Foolbox Documentation

Release 1.0.0

Jonas Rauber & Wieland Brendel

Mar 20, 2018

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

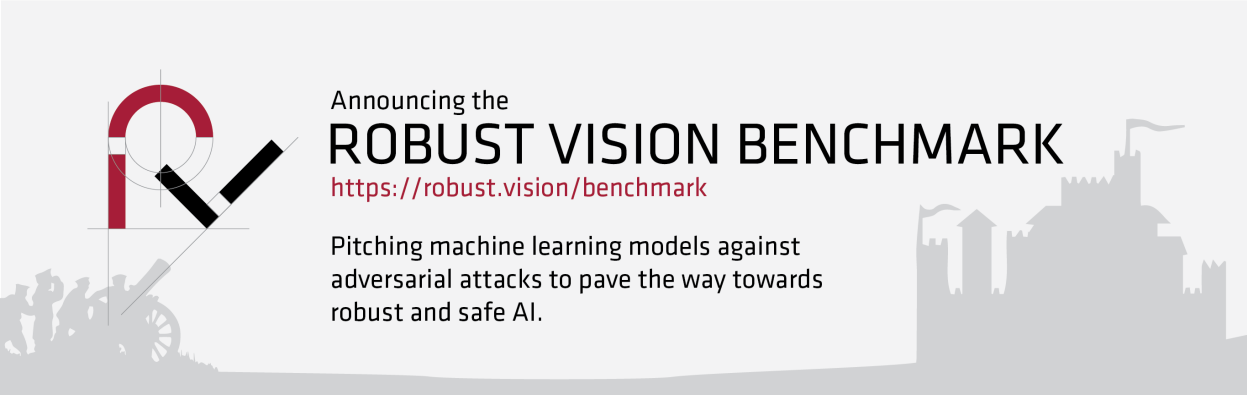
and it is easy to extend to other frameworks.

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

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

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 a targeted adversaial 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()

# 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 Development

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

1.5.1 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.

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 .
```

1.5.2 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.

1.6 `foolbox.models`

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

1.6.1 Models

Model	Base class to provide attacks with a unified interface to models.
DifferentiableModel	Base class for differentiable models that provide gradients.
TensorFlowModel	Creates a *Model* instance from existing *TensorFlow* tensors.
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.

Continued on next page

Table 1.1 – continued from previous page

LasagneModel	Creates a *Model* instance from a *Lasagne* network.
MXNetModel	Creates a *Model* instance from existing *MXNet* symbols and weights.

1.6.2 Wrappers

ModelWrapper	Base class for models that wrap other models.
GradientLess	Turns a model into a model without gradients.
CompositeModel	Combines predictions of a (black-box) model with the gradient of a (substitute) model.

1.6.3 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(*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:

prediction()

num_classes()

Determines the number of classes.

Returns int

The number of classes for which the model creates predictions.

predictions(*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(*gradient*, *image*)

Backpropages 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(*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(*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.

class foolbox.models.**PyTorchModel**(*model*, *bounds*, *num_classes*, *channel_axis=1*, *cuda=True*, *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.

cuda : bool

A boolean specifying whether the model uses CUDA.

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.

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.

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.

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.

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.

class foolbox.models.**GradientLess**(*model*)
Turns a model into a model without gradients.

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.

1.7 foolbox.criteria

Provides classes that define what is adversarial.

1.7.1 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.

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

Continued on next page

Table 1.4 – continued from previous page

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

1.7.2 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
```

1.7.3 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(*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()

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.

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.

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.

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.

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.

1.8 foolbox.distances

Provides classes to measure the distance between images.

1.8.1 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.

1.8.2 Aliases

MSE	alias of *MeanSquaredDistance*
MAE	alias of *MeanAbsoluteDistance*
Linf	alias of *Linfinity*

1.8.3 Base class

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

Distance	Base class for distances.

1.8.4 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 *MeanSquaredDistance*

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

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

1.9 foolbox.attacks

1.9.1 Gradient-based attacks

class foolbox.attacks.**GradientSignAttack**(*model=None*, *criterion=<foolbox.criteria.Misclassification object>*)
Adds the sign of the gradient to the image, gradually increasing the magnitude until the image is misclassified.

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

class foolbox.attacks.**IterativeGradientSignAttack**(*model=None*, *criterion=<foolbox.criteria.Misclassification object>*)
Like GradientSignAttack but with several steps for each epsilon.

class foolbox.attacks.**GradientAttack**(*model=None*, *criterion=<foolbox.criteria.Misclassification object>*)
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.

class foolbox.attacks.**IterativeGradientAttack**(*model=None*, *criterion=<foolbox.criteria.Misclassification object>*)
Like GradientAttack but with several steps for each epsilon.

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

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 *[R1717]* to support other targeted criteria and other distance measures.

References

[R1717]

class foolbox.attacks.**DeepFoolAttack**(*model=None*, *criterion=<foolbox.criteria.Misclassification object>*)

Simple and close to optimal gradient-based adversarial attack.

Implementes DeepFool introduced in *[R1919]*.

References

[R1919]

class foolbox.attacks.**SLSQPAttack**(*model=None*, *criterion=<foolbox.criteria.Misclassification object>*)

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

class foolbox.attacks.**SaliencyMapAttack**(*model=None*, *criterion=<foolbox.criteria.Misclassification object>*)

Implements the Saliency Map Attack.

The attack was introduced in *[R2121]*.

References

[R2121]

1.9.2 Score-based attacks

class foolbox.attacks.**SinglePixelAttack**(*model=None*, *criterion=<foolbox.criteria.Misclassification object>*)

Perturbs just a single pixel and sets it to the min or max.

class foolbox.attacks.**LocalSearchAttack**(*model=None*, *criterion=<foolbox.criteria.Misclassification object>*)

A black-box attack based on the idea of greedy local search.

This implementation is based on the algorithm in *[R2323]*.

References

[R2323]

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

Same as *LBFGSAttack* with approximate_gradient set to True.

1.9.3 Decision-based attacks

class foolbox.attacks.**BoundaryAttack**(*model=None*, *criterion=<foolbox.criteria.Misclassification object>*)

A powerful adversarial attack that requires neither gradients nor probabilities.

This is the reference implementation for the attack introduced in *[R1313]*.

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

[R1313]

class foolbox.attacks.**GaussianBlurAttack**(*model=None*, *criterion=<foolbox.criteria.Misclassification object>*)

Blurs the image until it is misclassified.

class foolbox.attacks.**ContrastReductionAttack**(*model=None*, *criterion=<foolbox.criteria.Misclassification object>*)

Reduces the contrast of the image until it is misclassified.

class foolbox.attacks.**AdditiveUniformNoiseAttack**(*model=None*, *criterion=<foolbox.criteria.Misclassification object>*)

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

class foolbox.attacks.**AdditiveGaussianNoiseAttack**(*model=None*, *criterion=<foolbox.criteria.Misclassification object>*)

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

class foolbox.attacks.**SaltAndPepperNoiseAttack**(*model=None*, *criterion=<foolbox.criteria.Misclassification object>*)

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

class foolbox.attacks.**ResetAttack**(*model=None*, *criterion=<foolbox.criteria.Misclassification object>*)

Starts with an adversarial and resets as many values as possible to the values of the original.

Based on the FindWitness algorithm described in [1].

References

[R1515]

1.9.4 Other attacks

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.

Gradient-based attacks

GradientSignAttack	Adds the sign of the gradient to the image, gradually increasing the magnitude until the image is misclassified.
IterativeGradientSignAttack	Like GradientSignAttack but with several steps for each epsilon.
GradientAttack	Perturbs the image with the gradient of the loss w.r.t.
IterativeGradientAttack	Like GradientAttack but with several steps for each epsilon.
FGSM	alias of *GradientSignAttack*
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.
DeepFoolL2Attack	
DeepFoolLinfinityAttack	
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.

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.
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.
BlendedUniformNoiseAttack	Blends the image with a uniform noise image until it is misclassified.
SaltAndPepperNoiseAttack	Increases the amount of salt and pepper noise until the image is misclassified.
ResetAttack	Starts with an adversarial and resets as many values as possible to the values of the original.

Other attacks

PrecomputedImagesAttack	Attacks a model using precomputed adversarial candidates.

1.10 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'>*, *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.

batch_predictions(*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(*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.

gradient(*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()
Returns true if _backward and _forward_backward can be called by an attack, False otherwise.

normalized_distance(*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.

predictions(*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(*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.

target_class()
Interface to criterion.target_class for attacks.

1.11 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).

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Bibliography

[R1717] https://arxiv.org/abs/1510.05328

[R1919] 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

[R2121] 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

[R2323] Nina Narodytska, Shiva Prasad Kasiviswanathan, "Simple Black-Box Adversarial Perturbations for Deep Networks", https://arxiv.org/pdf/1612.06299.pdf

[R1313] Wieland Brendel (*), Jonas Rauber (*), Matthias Bethge, "Decision-Based Adversarial Attacks: Reliable Attacks Against Black-Box Machine Learning Models", https://arxiv.org/abs/1712.04248

[R1515] Daniel Lowd, Christopher Meek, "Adversarial Learning", Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining, KDD 2005.

Python Module Index

f

Index

N

O

P

R

S

T