` instance.

label [int] The reference label of the original input. Must be passed if *a* is a *numpy.ndarray*, must not be passed if *a* is an `Adversarial` instance.

unpack [bool] If true, returns the adversarial input, otherwise returns the Adversarial object.

steps [int] Maximum number of steps to perform.

subsample [int] Limit on the number of the most likely classes that should be considered. A small value is usually sufficient and much faster.

p [int or float] Lp-norm that should be minimzed, must be 2 or np.inf.

class foolbox.attacks.**DeepFoolLinfinityAttack**(*model=None*, *criterion=<foolbox.criteria.Misclassification object>*, *distance=<class 'foolbox.distances.MeanSquaredDistance'>*, *threshold=None*)

__call__(*self*, *input_or_adv*, *label=None*, *unpack=True*, *steps=100*, *subsample=10*)
Simple and close to optimal gradient-based adversarial attack.

Parameters

input_or_adv [*numpy.ndarray* or `Adversarial`] The original, unperturbed input as a *numpy.ndarray* or an `Adversarial` instance.

label [int] The reference label of the original input. Must be passed if *a* is a *numpy.ndarray*, must not be passed if *a* is an `Adversarial` instance.

unpack [bool] If true, returns the adversarial input, otherwise returns the Adversarial object.

steps [int] Maximum number of steps to perform.

subsample [int] Limit on the number of the most likely classes that should be considered. A small value is usually sufficient and much faster.

p [int or float] Lp-norm that should be minimzed, must be 2 or np.inf.

class foolbox.attacks.**ADefAttack**(*model=None*, *criterion=<foolbox.criteria.Misclassification object>*, *distance=<class 'foolbox.distances.MeanSquaredDistance'>*, *threshold=None*)
Adversarial attack that distorts the image, i.e. changes the locations of pixels. The algorithm is described

in [Rf241e6d2664d-1], a Repository with the original code can be found in [Rf241e6d2664d-2]. References ——— .. [Rf241e6d2664d-1] Rima Alaifari, Giovanni S. Alberti, and Tandri Gauksson:

"ADef: an Iterative Algorithm to Construct Adversarial Deformations", https://arxiv.org/abs/1804.07729

__call__(*self*, *input_or_adv*, *unpack=True*, *max_iter=100*, *max_norm=<Mock name='mock.inf' id='140580051901352'>*, *label=None*, *smooth=1.0*, *subsample=10*)

Parameters

input_or_adv [*numpy.ndarray* or `Adversarial`] The original, unperturbed input as a *numpy.ndarray* or an `Adversarial` instance.

label [int] The reference label of the original input. Must be passed if *a* is a *numpy.ndarray*, must not be passed if *a* is an `Adversarial` instance.

unpack [bool] If true, returns the adversarial input, otherwise returns the Adversarial object.

max_iter [int > 0] Maximum number of iterations (default max_iter = 100).

max_norm [float] Maximum l2 norm of vector field (default max_norm = numpy.inf).

smooth [float >= 0] Width of the Gaussian kernel used for smoothing. (default is smooth = 0 for no smoothing).

subsample [int >= 2] Limit on the number of the most likely classes that should be considered. A small value is usually sufficient and much faster. (default subsample = 10)

class foolbox.attacks.**SLSQPAttack**(*model=None*, *criterion=<foolbox.criteria.Misclassification object>*, *distance=<class 'foolbox.distances.MeanSquaredDistance'>*, *threshold=None*)

Uses SLSQP to minimize the distance between the image and the adversarial under the constraint that the image is adversarial.

__call__(*self*, *input_or_adv*, *label=None*, *unpack=True*)

Uses SLSQP to minimize the distance between the image and the adversarial under the constraint that the image is adversarial.

Parameters

input_or_adv [*numpy.ndarray* or `Adversarial`] The original, correctly classified image. If image is a numpy array, label must be passed as well. If image is an `Adversarial` instance, label must not be passed.

label [int] The reference label of the original image. Must be passed if image is a numpy array, must not be passed if image is an `Adversarial` instance.

unpack [bool] If true, returns the adversarial image, otherwise returns the Adversarial object.

class foolbox.attacks.**SaliencyMapAttack**(*model=None*, *criterion=<foolbox.criteria.Misclassification object>*, *distance=<class 'foolbox.distances.MeanSquaredDistance'>*, *threshold=None*)

Implements the Saliency Map Attack.

The attack was introduced in [R08e06ca693ba-1].

References

[R08e06ca693ba-1]

__call__(*self*, *input_or_adv*, *label=None*, *unpack=True*, *max_iter=2000*, *num_random_targets=0*, *fast=True*, *theta=0.1*, *max_perturbations_per_pixel=7*)

Implements the Saliency Map Attack.

Parameters

input_or_adv [*numpy.ndarray* or `Adversarial`] The original, unperturbed input as a *numpy.ndarray* or an `Adversarial` instance.

label [int] The reference label of the original input. Must be passed if *a* is a *numpy.ndarray*, must not be passed if *a* is an `Adversarial` instance.

max_iter [int] The maximum number of iterations to run.

num_random_targets [int] Number of random target classes if no target class is given by the criterion.

fast [bool] Whether to use the fast saliency map calculation.

theta [float] perturbation per pixel relative to [min, max] range.

max_perturbations_per_pixel [int] Maximum number of times a pixel can be modified.

class foolbox.attacks.**IterativeGradientAttack**(*model=None*, *criterion=<foolbox.criteria.Misclassification object>*, *distance=<class 'foolbox.distances.MeanSquaredDistance'>*, *threshold=None*)

Like GradientAttack but with several steps for each epsilon.

__call__(*self*, *input_or_adv*, *label=None*, *unpack=True*, *epsilons=100*, *max_epsilon=1*, *steps=10*)

Like GradientAttack but with several steps for each epsilon.

Parameters

input_or_adv [*numpy.ndarray* or `Adversarial`] The original, unperturbed input as a *numpy.ndarray* or an `Adversarial` instance.

label [int] The reference label of the original input. Must be passed if *a* is a *numpy.ndarray*, must not be passed if *a* is an `Adversarial` instance.

unpack [bool] If true, returns the adversarial input, otherwise returns the Adversarial object.

epsilons [int or Iterable[float]] Either Iterable of step sizes in the gradient direction or number of step sizes between 0 and max_epsilon that should be tried.

max_epsilon [float] Largest step size if epsilons is not an iterable.

steps [int] Number of iterations to run.

class foolbox.attacks.**IterativeGradientSignAttack**(*model=None*, *criterion=<foolbox.criteria.Misclassification object>*, *distance=<class 'foolbox.distances.MeanSquaredDistance'>*, *threshold=None*)

Like GradientSignAttack but with several steps for each epsilon.

__call__(*self*, *input_or_adv*, *label=None*, *unpack=True*, *epsilons=100*, *max_epsilon=1*, *steps=10*)

Like GradientSignAttack but with several steps for each epsilon.

Parameters

input_or_adv [*numpy.ndarray* or `Adversarial`] The original, unperturbed input as a *numpy.ndarray* or an `Adversarial` instance.

label [int] The reference label of the original input. Must be passed if *a* is a *numpy.ndarray*, must not be passed if *a* is an `Adversarial` instance.

unpack [bool] If true, returns the adversarial input, otherwise returns the Adversarial object.

epsilons [int or Iterable[float]] Either Iterable of step sizes in the direction of the sign of the gradient or number of step sizes between 0 and max_epsilon that should be tried.

max_epsilon [float] Largest step size if epsilons is not an iterable.

steps [int] Number of iterations to run.

class foolbox.attacks.**CarliniWagnerL2Attack**(*model=None*, *criterion=<foolbox.criteria.Misclassification object>*, *distance=<class 'foolbox.distances.MeanSquaredDistance'>*, *threshold=None*)

The L2 version of the Carlini & Wagner attack.

This attack is described in [Rc2cb572b91c5-1]. This implementation is based on the reference implementation by Carlini [Rc2cb572b91c5-2]. For bounds (0, 1), it differs from [Rc2cb572b91c5-2] because we normalize the squared L2 loss with the bounds.

References

[Rc2cb572b91c5-1], [Rc2cb572b91c5-2]

__call__(*self*, *input_or_adv*, *label=None*, *unpack=True*, *binary_search_steps=5*, *max_iterations=1000*, *confidence=0*, *learning_rate=0.005*, *initial_const=0.01*, *abort_early=True*)

The L2 version of the Carlini & Wagner attack.

Parameters

input_or_adv [*numpy.ndarray* or `Adversarial`] The original, unperturbed input as a *numpy.ndarray* or an `Adversarial` instance.

label [int] The reference label of the original input. Must be passed if *a* is a *numpy.ndarray*, must not be passed if *a* is an `Adversarial` instance.

unpack [bool] If true, returns the adversarial input, otherwise returns the Adversarial object.

binary_search_steps [int] The number of steps for the binary search used to find the optimal tradeoff-constant between distance and confidence.

max_iterations [int] The maximum number of iterations. Larger values are more accurate; setting it too small will require a large learning rate and will produce poor results.

confidence [int or float] Confidence of adversarial examples: a higher value produces adversarials that are further away, but more strongly classified as adversarial.

learning_rate [float] The learning rate for the attack algorithm. Smaller values produce better results but take longer to converge.

initial_const [float] The initial tradeoff-constant to use to tune the relative importance of distance and confidence. If *binary_search_steps* is large, the initial constant is not important.

abort_early [bool] If True, Adam will be aborted if the loss hasn't decreased for some time (a tenth of max_iterations).

static best_other_class(*logits*, *exclude*)

Returns the index of the largest logit, ignoring the class that is passed as *exclude*.

classmethod loss_function(*const*, *a*, *x*, *logits*, *reconstructed_original*, *confidence*, *min_*, *max_*)
Returns the loss and the gradient of the loss w.r.t. x, assuming that logits = model(x).

1.12.2 Score-based attacks

class foolbox.attacks.**SinglePixelAttack**(*model=None*, *criterion=<foolbox.criteria.Misclassification object>*, *distance=<class 'foolbox.distances.MeanSquaredDistance'>*, *threshold=None*)
Perturbs just a single pixel and sets it to the min or max.

__call__(*self*, *input_or_adv*, *label=None*, *unpack=True*, *max_pixels=1000*)
Perturbs just a single pixel and sets it to the min or max.

Parameters

input_or_adv [*numpy.ndarray* or `Adversarial`] The original, correctly classified image. If image is a numpy array, label must be passed as well. If image is an `Adversarial` instance, label must not be passed.

label [int] The reference label of the original image. Must be passed if image is a numpy array, must not be passed if image is an `Adversarial` instance.

unpack [bool] If true, returns the adversarial image, otherwise returns the Adversarial object.

max_pixels [int] Maximum number of pixels to try.

class foolbox.attacks.**LocalSearchAttack**(*model=None*, *criterion=<foolbox.criteria.Misclassification object>*, *distance=<class 'foolbox.distances.MeanSquaredDistance'>*, *threshold=None*)
A black-box attack based on the idea of greedy local search.

This implementation is based on the algorithm in [Rb320cee6998a-1].

References

[Rb320cee6998a-1]

__call__(*self*, *input_or_adv*, *label=None*, *unpack=True*, *r=1.5*, *p=10.0*, *d=5*, *t=5*, *R=150*)
A black-box attack based on the idea of greedy local search.

Parameters

input_or_adv [*numpy.ndarray* or `Adversarial`] The original, correctly classified image. If image is a numpy array, label must be passed as well. If image is an `Adversarial` instance, label must not be passed.

label [int] The reference label of the original image. Must be passed if image is a numpy array, must not be passed if image is an `Adversarial` instance.

unpack [bool] If true, returns the adversarial image, otherwise returns the Adversarial object.

r [float] Perturbation parameter that controls the cyclic perturbation; must be in [0, 2]

p [float] Perturbation parameter that controls the pixel sensitivity estimation

d [int] The half side length of the neighborhood square

t [int] The number of pixels perturbed at each round

R [int] An upper bound on the number of iterations

class foolbox.attacks.**ApproximateLBFGSAttack**(**args, **kwargs*)

Same as *LBFGSAttack* with approximate_gradient set to True.

__init__(*self, *args, **kwargs*)

Initialize self. See help(type(self)) for accurate signature.

1.12.3 Decision-based attacks

class foolbox.attacks.**BoundaryAttack**(*model=None, criterion=<foolbox.criteria.Misclassification object>, distance=<class 'foolbox.distances.MeanSquaredDistance'>, threshold=None*)

A powerful adversarial attack that requires neither gradients nor probabilities.

This is the reference implementation for the attack introduced in [Re72ca268aa55-1].

Notes

This implementation provides several advanced features:

- ability to continue previous attacks by passing an instance of the Adversarial class
- ability to pass an explicit starting point; especially to initialize a targeted attack
- ability to pass an alternative attack used for initialization
- fine-grained control over logging
- ability to specify the batch size
- optional automatic batch size tuning
- optional multithreading for random number generation
- optional multithreading for candidate point generation

References

[Re72ca268aa55-1]

__call__(*self, input_or_adv, label=None, unpack=True, iterations=5000, max_directions=25, starting_point=None, initialization_attack=None, log_every_n_steps=1, spherical_step=0.01, source_step=0.01, step_adaptation=1.5, batch_size=1, tune_batch_size=True, threaded_rnd=True, threaded_gen=True, alternative_generator=False, internal_dtype=<Mock name='mock.float64' id='140579968988608'>, verbose=False*)

Applies the Boundary Attack.

Parameters

input_or_adv [*numpy.ndarray* or `Adversarial`] The original, correctly classified image. If image is a numpy array, label must be passed as well. If image is an `Adversarial` instance, label must not be passed.

label [int] The reference label of the original image. Must be passed if image is a numpy array, must not be passed if image is an `Adversarial` instance.

unpack [bool] If true, returns the adversarial image, otherwise returns the Adversarial object.

iterations [int] Maximum number of iterations to run. Might converge and stop before that.

max_directions [int] Maximum number of trials per ieration.

starting_point [*numpy.ndarray*] Adversarial input to use as a starting point, in particular for targeted attacks.

initialization_attack [`Attack`] Attack to use to find a starting point. Defaults to BlendedUniformNoiseAttack.

log_every_n_steps [int] Determines verbositity of the logging.

spherical_step [float] Initial step size for the orthogonal (spherical) step.

source_step [float] Initial step size for the step towards the target.

step_adaptation [float] Factor by which the step sizes are multiplied or divided.

batch_size [int] Batch size or initial batch size if tune_batch_size is True

tune_batch_size [bool] Whether or not the batch size should be automatically chosen between 1 and max_directions.

threaded_rnd [bool] Whether the random number generation should be multithreaded.

threaded_gen [bool] Whether the candidate point generation should be multithreaded.

alternative_generator: bool Whether an alternative implemenation of the candidate generator should be used.

internal_dtype [np.float32 or np.float64] Higher precision might be slower but is numerically more stable.

verbose [bool] Controls verbosity of the attack.

class foolbox.attacks.**SpatialAttack**(*model=None*, *criterion=<foolbox.criteria.Misclassification object>*, *distance=<class 'foolbox.distances.MeanSquaredDistance'>*, *threshold=None*)

Adversarially chosen rotations and translations [1].

This implementation is based on the reference implementation by Madry et al.: https://github.com/MadryLab/adversarial_spatial

References

[Rdffd25498f9d-1]

__call__(*self*, *input_or_adv*, *label=None*, *unpack=True*, *do_rotations=True*, *do_translations=True*, *x_shift_limits=(-5, 5)*, *y_shift_limits=(-5, 5)*, *angular_limits=(-5, 5)*, *granularity=10*, *random_sampling=False*, *abort_early=True*)

Adversarially chosen rotations and translations.

Parameters

input_or_adv [*numpy.ndarray* or `Adversarial`] The original, unperturbed input as a *numpy.ndarray* or an `Adversarial` instance.

label [int] The reference label of the original input. Must be passed if *a* is a *numpy.ndarray*, must not be passed if *a* is an `Adversarial` instance.

unpack [bool] If true, returns the adversarial input, otherwise returns the Adversarial object.

do_rotations [bool] If False no rotations will be applied to the image.

do_translations [bool] If False no translations will be applied to the image.

x_shift_limits [int or (int, int)] Limits for horizontal translations in pixels. If one integer is provided the limits will be (-x_shift_limits, x_shift_limits).

y_shift_limits [int or (int, int)] Limits for vertical translations in pixels. If one integer is provided the limits will be (-y_shift_limits, y_shift_limits).

angular_limits [int or (int, int)] Limits for rotations in degrees. If one integer is provided the limits will be [-angular_limits, angular_limits].

granularity [int] Density of sampling within limits for each dimension.

random_sampling [bool] If True we sample translations/rotations randomly within limits, otherwise we use a regular grid.

abort_early [bool] If True, the attack stops as soon as it finds an adversarial.

class foolbox.attacks.**PointwiseAttack**(*model=None, criterion=<foolbox.criteria.Misclassification object>, distance=<class 'foolbox.distances.MeanSquaredDistance'>, threshold=None*)

Starts with an adversarial and performs a binary search between the adversarial and the original for each dimension of the input individually.

__call__(*self, input_or_adv, label=None, unpack=True, starting_point=None, initialization_attack=None*)

Starts with an adversarial and performs a binary search between the adversarial and the original for each dimension of the input individually.

Parameters

input_or_adv [*numpy.ndarray* or `Adversarial`] The original, unperturbed input as a *numpy.ndarray* or an `Adversarial` instance.

label [int] The reference label of the original input. Must be passed if *a* is a *numpy.ndarray*, must not be passed if *a* is an `Adversarial` instance.

unpack [bool] If true, returns the adversarial input, otherwise returns the Adversarial object.

starting_point [*numpy.ndarray*] Adversarial input to use as a starting point, in particular for targeted attacks.

initialization_attack [`Attack`] Attack to use to find a starting point. Defaults to SaltAndPepperNoiseAttack.

class foolbox.attacks.**GaussianBlurAttack**(*model=None, criterion=<foolbox.criteria.Misclassification object>, distance=<class 'foolbox.distances.MeanSquaredDistance'>, threshold=None*)

Blurs the image until it is misclassified.

__call__(*self, input_or_adv, label=None, unpack=True, epsilons=1000*)

Blurs the image until it is misclassified.

Parameters

input_or_adv [*numpy.ndarray* or `Adversarial`] The original, unperturbed input as a *numpy.ndarray* or an `Adversarial` instance.

label [int] The reference label of the original input. Must be passed if *a* is a *numpy.ndarray*, must not be passed if *a* is an `Adversarial` instance.

unpack [bool] If true, returns the adversarial input, otherwise returns the Adversarial object.

epsilons [int or Iterable[float]] Either Iterable of standard deviations of the Gaussian blur or number of standard deviations between 0 and 1 that should be tried.

class foolbox.attacks.**ContrastReductionAttack**(*model=None*, *criterion=<foolbox.criteria.Misclassification object>*, *distance=<class 'foolbox.distances.MeanSquaredDistance'>*, *threshold=None*)

Reduces the contrast of the image until it is misclassified.

__call__(*self*, *input_or_adv*, *label=None*, *unpack=True*, *epsilons=1000*)

Reduces the contrast of the image until it is misclassified.

Parameters

input_or_adv [*numpy.ndarray* or `Adversarial`] The original, unperturbed input as a *numpy.ndarray* or an `Adversarial` instance.

label [int] The reference label of the original input. Must be passed if *a* is a *numpy.ndarray*, must not be passed if *a* is an `Adversarial` instance.

unpack [bool] If true, returns the adversarial input, otherwise returns the Adversarial object.

epsilons [int or Iterable[float]] Either Iterable of contrast levels or number of contrast levels between 1 and 0 that should be tried. Epsilons are one minus the contrast level.

class foolbox.attacks.**AdditiveUniformNoiseAttack**(*model=None*, *criterion=<foolbox.criteria.Misclassification object>*, *distance=<class 'foolbox.distances.MeanSquaredDistance'>*, *threshold=None*)

Adds uniform noise to the image, gradually increasing the standard deviation until the image is misclassified.

__call__(*self*, *input_or_adv*, *label=None*, *unpack=True*, *epsilons=1000*)

Adds uniform or Gaussian noise to the image, gradually increasing the standard deviation until the image is misclassified.

Parameters

input_or_adv [*numpy.ndarray* or `Adversarial`] The original, unperturbed input as a *numpy.ndarray* or an `Adversarial` instance.

label [int] The reference label of the original input. Must be passed if *a* is a *numpy.ndarray*, must not be passed if *a* is an `Adversarial` instance.

unpack [bool] If true, returns the adversarial input, otherwise returns the Adversarial object.

epsilons [int or Iterable[float]] Either Iterable of noise levels or number of noise levels between 0 and 1 that should be tried.

__class__

alias of `abc.ABCMeta`

__delattr__(*self*, *name*, /)

Implement delattr(self, name).

__dir__()

default dir() implementation

__eq__(*self*, *value*, */*)
Return self==value.

__format__()
default object formatter

__ge__(*self*, *value*, */*)
Return self>=value.

__getattribute__(*self*, *name*, */*)
Return getattr(self, name).

__gt__(*self*, *value*, */*)
Return self>value.

__hash__(*self*, */*)
Return hash(self).

__init__(*self*, *model=None*, *criterion=<foolbox.criteria.Misclassification object at 0x7fdb53218908>*, *distance=<class 'foolbox.distances.MeanSquaredDistance'>*, *threshold=None*)
Initialize self. See help(type(self)) for accurate signature.

__le__(*self*, *value*, */*)
Return self<=value.

__lt__(*self*, *value*, */*)
Return self<value.

__ne__(*self*, *value*, */*)
Return self!=value.

__new__(**args*, ***kwargs*)
Create and return a new object. See help(type) for accurate signature.

__reduce__()
helper for pickle

__reduce_ex__()
helper for pickle

__repr__(*self*, */*)
Return repr(self).

__setattr__(*self*, *name*, *value*, */*)
Implement setattr(self, name, value).

__sizeof__()
size of object in memory, in bytes

__str__(*self*, */*)
Return str(self).

__subclasshook__()
Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

__weakref__
list of weak references to the object (if defined)

name (*self*)
Returns a human readable name that uniquely identifies the attack with its hyperparameters.

Returns

str Human readable name that uniquely identifies the attack with its hyperparameters.

Notes

Defaults to the class name but subclasses can provide more descriptive names and must take hyperparameters into account.

class foolbox.attacks.**AdditiveGaussianNoiseAttack**(*model=None, criterion=<foolbox.criteria.Misclassification object>, distance=<class 'foolbox.distances.MeanSquaredDistance'>, threshold=None*)
Adds Gaussian noise to the image, gradually increasing the standard deviation until the image is misclassified.

__call__(*self, input_or_adv, label=None, unpack=True, epsilons=1000*)
Adds uniform or Gaussian noise to the image, gradually increasing the standard deviation until the image is misclassified.

Parameters

input_or_adv [*numpy.ndarray* or `Adversarial`] The original, unperturbed input as a *numpy.ndarray* or an `Adversarial` instance.

label [int] The reference label of the original input. Must be passed if *a* is a *numpy.ndarray*, must not be passed if *a* is an `Adversarial` instance.

unpack [bool] If true, returns the adversarial input, otherwise returns the Adversarial object.

epsilons [int or Iterable[float]] Either Iterable of noise levels or number of noise levels between 0 and 1 that should be tried.

__class__
alias of `abc.ABCMeta`

__delattr__(*self, name, /*)
Implement delattr(self, name).

__dir__()
default dir() implementation

__eq__(*self, value, /*)
Return self==value.

__format__()
default object formatter

__ge__(*self, value, /*)
Return self>=value.

__getattribute__(*self, name, /*)
Return getattr(self, name).

__gt__(*self, value, /*)
Return self>value.

__hash__(*self, /*)
Return hash(self).

__init__(*self*, *model=None*, *criterion=<foolbox.criteria.Misclassification object at 0x7fdb53218908>*, *distance=<class 'foolbox.distances.MeanSquaredDistance'>*, *threshold=None*)
Initialize self. See help(type(self)) for accurate signature.

__le__(*self*, *value*, /)
Return self<=value.

__lt__(*self*, *value*, /)
Return self<value.

__ne__(*self*, *value*, /)
Return self!=value.

__new__(**args*, ***kwargs*)
Create and return a new object. See help(type) for accurate signature.

__reduce__()
helper for pickle

__reduce_ex__()
helper for pickle

__repr__(*self*, /)
Return repr(self).

__setattr__(*self*, *name*, *value*, /)
Implement setattr(self, name, value).

__sizeof__()
size of object in memory, in bytes

__str__(*self*, /)
Return str(self).

__subclasshook__()
Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

__weakref__
list of weak references to the object (if defined)

name(*self*)
Returns a human readable name that uniquely identifies the attack with its hyperparameters.

Returns

str Human readable name that uniquely identifies the attack with its hyperparameters.

Notes

Defaults to the class name but subclasses can provide more descriptive names and must take hyperparameters into account.

class foolbox.attacks.**SaltAndPepperNoiseAttack**(*model=None*, *criterion=<foolbox.criteria.Misclassification object>*, *distance=<class 'foolbox.distances.MeanSquaredDistance'>*, *threshold=None*)

Increases the amount of salt and pepper noise until the image is misclassified.

`__call__`(*self*, *input_or_adv*, *label=None*, *unpack=True*, *epsilons=100*, *repetitions=10*)
Increases the amount of salt and pepper noise until the image is misclassified.

Parameters

input_or_adv [*numpy.ndarray* or `Adversarial`] The original, unperturbed input as a *numpy.ndarray* or an `Adversarial` instance.

label [int] The reference label of the original input. Must be passed if *a* is a *numpy.ndarray*, must not be passed if *a* is an `Adversarial` instance.

unpack [bool] If true, returns the adversarial input, otherwise returns the Adversarial object.

epsilons [int] Number of steps to try between probability 0 and 1.

repetitions [int] Specifies how often the attack will be repeated.

class `foolbox.attacks.`**`BlendedUniformNoiseAttack`**(*model=None*, *criterion=<foolbox.criteria.Misclassification object>*, *distance=<class 'foolbox.distances.MeanSquaredDistance'>*, *threshold=None*)
Blends the image with a uniform noise image until it is misclassified.

`__call__`(*self*, *input_or_adv*, *label=None*, *unpack=True*, *epsilons=1000*, *max_directions=1000*)
Blends the image with a uniform noise image until it is misclassified.

Parameters

input_or_adv [*numpy.ndarray* or `Adversarial`] The original, unperturbed input as a *numpy.ndarray* or an `Adversarial` instance.

label [int] The reference label of the original input. Must be passed if *a* is a *numpy.ndarray*, must not be passed if *a* is an `Adversarial` instance.

unpack [bool] If true, returns the adversarial input, otherwise returns the Adversarial object.

epsilons [int or Iterable[float]] Either Iterable of blending steps or number of blending steps between 0 and 1 that should be tried.

max_directions [int] Maximum number of random images to try.

1.12.4 Other attacks

class `foolbox.attacks.`**`BinarizationRefinementAttack`**(*model=None*, *criterion=<foolbox.criteria.Misclassification object>*, *distance=<class 'foolbox.distances.MeanSquaredDistance'>*, *threshold=None*)
For models that preprocess their inputs by binarizing the inputs, this attack can improve adversarials found by other attacks. It does os by utilizing information about the binarization and mapping values to the corresponding value in the clean input or to the right side of the threshold.

`__call__`(*self*, *input_or_adv*, *label=None*, *unpack=True*, *starting_point=None*, *threshold=None*, *included_in='upper'*)
For models that preprocess their inputs by binarizing the inputs, this attack can improve adversarials found by other attacks. It does os by utilizing information about the binarization and mapping values to the corresponding value in the clean input or to the right side of the threshold.

Parameters

input_or_adv [*numpy.ndarray* or `Adversarial`] The original, unperturbed input as a *numpy.ndarray* or an `Adversarial` instance.

label [int] The reference label of the original input. Must be passed if *a* is a *numpy.ndarray*, must not be passed if *a* is an `Adversarial` instance.

unpack [bool] If true, returns the adversarial input, otherwise returns the Adversarial object.

starting_point [*numpy.ndarray*] Adversarial input to use as a starting point.

threshold [float] The treshold used by the models binarization. If none, defaults to (model.bounds()[1] - model.bounds()[0]) / 2.

included_in [str] Whether the threshold value itself belongs to the lower or upper interval.

class foolbox.attacks.**PrecomputedImagesAttack**(*input_images*, *output_images*, **args*, ***kwargs*)

Attacks a model using precomputed adversarial candidates.

Parameters

input_images [*numpy.ndarray*] The original images that will be expected by this attack.

output_images [*numpy.ndarray*] The adversarial candidates corresponding to the input_images.

***args** [positional args] Poistional args passed to the *Attack* base class.

****kwargs** [keyword args] Keyword args passed to the *Attack* base class.

__call__(*self*, *input_or_adv*, *label=None*, *unpack=True*)

Attacks a model using precomputed adversarial candidates.

Parameters

input_or_adv [*numpy.ndarray* or `Adversarial`] The original, unperturbed input as a *numpy.ndarray* or an `Adversarial` instance.

label [int] The reference label of the original input. Must be passed if *a* is a *numpy.ndarray*, must not be passed if *a* is an `Adversarial` instance.

unpack [bool] If true, returns the adversarial input, otherwise returns the Adversarial object.

__init__(*self*, *input_images*, *output_images*, **args*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

Gradient-based attacks

GradientAttack	Perturbs the image with the gradient of the loss w.r.t.
GradientSignAttack	Adds the sign of the gradient to the image, gradually increasing the magnitude until the image is misclassified.
FGSM	alias of `foolbox.attacks.gradient.GradientSignAttack`
LinfinityBasicIterativeAttack	The Basic Iterative Method introduced in [R37dbc8f24aee-1].
BasicIterativeMethod	alias of `foolbox.attacks.iterative_projected_gradient.LinfinityBasicIterativeAttack`

Continued on next page

Table 8 – continued from previous page

BIM	alias of `foolbox.attacks.iterative_projected_gradient.LinfinityBasicIterativeAttack`
L1BasicIterativeAttack	Modified version of the Basic Iterative Method that minimizes the L1 distance.
L2BasicIterativeAttack	Modified version of the Basic Iterative Method that minimizes the L2 distance.
ProjectedGradientDescentAttack	The Projected Gradient Descent Attack introduced in [R367e8e10528a-1] without random start.
ProjectedGradientDescent	alias of `foolbox.attacks.iterative_projected_gradient.ProjectedGradientDescentAttack`
`PGD`	alias of `foolbox.attacks.iterative_projected_gradient.ProjectedGradientDescentAttack`
RandomStartProjectedGradientDescentAttack	The Projected Gradient Descent Attack introduced in [Re6066bc39e14-1] with random start.
RandomProjectedGradientDescent	alias of `foolbox.attacks.iterative_projected_gradient.RandomStartProjectedGradientDescentAttack`
RandomPGD	alias of `foolbox.attacks.iterative_projected_gradient.RandomStartProjectedGradientDescentAttack`
MomentumIterativeAttack	The Momentum Iterative Method attack introduced in [R86d363e1fb2f-1].
MomentumIterativeMethod	alias of `foolbox.attacks.iterative_projected_gradient.MomentumIterativeAttack`
LBFGSAttack	Uses L-BFGS-B to minimize the distance between the image and the adversarial as well as the cross-entropy between the predictions for the adversarial and the the one-hot encoded target class.
DeepFoolAttack	Simple and close to optimal gradient-based adversarial attack.
NewtonFoolAttack	Implements the NewtonFool Attack.
DeepFoolL2Attack	
DeepFoolLinfinityAttack	
ADefAttack	Adversarial attack that distorts the image, i.e.
SLSQPAttack	Uses SLSQP to minimize the distance between the image and the adversarial under the constraint that the image is adversarial.
SaliencyMapAttack	Implements the Saliency Map Attack.
IterativeGradientAttack	Like GradientAttack but with several steps for each epsilon.
IterativeGradientSignAttack	Like GradientSignAttack but with several steps for each epsilon.
CarliniWagnerL2Attack	The L2 version of the Carlini & Wagner attack.

Score-based attacks

SinglePixelAttack	Perturbs just a single pixel and sets it to the min or max.
LocalSearchAttack	A black-box attack based on the idea of greedy local search.
ApproximateLBFGSAttack	Same as *LBFGSAttack* with approximate_gradient set to True.

Decision-based attacks

BoundaryAttack	A powerful adversarial attack that requires neither gradients nor probabilities.
SpatialAttack	Adversarially chosen rotations and translations [1].
PointwiseAttack	Starts with an adversarial and performs a binary search between the adversarial and the original for each dimension of the input individually.
GaussianBlurAttack	Blurs the image until it is misclassified.
ContrastReductionAttack	Reduces the contrast of the image until it is misclassified.
AdditiveUniformNoiseAttack	Adds uniform noise to the image, gradually increasing the standard deviation until the image is misclassified.
AdditiveGaussianNoiseAttack	Adds Gaussian noise to the image, gradually increasing the standard deviation until the image is misclassified.
SaltAndPepperNoiseAttack	Increases the amount of salt and pepper noise until the image is misclassified.
BlendedUniformNoiseAttack	Blends the image with a uniform noise image until it is misclassified.

Other attacks

BinarizationRefinementAttack	For models that preprocess their inputs by binarizing the inputs, this attack can improve adversarials found by other attacks.
PrecomputedImagesAttack	Attacks a model using precomputed adversarial candidates.

1.13 `foolbox.adversarial`

Provides a class that represents an adversarial example.

class foolbox.adversarial.**Adversarial**(*model*, *criterion*, *original_image*, *original_class*, *distance=<class 'foolbox.distances.MeanSquaredDistance'>*, *threshold=None*, *verbose=False*)

Defines an adversarial that should be found and stores the result.

The *Adversarial* class represents a single adversarial example for a given model, criterion and reference image. It can be passed to an adversarial attack to find the actual adversarial.

Parameters

model [a `Model` instance] The model that should be fooled by the adversarial.

criterion [a `Criterion` instance] The criterion that determines which images are adversarial.

original_image [a `numpy.ndarray`] The original image to which the adversarial image should be as close as possible.

original_class [int] The ground-truth label of the original image.

distance [a `Distance` class] The measure used to quantify similarity between images.

threshold [float or `Distance`] If not None, the attack will stop as soon as the adversarial perturbation has a size smaller than this threshold. Can be an instance of the `Distance` class passed to the distance argument, or a float assumed to have the same unit as the the given distance. If None, the attack will simply minimize the distance as good as possible. Note that the threshold only influences early stopping of the attack; the returned adversarial does not necessarily have smaller perturbation size than this threshold; the *reached_threshold()* method can be used to check if the threshold has been reached.

adversarial_class

The argmax of the model predictions for the best adversarial found so far.

None if no adversarial has been found.

batch_predictions(*self*, *images*, *greedy=False*, *strict=True*, *return_details=False*)

Interface to model.batch_predictions for attacks.

Parameters

images [*numpy.ndarray*] Batch of images with shape (batch size, height, width, channels).

greedy [bool] Whether the first adversarial should be returned.

strict [bool] Controls if the bounds for the pixel values should be checked.

channel_axis(*self*, *batch*)

Interface to model.channel_axis for attacks.

Parameters

batch [bool] Controls whether the index of the axis for a batch of images (4 dimensions) or a single image (3 dimensions) should be returned.

distance

The distance of the adversarial input to the original input.

gradient(*self*, *image=None*, *label=None*, *strict=True*)

Interface to model.gradient for attacks.

Parameters

image [*numpy.ndarray*] Image with shape (height, width, channels). Defaults to the original image.

label [int] Label used to calculate the loss that is differentiated. Defaults to the original label.

strict [bool] Controls if the bounds for the pixel values should be checked.

has_gradient(*self*)

Returns true if _backward and _forward_backward can be called by an attack, False otherwise.

image

The best adversarial found so far.

normalized_distance(*self*, *image*)

Calculates the distance of a given image to the original image.

Parameters

image [*numpy.ndarray*] The image that should be compared to the original image.

Returns

Distance The distance between the given image and the original image.

original_class
The class of the original input (ground-truth, not model prediction).

original_image
The original input.

output
The model predictions for the best adversarial found so far.

None if no adversarial has been found.

predictions (*self*, *image*, *strict=True*, *return_details=False*)
Interface to model.predictions for attacks.

Parameters

image [*numpy.ndarray*] Image with shape (height, width, channels).

strict [bool] Controls if the bounds for the pixel values should be checked.

predictions_and_gradient (*self*, *image=None*, *label=None*, *strict=True*, *return_details=False*)
Interface to model.predictions_and_gradient for attacks.

Parameters

image [*numpy.ndarray*] Image with shape (height, width, channels). Defaults to the original image.

label [int] Label used to calculate the loss that is differentiated. Defaults to the original label.

strict [bool] Controls if the bounds for the pixel values should be checked.

reached_threshold (*self*)
Returns True if a threshold is given and the currently best adversarial distance is smaller than the threshold.

target_class (*self*)
Interface to criterion.target_class for attacks.

1.14 foolbox.utils

foolbox.utils.**softmax** (*logits*)
Transforms predictions into probability values.

Parameters

logits [array_like] The logits predicted by the model.

Returns

numpy.ndarray Probability values corresponding to the logits.

foolbox.utils.**crossentropy** (*label*, *logits*)
Calculates the cross-entropy.

Parameters

logits [array_like] The logits predicted by the model.

label [int] The label describing the target distribution.

Returns

float The cross-entropy between softmax(logits) and onehot(label).

`foolbox.utils.`**`batch_crossentropy`**(*label*, *logits*)

Calculates the cross-entropy for a batch of logits.

Parameters

logits [array_like] The logits predicted by the model for a batch of inputs.

label [int] The label describing the target distribution.

Returns

np.ndarray The cross-entropy between softmax(logits[i]) and onehot(label) for all i.

`foolbox.utils.`**`binarize`**(*x*, *values*, *threshold=None*, *included_in='upper'*)

Binarizes the values of x.

Parameters

values [tuple of two floats] The lower and upper value to which the inputs are mapped.

threshold [float] The threshold; defaults to (values[0] + values[1]) / 2 if None.

included_in [str] Whether the threshold value itself belongs to the lower or upper interval.

`foolbox.utils.`**`imagenet_example`**(*shape=(224, 224)*, *data_format='channels_last'*)

Returns an example image and its imagenet class label.

Parameters

shape [list of integers] The shape of the returned image.

data_format [str] "channels_first" or "channels_last"

Returns

image [array_like] The example image.

label [int] The imagenet label associated with the image.

`foolbox.utils.`**`onehot_like`**(*a*, *index*, *value=1*)

Creates an array like a, with all values set to 0 except one.

Parameters

a [array_like] The returned one-hot array will have the same shape and dtype as this array

index [int] The index that should be set to *value*

value [single value compatible with a.dtype] The value to set at the given index

Returns

numpy.ndarray One-hot array with the given value at the given location and zeros everywhere else.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Bibliography

[R20d0064ee4c9-1] Ian J. Goodfellow, Jonathon Shlens, Christian Szegedy, "Explaining and Harnessing Adversarial Examples", https://arxiv.org/abs/1412.6572

[R37dbc8f24aee-1] Alexey Kurakin, Ian Goodfellow, Samy Bengio, "Adversarial examples in the physical world",

https://arxiv.org/abs/1607.02533

[R367e8e10528a-1] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, Adrian Vladu, "Towards Deep Learning Models Resistant to Adversarial Attacks", https://arxiv.org/abs/1706.06083

[Re6066bc39e14-1] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, Adrian Vladu, "Towards Deep Learning Models Resistant to Adversarial Attacks", https://arxiv.org/abs/1706.06083

[R86d363e1fb2f-1] Yinpeng Dong, Fangzhou Liao, Tianyu Pang, Hang Su, Jun Zhu, Xiaolin Hu, Jianguo Li, "Boosting Adversarial Attacks with Momentum", https://arxiv.org/abs/1710.06081

[Rf3ff9c7ff5d3-1] https://arxiv.org/abs/1510.05328

[Rb4dd02640756-1] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, Pascal Frossard, "DeepFool: a simple and accurate method to fool deep neural networks", https://arxiv.org/abs/1511.04599

[R6a972939b320-1] Uyeong Jang et al., "Objective Metrics and Gradient Descent Algorithms for Adversarial Examples in Machine Learning", https://dl.acm.org/citation.cfm?id=3134635

[R08e06ca693ba-1] Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z. Berkay Celik, Ananthram Swami, "The Limitations of Deep Learning in Adversarial Settings", https://arxiv.org/abs/1511.07528

[Rc2cb572b91c5-1] Nicholas Carlini, David Wagner: "Towards Evaluating the Robustness of Neural Networks", https://arxiv.org/abs/1608.04644

[Rc2cb572b91c5-2] https://github.com/carlini/nn_robust_attacks

[Rb320cee6998a-1] Nina Narodytska, Shiva Prasad Kasiviswanathan, "Simple Black-Box Adversarial Perturbations for Deep Networks", https://arxiv.org/abs/1612.06299

[Re72ca268aa55-1] Wieland Brendel (*), Jonas Rauber (*), Matthias Bethge, "Decision-Based Adversarial Attacks: Reliable Attacks Against Black-Box Machine Learning Models", https://arxiv.org/abs/1712.04248

[Rdffd25498f9d-1] Logan Engstrom*, Brandon Tran*, Dimitris Tsipras*, Ludwig Schmidt, Aleksander Mądry: "A Rotation and a Translation Suffice: Fooling CNNs with Simple Transformations", http://arxiv.org/abs/1712.02779

Python Module Index

f

Index

Symbols

A

B

C

D

F

G

H

I

K

L

M

N