# Perceptron Benchmark Documentation

['Yunhan Jia']

Jun 04, 2019

# User Guide

Perceptron is a benchmark to test safety and security properties of neural networks for perceptual tasks.

It comes with support for many frameworks to build models including

- TensorFlow
- PyTorch
- Keras
- Cloud API
- PaddlePaddle

See currently supported evaluation metrics, models, adversarial criteria, and verification methods in *Summary*.

See current *Leaderboard*.

# CHAPTER 1

# Overview

`perceptron` benchmark improves upon the existing adversarial toolbox such as `cleverhans`, `foolbox`, `IBM ART`, `advbox` in three important aspects:

- Consistent API design that enables easy evaluation of models across different deep learning **frameworks**, computer vision **tasks**, and adversarial **criterions**.
- Standardized metric design that enables DNN models' robustness to be compared on a large collection of **security** and safety properties.
- Gives verifiable robustness bounds for security and safety properties.

# CHAPTER 2

# Running benchmarks

You can run evaluation against DNN models with chosen parameters using `launcher`. For example:

```
python perceptron/launcher.py \
    --framework keras \
    --model resnet50 \
    --criteria misclassification\
    --metric carlini_wagner_l2 \
    --image example.png
```

In above command line, the user lets the framework as `keras`, the model as `resnet50`, the criterion as `misclassification` (i.e., we want to generate an adversary which is similar to the original image but has different predicted label), the metric as `carlini_wagner_l2`, the input image as `example.png`.

You can try different combinations of frameworks, models, criteria, and metrics. To see more options using *-h* for help message.

```
python perceptron/launcher.py -h
```

We also provide a coding example which serves the same purpose as above command line. Please refer to *Examples* for more details.

## 2.1 Examples

Here you can find a collection of examples.

### 2.1.1 Keras Framework

#### Keras: ResNet50 - C&W2 Benchmarking

This example benchmarks the robustness of ResNet50 model against $C\&W_2$ attack by measuring the minimal required $L_\infty$ perturbation for a $C\&W_2$ attack to success. You can find the example in the file *example/keras_cw_example.py*.

Run the example with command *python example/keras_cw_example.py*

```
from __future__ import absolute_import

import numpy as np
import keras.applications as models
from perceptron.models.classification.keras import KerasModel
from perceptron.utils.image import imagenet_example
from perceptron.benchmarks.carlini_wagner import CarliniWagnerL2Metric
from perceptron.utils.criteria.classification import Misclassification

# instantiate the model from keras applications
resnet50 = models.ResNet50(weights='imagenet')

# initialize the KerasModel
preprocessing = (np.array([104, 116, 123]), 1)  # the mean and stv of the whole␣
↪dataset
kmodel = KerasModel(resnet50, bounds=(0, 255), preprocessing=preprocessing)

# get source image and label
# the model expects values in [0, 255], and channles_last
image, _ = imagenet_example(data_format='channels_last')
label = np.argmax(kmodel.predictions(image))
metric = CarliniWagnerL2Metric(kmodel, criterion=Misclassification())
adversary = metric(image, label, unpack=False)
```

By running the file *examples/keras_cw_example.py*, you will get the following output:

```
Summary:
Configuration: --framework keras --model resnet50 --criterion misclassification --metric carlini_wagner_l2
The predicted label of original image is otter
The predicted label of adversary image is weasel
Minimum perturbation required: normalized MSE = 2.01e-06
```


The command line tool *perceptron/launcher.py* provide users a way to play different combinations of frameworks and models without writing code. For example, the following command line serves the same purpose as above example.

```
python perceptron/launcher.py \
        --framework keras \
```

(continues on next page)

(continued from previous page)

```
        --model resnet50 \
        --criteria misclassification \
        --metric carlini_wagner_l2 \
        --image example.png
```

### Keras: YOLOv3 - C&W2 Benchmarking (Object detection)

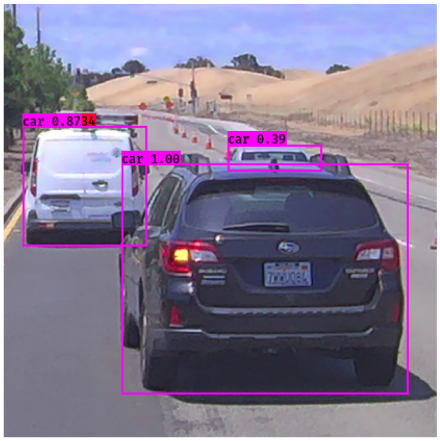
This example benchmarks the robustness of YOLOv3 model (the state-of-the-art object detection model) against $C\&W_2$ attack by measuring the minimal required $L_\infty$ perturbation for a $C\&W_2$ attack to success. The minimum Mean Squred Distance (MSE) will be logged. You can find the example in the file *example/keras_cw_yolo_example.py*. Run the example with command *python example/keras_cw_yolo_example.py*

```
from perceptron.zoo.yolov3.model import YOLOv3
from perceptron.models.detection.keras_yolov3 import KerasYOLOv3Model
from perceptron.utils.image import load_image
from perceptron.benchmarks.carlini_wagner import CarliniWagnerLinfMetric
from perceptron.utils.criteria.detection import TargetClassMiss

# instantiate the model from keras applications
yolov3 = YOLOv3()

# initialize the KerasYOLOv3Model
kmodel = KerasYOLOv3Model(yolov3, bounds=(0, 1))

# get source image and label
# the model expects values in [0, 1], and channles_last
image = load_image(shape=(416, 416), bounds=(0, 1), fname='car.png')
metric = CarliniWagnerLinfMetric(kmodel, criterion=TargetClassMiss(2))
adversary = metric(image, binary_search_steps=1, unpack=False)
```

The object detection results for the original image and the adversarial one is shown as follows.

**Keras, YoLoV3, TargetClassMiss, CarliniWagnerLINF**

Origin




Adversary

In the following, we provide a command line which serves the same purpose as above piece of code.

```
python perceptron/launcher.py \
        --framework keras \
        --model yolo_v3 \
        --criteria target_class_miss --target_class 2 \
        --metric carlini_wagner_linf \
        --image car.png
```

The command line tool *perceptron/launcher.py* allows users to try different combinations of framework, model, criteria, metric and image without writing detailed code. In the following, we provide more API examples and their corresponding command lines.

### Keras: VGG16 - Blended Uniform Noise

In this example, we try different model and metric. Besides, we choose *TopKMisclassification*. The $K$ equals to 10 in this example. It means we want to generate an adversary whose predicted label is not among the top 10 highest possible predictions.

```
import numpy as np
import keras.applications as models
from perceptron.models.classification.keras import KerasModel
from perceptron.utils.image import imagenet_example
from perceptron.benchmarks.blended_noise import BlendedUniformNoiseMetric
from perceptron.utils.criteria.classification import TopKMisclassification

# instantiate the model from keras applications
vgg16 = models.VGG16(weights='imagenet')

# initialize the KerasModel
# keras vgg16 has input bound (0, 255)
preprocessing = (np.array([104, 116, 123]), 1)  # the mean and stv of the whole␣
↪dataset
kmodel = KerasModel(vgg16, bounds=(0, 255), preprocessing=preprocessing)

# get source image and label
# the model expects values in [0, 255], and channles_last
image, _ = imagenet_example(data_format='channels_last')
label = np.argmax(kmodel.predictions(image))
metric = BlendedUniformNoiseMetric(kmodel, criterion=TopKMisclassification(10))
adversary = metric(image, label, unpack=False, epsilons=100)  # choose 100 different␣
↪epsilon values in [0, 1]
```

The corresponding command line is

```
python perceptron/launcher.py \
        --framework keras \
        --model vgg16 \
        --criteria topk_misclassification \
        --metric blend_uniform_noise \
        --image example.png
```

By running the example file *examples/keras_bu_example.py*, you will get the output as follows.

```
Summary:
Configuration: --framework Keras --model VGG16 --criterion Top10Misclassification --metric BlendedUniform
The predicted label of original image is otter
The predicted label of adversary image is guinea pig, Cavia cobaya
Minimum perturbation required: normalized MSE = 3.43e-02
```


### Keras: Xception - Additive Gaussion Noise

In this example, we show the performance of model *Xception* under the metric *Additive Gaussian Noise*.

```
import numpy as np
import keras.applications as models
from perceptron.models.classification.keras import KerasModel
from perceptron.utils.image import imagenet_example
from perceptron.benchmarks.additive_noise import AdditiveGaussianNoiseMetric
from perceptron.utils.criteria.classification import TopKMisclassification

# instantiate the model from keras applications
xception = models.Xception(weights='imagenet')

# initialize the KerasModel
# keras xception has input bound (0, 1)
mean = np.array([0.485, 0.456, 0.406]).reshape((1, 1, 3))
std = np.array([0.229, 0.224, 0.225]).reshape((1, 1, 3))
kmodel = KerasModel(xception, bounds=(0, 1), preprocessing=(mean, std))

# get source image and label
# the model Xception expects values in [0, 1] with shape (299, 299), and channles_last
image, _ = imagenet_example(shape=(299, 299), data_format='channels_last')
image /= 255.0
label = np.argmax(kmodel.predictions(image))
metric = AdditiveGaussianNoiseMetric(kmodel, criterion=TopKMisclassification(10))
adversary = metric(image, label, unpack=False, epsilons=1000)  # choose 1000␣
↪different epsilon values in [0, 1]
```

The corresponding command line is:

```
python perceptron/launcher.py \
       --framework keras \
```

(continues on next page)

(continued from previous page)

```
        --model xception \
        --criteria topk_misclassification \
        --metric additive_gaussian_noise \
        --image example.png
```

By runnig the corresponding example file *examples/keras_ag_example.py*, you will get the following output.

```
Summary:
Configuration: --framework Keras --model Xception --criterion Top10Misclassification --metric AdditiveGaussian
The predicted label of original image is otter
The predicted label of adversary image is meerkat, mierkat
Minimum perturbation required: normalized MSE = 4.33e-02
```


### 2.1.2 PyTorch Framework

Note: for the pytorch model, the range of the input image is always in $[0, 1]$.

#### PyTorch: ResNet18 - C&W Benchmarking

This example verifies the robustness of ResNet18 model against $C\&W_2$ by measuring the required $L_2$ pertubutation for a $C\&W_2$ attack to success. The example with more details can be found in the file *example/torch_cw_example.py*.

```
import torch
import torchvision.models as models
import numpy as np
from perceptron.models.classification.pytorch import PyTorchModel
from perceptron.utils.image import imagenet_example
from perceptron.benchmarks.carlini_wagner import CarliniWagnerL2Metric
from perceptron.utils.criteria.classification import Misclassification

# instantiate the model
resnet18 = models.resnet18(pretrained=True).eval()
if torch.cuda.is_available():
    resnet18 = resnet18.cuda()
```

(continues on next page)

(continued from previous page)

```
# initialize the PyTorchModel
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 = PyTorchModel(
    resnet18, bounds=(0, 1), num_classes=1000, preprocessing=(mean, std))

# get source image and label
image, label = imagenet_example(data_format='channels_first')
image = image / 255.  # because our model expects values in [0, 1]
metric = CarliniWagnerL2Metric(fmodel, criterion=Misclassification())
adversary = metric(image, label, unpack=False)
```

The corresponding command line is

```
python perceptron/launcher.py \
        --framework pytorch \
        --model resnet18 \
        --criteria misclassification \
        --metric carlini_wagner_l2 \
        --image example.png
```

By running the example file *example/torch_cw_example.py*, you will get the following output.

```
Summary:
Configuration: --framework PyTorch --model Resent18 --criterion Misclassification --metric CarliniWagnerL2
The predicted label of original image is otter
The predicted label of adversary image is weasel
Minimum perturbation required: normalized MSE = 8.92e-08
```


Above example shows that it is easy to fool a well-tuned classifier by adding small pertubation to the target.

### PyTorch: VGG11 - SaltAndPepper Noise

In this example, we choose a different model and metric from above example. The detailed example file is at *examples/torch_sp_example.py*

```
from __future__ import absolute_import
import torch
import torchvision.models as models
import numpy as np
from perceptron.models.classification.pytorch import PyTorchModel
from perceptron.utils.image import imagenet_example
from perceptron.benchmarks.salt_pepper import SaltAndPepperNoiseMetric
from perceptron.utils.criteria.classification import Misclassification

# instantiate the model
vgg11 = models.vgg11(pretrained=True).eval()
if torch.cuda.is_available():
    vgg11 = vgg11.cuda()

# initialize the PyTorchModel
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 = PyTorchModel(
    vgg11, bounds=(0, 1), num_classes=1000, preprocessing=(mean, std))

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

# set the label as the predicted one
true_label = np.argmax(fmodel.predictions(image))
# set the type of noise which will used to generate the adversarial examples
metric = SaltAndPepperNoiseMetric(fmodel, criterion=Misclassification())
adversary = metric(image, true_label, unpack=False) # set 'unpack' as false so we can␣
↪access the detailed info of adversary
```

The corresponding command line is

```
python perceptron/launcher.py \
        --framework pytorch \
        --model vgg11 \
        --criteria misclassification \
        --metric salt_and_pepper_noise \
        --image example.png
```

By running the example file *example/torch_sp_example.py*, you will get the following output:

```
Summary:
Configuration: --framework PyTorch --model Vgg11 --criterion Misclassification --metric SaltAndPepper
The predicted label of original image is weasel
The predicted label of adversary image is black-footed ferret, ferret, Mustela nigripes
Minimum perturbation required: normalized MSE = 3.35e-05
```

The model *VGG11* does not produce the correct prediction at the beginning. By revising only two pixels, we are able to fool the classifier to make another wrong prediction.

### PyTorch: ResNet18 - Brightness Verification

This example verifies the robustness of ResNet18 model against brightness variations, and it will give a verifiable bound. The detailed example file is at *examples/torch_br_example.py*


```
from __future__ import absolute_import

import torch
import torchvision.models as models
import numpy as np
from perceptron.models.classification.pytorch import PyTorchModel
from perceptron.utils.image import imagenet_example
from perceptron.benchmarks.brightness import BrightnessMetric
from perceptron.utils.criteria.classification import Misclassification

# instantiate the model
resnet18 = models.resnet18(pretrained=True).eval()
if torch.cuda.is_available():
    resnet18 = resnet18.cuda()

# initialize the PyTorchModel
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 = PyTorchModel(
    resnet18, bounds=(0, 1), num_classes=1000, preprocessing=(mean, std))

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

# set the type of noise which will used to generate the adversarial examples
metric = BrightnessMetric(fmodel, criterion=Misclassification())

# set the label as the predicted one
label = np.argmax(fmodel.predictions(image))

adversary = metric(image, label, unpack=False, verify=True)  # set 'unpack' as false␣
↪so we can access the detailed info of adversary
```

The corresponding command line is

```
python perceptron/launcher.py \
        --framework pytorch \
```

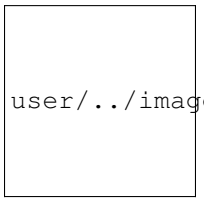
(continues on next page)

(continued from previous page)

```
        --model resnet18 \
        --criteria misclassification \
        --metric brightness --verify\
        --image example.png
```

By running the example file *example/torch_sp_example.py*, you will get the following output:

```
Summary:
Configuration: --framework PyTorch --model ResNet18 --criterion Misclassification --metric Brightness
The predicted label of original image is otter
The predicted label of adversary image is giant panda, panda, panda bear, coon bear, Ailuropoda melanoleuca
Minimum perturbation required: normalized MSE = 1.01e-01
Verifiable bound: (2.0642983467839073, 0.08535178777230501)
```



user/../images/pytorch_resnet18_misclassification_brightn

In above figure, we only show the adversarial exampe when we increase the brightness. Let $(L, U)$ denotes the verifiable bound. The value $(L)$ (resp. $U$) means if we want to increase (resp. decrease) the brightness to generate adversary example, the minimal brightness pertubation we need is $(L)$ (resp. $U$). (Note: brightness pertubation $L$ means multiplying each pixel in the image with value $L$)

Caution: when using the verifiable metric, if the parameter *verify* is not set to *True*, the program will just slice the search space according to the epsilon value, not the minimal step the verifiable metric requires. Hence, the output verifiable bound has a wider range than the actual verifiable bound.

### 2.1.3 Cloud Framework

#### Cloud API: Google SafeSearch - Spatial Benchmarking

This example benchmarks the robustness of the SafeSearch API provided by Google cloud AI platform against spatial transformation attacks by attack by measuring the minimal required pixel-wise shift for a spatial attack to success.

```
from __future__ import absolute_import
from perceptron.utils.image import imagenet_example, load_image
from perceptron.models.classification.cloud import GoogleSafeSearchModel
from perceptron.benchmarks.contrast_reduction import ContrastReductionMetric
from perceptron.utils.criteria.classification import MisclassificationSafeSearch
import numpy as np
from perceptron.utils.tools import plot_image
from perceptron.utils.tools import bcolors


# before running the example, please set the variable GOOGLE_APPLICATION_CREDENTIALS␣
↪as follows
# export GOOGLE_APPLICATION_CREDENTIALS="[PATH]"
# replace [PATH] with the file path of the JSON file that contains your service␣
↪account key
# For more details, please refer to https://cloud.google.com/docs/authentication/
↪getting-started#auth-cloud-implicit-python
```

(continues on next page)

(continued from previous page)

```
model = GoogleSafeSearchModel()

# get source image and label
image = load_image(dtype=np.uint8, fname='porn.jpeg')
metric = ContrastReductionMetric(model, criterion=MisclassificationSafeSearch())
adversary = metric(image, epsilons=10, unpack=False)
```

The corresponding command line is

```
python perceptron/launcher.py \
        --framework cloud \
        --model google_safesearch \
        --criteria misclassification_safesearch \
        --metric contrast_reduction \
        --image porn.jpeg
```

Run the corresponding example file *examples/safesearch_noise_example.py* The information printed on the screen is:

```
Summary:
Configuration: --framework Cloud --model GoogleSafeSearch --criterion Misclassification --metric ContrastReduction
The predicted label of original image is {'adult': 5, 'spoof': 1, 'medical': 1, 'violence': 1, 'racy': 5}
The predicted label of adversary image is {'adult': 2, 'spoof': 1, 'medical': 1, 'violence': 1, 'racy': 3}
Minimum perturbation required: normalized MSE = 9.13e-02
```

Note: For each cloud model, we have a specific criterion. Here the criterion for *Google Safesearch* is *misclassification_safesearcha*.

## 2.2 Leaderboard

### 2.2.1 Salt & Pepper noise benchmarking

Robustness measured by the minimum perturbation required to successfully fool the model. Perturbation size measured by the Mean Squared Error (MSE) between adversarial and original image.

### 2.2.2 Rotation angle with verifiable Robustness

The prediction on test image is verifiable robust in the range of rotation angles indicated in the plot.

### 2.2.3 More being visualized and added

Coming Soon...

## 2.3 Summary

### 2.3.1 Supported attacks

Provides different attack and evaluation approaches.

| | |
|---|---|
| *CarliniWagnerL2Metric* | The L2 version of C&W attack. |
| *CarliniWagnerLinfMetric* | The L_inf version of C&W attack. |
| *AdditiveNoiseMetric* | Base class for metric that tests models against additive noise. |
| *AdditiveGaussianNoiseMetric* | Metric that tests models against Gaussian noise. |
| *AdditiveUniformNoiseMetric* | Metric that tests models against uniform noise. |
| *BlendedUniformNoiseMetric* | Blends the image with a uniform noise image until it is misclassified. |
| *GaussianBlurMetric* | Metric that tests models against Gaussian blurs. |
| *BrightnessMetric* | Metric that tests models against brightness variations. |
| *ContrastReductionMetric* | Metric that tests models against brightness variations. |
| *MotionBlurMetric* | Motion blurs the image until it is misclassified. |
| *RotationMetric* | Metric that tests models against rotations. |
| *SaltAndPepperNoiseMetric* | Add salt and pepper noise. |
| *SpatialMetric* | Metric that tests models against spatial transformations. |

### 2.3.2 Supported models

Provides class to wrap existing models in different frameworks so that they provide a unified API to the benchmarks.

| | |
|---|---|
| *KerasModel* | Create a `Model` instance from a *Keras* model. |
| *PyTorchModel* | Creates a `Model` instance from a *PyTorch* module. |
| *AipModel* | Base class for models hosted on Baidu AIP platform. |
| *AipAntiPornModel* | Create a `Model` instance from an *AipAntiPorn* model. |
| *GoogleCloudModel* | Base class for models in Google Cloud. |
| *GoogleSafeSearchModel* | Create a :class: *Model* instance from a *GoogleSafeSearchModel* model. |
| *GoogleObjectDetectionModel* | Create a :class: *Model* instance from a *GoogleObjectDetectionModel* model. |

| |
|---|
| `KerasYOLOv3Model` |
| `KerasSSD300Model` |
| `KerasResNet50RetinaNetModel` |

### 2.3.3 Supported adversarial criterions

Provides class to wrap all adversarial criterions so that attacks has uniform API access.

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

Continued on next page

Table 4 – continued from previous page

| | |
|---|---|
| `OriginalClassProbability` | Defines adversarials as images for which the probability of original class is below a given threshold. |
| `TargetClassProbability` | Defines adversarials as images for which the probability of a given target class is above a given threshold. |
| `MisclassificationAntiPorn` | Defines adversarials as image for which the probability of being *normal* is larger than the probability of being *porn*. |
| `MisclassificationSafeSearch` | Defines adversarials as image for which the probability of being *unsafe* is lower than a threshold. |
| `TargetClassMiss` | Defines adversarials as images for which the target class is not in the detection result. |
| `TargetClassMissGoogle` | Defines adversarials as images for which the target class is not in the Google object detection result. |
| `WeightedAP` | Defines adversarials as weighted AP value larger than given threshold. |

### 2.3.4 Supported distance metrics

Provides classes to measure the distance between two images.

| | |
|---|---|
| *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. |
| *MSE* | alias of *perceptron.utils.distances.MeanSquaredDistance* |
| *MAE* | alias of *perceptron.utils.distances.MeanAbsoluteDistance* |
| *Linf* | alias of *perceptron.utils.distances.Linfinity* |

## 2.4 perceptron.benchmarks

Provides different attack and evaluation approaches.

| | |
|---|---|
| *CarliniWagnerL2Metric* | The L2 version of C&W attack. |
| *CarliniWagnerLinfMetric* | The L_inf version of C&W attack. |
| *AdditiveNoiseMetric* | Base class for metric that tests models against additive noise. |
| *AdditiveGaussianNoiseMetric* | Metric that tests models against Gaussian noise. |
| *AdditiveUniformNoiseMetric* | Metric that tests models against uniform noise. |
| *BlendedUniformNoiseMetric* | Blends the image with a uniform noise image until it is misclassified. |
| *GaussianBlurMetric* | Metric that tests models against Gaussian blurs. |
| *BrightnessMetric* | Metric that tests models against brightness variations. |
| *ContrastReductionMetric* | Metric that tests models against brightness variations. |

Continued on next page

Table 6 – continued from previous page

| | |
|---|---|
| *MotionBlurMetric* | Motion blurs the image until it is misclassified. |
| *RotationMetric* | Metric that tests models against rotations. |
| *SaltAndPepperNoiseMetric* | Add salt and pepper noise. |
| *SpatialMetric* | Metric that tests models against spatial transformations. |

**class** perceptron.benchmarks.**Metric**(*model=None*, *criterion=None*, *distance=<class 'perceptron.utils.distances.MeanSquaredDistance'>*, *threshold=None*)

Abstract base class for DNN robustness metrics.

The *Metric* class represents a robustness testing metric that searches for adversarial examples with minimum perturbation. It should be subclassed when implementing new metrics.

**Parameters**

**model** [a Model instance] The model that should be tested by the metric.

**criterion** [a Criterion instance] The criterion that determines which images are adversarial.

**distance** [a Distance class] The measure used to quantify similarity between images.

**threshold** [float or Distance] If not None, the testing 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 test will simply minimize the distance as good as possible. Note that the threshold only influences early stopping of the test; 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.

### Notes

If a subclass overwrites the constructor, it should call the super constructor with args and kwargs.

**__call__**(*self*, *input*, ***kwargs*)
Call self as a function.

**__init__**(*self*, *model=None*, *criterion=None*, *distance=<class 'perceptron.utils.distances.MeanSquaredDistance'>*, *threshold=None*)
Initialize self. See help(type(self)) for accurate signature.

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

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

**Returns**

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

### Notes

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

### 2.4.1 Safety Metrics

**class** perceptron.benchmarks.**AdditiveNoiseMetric**(*model=None*, *criterion=None*, *distance=<class 'perceptron.utils.distances.MeanSquaredDistance'>*, *threshold=None*)

Base class for metric that tests models against additive noise.

**__call__**(*self*, *adv*, *annotation=None*, *unpack=True*, *abort_early=True*, *epsilons=10000*)

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

**Parameters**

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

**annotation** [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.

**abort_early** [bool] If true, returns when got first adversarial, otherwise returns when all the iterations are finished.

**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** perceptron.benchmarks.**AdditiveGaussianNoiseMetric**(*model=None*, *criterion=None*, *distance=<class 'perceptron.utils.distances.MeanSquaredDistance'>*, *threshold=None*)

Metric that tests models against Gaussian noise.

**class** perceptron.benchmarks.**AdditiveUniformNoiseMetric**(*model=None*, *criterion=None*, *distance=<class 'perceptron.utils.distances.MeanSquaredDistance'>*, *threshold=None*)

Metric that tests models against uniform noise.

**class** perceptron.benchmarks.**BlendedUniformNoiseMetric**(*model=None*, *criterion=None*, *distance=<class 'perceptron.utils.distances.MeanSquaredDistance'>*, *threshold=None*)

Blends the image with a uniform noise image until it is misclassified.

**__call__**(*self*, *adv*, *annotation=None*, *unpack=True*, *abort_early=True*, *epsilons=10000*, *max_directions=1000*)

Metric that tests models against blended uniform noise.

**Parameters**

**adv** [*numpy.ndarray*] The original, unperturbed input as a *numpy.ndarray*.

**annotation** [int] The reference label of the original input.

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

**abort_early** [bool] If true, returns when got first adversarial, otherwise returns when all the iterations are finished.

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

**max_directions** [int] Maximum number of random images to try.

**class** perceptron.benchmarks.**GaussianBlurMetric**(*model=None*, *criterion=None*, *distance=<class 'perceptron.utils.distances.MeanSquaredDistance'>*, *threshold=None*)

Metric that tests models against Gaussian blurs.

**__call__**(*self*, *adv*, *annotation=None*, *unpack=True*, *abort_early=True*, *epsilons=10000*)

Blurs the image until it is misclassified.

**Parameters**

**adv** [*numpy.ndarray*] The original, unperturbed input as a *numpy.ndarray*.

**annotation** [int] The reference label of the original input.

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

**abort_early** [bool] If true, returns when got first adversarial, otherwise returns when all the iterations are finished.

**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** perceptron.benchmarks.**BrightnessMetric**(*model=None*, *criterion=None*, *distance=<class 'perceptron.utils.distances.MeanSquaredDistance'>*, *threshold=None*)

Metric that tests models against brightness variations.

**__call__**(*self*, *adv*, *annotation=None*, *unpack=True*, *abort_early=True*, *verify=False*, *epsilons=1000*)

Change the brightness of the image until it is misclassified.

**Parameters**

**adv** [*numpy.ndarray*] The original, unperturbed input as a *numpy.ndarray*.

**annotation** [int] The reference label of the original input. Must be passed if *a* is a *numpy.ndarray*.

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

**abort_early** [bool] If true, returns when got first adversarial, otherwise returns when all the iterations are finished.

**verify** [bool] If True, return verifiable bound.

**epsilons** [int or Iterable[float]] Either Iterable of contrast levels or number of brightness factors between 1 and 0 that should be tried. Epsilons are one minus the brightness factor. Epsilons are not used if verify = True.

**class** perceptron.benchmarks.**ContrastReductionMetric**(*model=None*, *criterion=None*, *distance=<class 'perceptron.utils.distances.MeanSquaredDistance'>*, *threshold=None*)

Metric that tests models against brightness variations.

**__call__**(*self*, *adv*, *annotation=None*, *unpack=True*, *abort_early=True*, *threshold=1.0*, *epsilons=1000*)

Reduces the contrast of the image until it is misclassified.

**Parameters**

**adv** [*numpy.ndarray*] The original, unperturbed input as a *numpy.ndarray*.

**annotation** [int] The reference label of the original input.

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

**abort_early** [bool] If true, returns when got first adversarial, otherwise returns when all the iterations are finished.

**threshold** [float] Upper bound for contrast factor

**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** perceptron.benchmarks.**MotionBlurMetric**(*model=None*, *criterion=None*, *distance=<class 'perceptron.utils.distances.MeanSquaredDistance'>*, *threshold=None*)

Motion blurs the image until it is misclassified.

**__call__**(*self*, *adv*, *annotation=None*, *unpack=True*, *abort_early=True*, *motion_angle=0*, *epsilons=10000*)

Blurs the image until it is misclassified.

**Parameters**

**adv** [*numpy.ndarray*] The original, unperturbed input as a *numpy.ndarray*.

**annotation** [int] The reference label of the original input.

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

**abort_early** [bool] If true, returns when got first adversarial, otherwise returns when all the iterations are finished.

**motion_angle** [float] Motion angle in degree between 0 and 180.

**epsilons** [int or Iterable[float]] Either Iterable of kernel size that should be tried.

**class** perceptron.benchmarks.**RotationMetric**(*model=None*, *criterion=None*, *distance=<class 'perceptron.utils.distances.MeanSquaredDistance'>*, *threshold=None*)

Metric that tests models against rotations.

**__call__**(*self*, *adv*, *ang_range=None*, *annotation=None*, *unpack=True*, *abort_early=True*, *verify=False*, *epsilons=1000*)

Rotate the image until it is misclassified.

**Parameters**

**adv** [*numpy.ndarray*] The original, unperturbed input as a *numpy.ndarray*.

**ang_range** [(float, float)] Range of angles for rotation metric.

**annotation** [int] The reference label of the original input.

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

**abort_early** [bool] If true, returns when got first adversarial, otherwise returns when all the iterations are finished.

**verify** [bool] if True, return verifiable bound.

**epsilons** [int or Iterable[float]] Either Iterable of rotation angles or number of angles between 1 and 0 that should be tried. Epsilons are one minus the contrast level. Epsilons are not used if verify = True.

**class** perceptron.benchmarks.**SaltAndPepperNoiseMetric**(*model=None*, *criterion=None*, *distance=<class 'perceptron.utils.distances.MeanSquaredDistance'>*, *threshold=None*)

Add salt and pepper noise.

**__call__**(*self*, *adv*, *annotation=None*, *unpack=True*, *abort_early=True*, *epsilons=10000*, *repetitions=10*)

Add salt and pepper noise.

**Parameters**

**adv** [*numpy.ndarray*] The original, unperturbed input as a *numpy.ndarray*.

**annotation** [int] The reference label of the original input.

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

**abort_early** [bool] If true, returns when got first adversarial, otherwise returns when all the iterations are finished.

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

**repetitions** [int] Specifies how often the attack will be repeated.

**class** perceptron.benchmarks.**SpatialMetric**(*model=None*, *criterion=None*, *distance=<class 'perceptron.utils.distances.MeanSquaredDistance'>*, *threshold=None*)

Metric that tests models against spatial transformations.

**__call__**(*self*, *adv*, *annotation=None*, *do_rotations=True*, *do_translations=True*, *x_shift_limits=(-5, 5)*, *y_shift_limits=(-5, 5)*, *angular_limits=None*, *unpack=True*, *abort_early=True*, *verify=False*, *epsilons=1000*)

Spatial transforms the image until it is misclassified.

**Parameters**

**adv** [*numpy.ndarray*] The original, unperturbed input as a *numpy.ndarray*.

**annotation** [int] The reference label of the original input.

**do_rotations** [bool] If False no rotations will be applied to the image (default True).

**do_translations** [bool] If False no translations will be applied to the image (default True).

**x_shift_limits** [(int, int)] Limits for horizontal translations in pixels (default (-5, 5)).

**y_shift_limits** [(int, int)] Limits for vertical translations in pixels (default (-5, 5)).

**angular_limits** [(int, int)] Limits for rotations in degrees (default None).

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

**abort_early** [bool] If true, returns when got first adversarial, otherwise returns when all the iterations are finished.

**verify** [bool] if True, return verifiable bound

**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.

### 2.4.2 Security Metrics

**class** perceptron.benchmarks.**CarliniWagnerL2Metric**(*model=None*, *criterion=None*, *distance=<class 'perceptron.utils.distances.MeanSquaredDistance'>*, *threshold=None*)

The L2 version of C&W attack.

**static lp_distance_and_grad**(*reference*, *other*, *span*)
Calculate L2 distance and gradient.

**class** perceptron.benchmarks.**CarliniWagnerLinfMetric**(*model=None*, *criterion=None*, *distance=<class 'perceptron.utils.distances.MeanSquaredDistance'>*, *threshold=None*)

The L_inf version of C&W attack.

**static lp_distance_and_grad**(*reference*, *other*, *span*)
Calculate L2 distance and gradient.

## 2.5 perceptron.models

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

| | |
|---|---|
| *Model* | Base class to provide metrics with a unified interface to models. |
| *DifferentiableModel* | Base class for differentiable models that provide gradients. |

Provides class to wrap existing models in different frameworks so that they provide a unified API to the benchmarks.

| | |
|---|---|
| *KerasModel* | Create a `Model` instance from a *Keras* model. |
| *PyTorchModel* | Creates a `Model` instance from a *PyTorch* module. |
| *AipModel* | Base class for models hosted on Baidu AIP platform. |
| *AipAntiPornModel* | Create a `Model` instance from an *AipAntiPorn* model. |
| *GoogleCloudModel* | Base class for models in Google Cloud. |
| *GoogleSafeSearchModel* | Create a :class: *Model* instance from a *GoogleSafeSearchModel* model. |
| *GoogleObjectDetectionModel* | Create a :class: *Model* instance from a *GoogleObjectDetectionModel* model. |

| |
|---|
| KerasYOLOv3Model |
| KerasSSD300Model |

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

**class** perceptron.models.**Model**(*bounds*, *channel_axis*, *preprocessing=(0, 1)*)
Base class to provide metrics with a unified interface to models.

**predictions**(*self*, *image*)
Calculate prediction for a single image.

**class** perceptron.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()`.

**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. the input image.

**Parameters**

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

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

**Returns**

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

**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, or with bounding boxes).

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

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

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

**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, or with bounding boxes).

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

### 2.5.1 Wrapper for Classification Models

Provides class to wrap existing models in different frameworks so that they provide a unified API to the benchmarks.

| | |
|---|---|
| *KerasModel* | Create a `Model` instance from a *Keras* model. |
| *PyTorchModel* | Creates a `Model` instance from a *PyTorch* module. |

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

Create a `Model` instance from a *Keras* model.

**Parameters**

**model** [*keras.model.Model*] The *Keras* model that are loaded.

**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 substract 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*)
Get gradients w.r.t. the original image.

**batch_predictions**(*self*, *images*)
Batch prediction of images.

**model_task**(*self*)
Get the task that the model is used for.

**num_classes**(*self*)
Return number of classes.

**predictions_and_gradient**(*self*, *image*, *label*)
Returns both predictions and gradients.

**class** perceptron.models.classification.**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 are loaded.

**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*)
Get gradients w.r.t. the original image.

**batch_predictions**(*self*, *images*)
Batch prediction of images.

**model_task**(*self*)
Get the task that the model is used for.

**num_classes**(*self*)
Return number of classes.

**predictions_and_gradient**(*self*, *image*, *label*)
Returns both predictions and gradients.

### 2.5.2 Wrapper for Object Detection Models

| |
|---|
| KerasYOLOv3Model |
| KerasSSD300Model |

### 2.5.3 Wrapper for Cloud API Models

Provides class to wrap existing models in different frameworks so that they provide a unified API to the benchmarks.

| | |
|---|---|
| *AipModel* | Base class for models hosted on Baidu AIP platform. |
| *AipAntiPornModel* | Create a Model instance from an *AipAntiPorn* model. |
| *GoogleCloudModel* | Base class for models in Google Cloud. |
| *GoogleSafeSearchModel* | Create a :class: *Model* instance from a *GoogleSafeSearchModel* model. |
| *GoogleObjectDetectionModel* | Create a :class: *Model* instance from a *GoogleObjectDetectionModel* model. |

**class** perceptron.models.classification.**AipModel**(*credential*, *bounds=(0, 255)*, *channel_axis=3*, *preprocessing=(0, 1)*)

Base class for models hosted on Baidu AIP platform.

**Parameters**

**credential** [tuple] Tuple of (appId, apiKey, secretKey) for using AIP API.

**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 substract the first element of preprocessing from the input and then divide the input by the second element.

**class** perceptron.models.classification.**AipAntiPornModel**(*credential*, *bounds=(0, 255)*, *channel_axis=3*, *preprocessing=(0, 1)*)

Create a Model instance from an *AipAntiPorn* model.

**Parameters**

**credential** [tuple] Tuple of (appId, apiKey, secretKey) for using AIP API.

**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 substract the first element of preprocessing from the input and then divide the input by the second element.

**model_task**(*self*)
Get the task that the model is used for.

**predictions**(*self*, *image*)
Get prediction for input image

**Parameters**

**image** [*numpy.ndarray*] The input image in [h, n, c] ndarry format.

**Returns**

**list** List of anitporn prediction resutls. Each element is a dictionary containing: {'class_name', 'probability'}

**class** perceptron.models.classification.**GoogleCloudModel**(*bounds=(0, 255)*, *channel_axis=3*, *preprocessing=(0, 1)*)

Base class for models in Google Cloud.

**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 substract the first element of preprocessing from the input and then divide the input by the second element.

### Notes

To use google cloud vision models, you need to install its package *pip instlal –upgrade google-cloud-vision*.

**class** perceptron.models.classification.**GoogleSafeSearchModel**(*bounds=(0, 255)*, *channel_axis=3*, *preprocessing=(0, 1)*)

Create a :class: *Model* instance from a *GoogleSafeSearchModel* model.

**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 substract the first element of preprocessing from the input and then divide the input by the second element.

**model_task**(*self*)
Get the task that the model is used for.

**predictions**(*self*, *image*)
Get prediction for input image.

**Parameters**

**image** [*numpy.ndarray*] The input image in [h, n, c] ndarry format.

**Returns**

**list** List of prediction resutls. Each element is a dictionary containing: {'adult', 'medical', 'racy', 'spoof', 'violence'}.

**class** perceptron.models.classification.**GoogleObjectDetectionModel**(*bounds=(0, 255)*, *channel_axis=3*, *preprocessing=(0, 1)*)

Create a :class: *Model* instance from a *GoogleObjectDetectionModel* model.

**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 substract the first element of preprocessing from the input and then divide the input by the second element.

**model_task**(*self*)

Get the task that the model is used for.

**predictions**(*self*, *image*)

Get detection result for input image.

**Parameters**

**image** [*numpy.ndarray*] The input image in [h, n, c] ndarry format.

**Returns**

**list** List of batch prediction resutls. Each element is a dictionary containing: {'name', 'score', 'mid', 'bounding_poly'}.

## 2.6 perceptron.utils.adversarial

Provides a class that represents an adversarial example.

| | |
|---|---|
| *Adversarial* | Defines the base class of an adversarial that should be found and stores the result. |
| ClsAdversarial | Defines an adversarial that should be found and stores the result. |
| DetAdversarial | Defines an adversarial that should be found and stores the result. |

**class** perceptron.utils.adversarial.**Adversarial**(*model*, *criterion*, *original_image*, *original_pred=None*, *threshold=None*, *distance=<class 'perceptron.utils.distances.MeanSquaredDistance'>*, *verbose=False*)

Defines the base class of 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 evaluated against 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_pred** [int(ClsAdversarial) or dict(DetAdversarial)] The ground-truth predictions 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.

**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, 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.

**bounds**(*self*)
Return bounds of model.

**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.

**in_bounds** (*self*, *input_*)
Check if input is in bounds.

**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**

**——-**

**:class:'Distance'** The distance between the given image and the original image.

**num_classes** (*self*)
Return number of classes.

**original_image**
The original input.

**original_pred**
The original label.

**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.

**reset_distance_dtype** (*self*)
Reset the dtype of Distance.

**set_distance_dtype** (*self*, *dtype*)
Set the dtype of Distance.

**target_class** (*self*)
Interface to criterion.target_class for attacks.

**verifiable_bounds**
The verifiable bounds obtained so far.

### 2.6.1 Adversarial for Classification Models

Provides a class that represents an adversarial example for image classfication tasks.

**class** perceptron.utils.adversarial.classification.**ClsAdversarial**(*model*, *criterion*, *original_image*, *original_pred*, *threshold=None*, *distance=<class 'perceptron.utils.distances.MeanSquaredDist*
*verbose=False*)

Defines an adversarial that should be found and stores the result.

**backward**(*self*, *gradient*, *image=None*, *strict=True*)
Interface for model.backward for attacks.

**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. Deefaults to the original label

**strict** [bool] Controls if the bounds for the pixel values should be checked.

**model_task**(*self*)
Interface to model.model_task for attacks.

**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.

### 2.6.2 Adversarial for Object Detection Models

Provides a class that represents an adversarial example for object detection tasks.

**class** perceptron.utils.adversarial.detection.**DetAdversarial**(*model*, *criterion*, *original_image*, *original_pred*, *threshold=None*, *distance=<class 'perceptron.utils.distances.MeanSquaredDistance'>*, *verbose=False*)

Defines an adversarial that should be found and stores the result.

**backward**(*self*, *target_class*, *image=None*, *strict=True*)

Interface to model.backward.

**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. Deefaults to the original label

**strict** [bool] Controls if the bounds for the pixel values should be checked.

**model_task**(*self*)

Interface to model.model_task for attacks.

**predictions_and_gradient**(*self*, *image=None*, *annotation=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.

## 2.7 perceptron.utils.criteria

Provides class to wrap all adversarial criterions so that attacks has uniform API access.

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

Continued on next page

Table 14 – continued from previous page

| | |
|---|---|
| `OriginalClassProbability` | Defines adversarials as images for which the probability of original class is below a given threshold. |
| `TargetClassProbability` | Defines adversarials as images for which the probability of a given target class is above a given threshold. |
| `MisclassificationAntiPorn` | Defines adversarials as image for which the probability of being *normal* is larger than the probability of being *porn*. |
| `MisclassificationSafeSearch` | Defines adversarials as image for which the probability of being *unsafe* is lower than a threshold. |
| `TargetClassMiss` | Defines adversarials as images for which the target class is not in the detection result. |
| `TargetClassMissGoogle` | Defines adversarials as images for which the target class is not in the Google object detection result. |
| `WeightedAP` | Defines adversarials as weighted AP value larger than given threshold. |

### 2.7.1 Detailed description

**class** perceptron.utils.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 shoud be subclassed when implementing new criteria. Subclasses must implement is_adversarial.

**is_adversarial**(*self*, *predictions*, *ground_truth*)

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

**name**(*self*)

Returns a human readable name.

**class** perceptron.utils.criteria.**CombinedCriteria**(**criteria*)

Meta criterion that combines several criteria into a new one.

**Parameters**

***criteria** [variable length list of *Criterion* instances] List of sub-criteria that will be combined.

#### Notes

This class uses lazy evaluation of the criteria in the order they are passed to the constructor.

**is_adversarial**(*self*, *predictions*, *ground_truth*)

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

**name**(*self*)

Concatenates the names of the given criteria in alphabetical order.

#### Criterial for Classification Models

Provide base classes that define what is adversarial.

**class** perceptron.utils.criteria.classification.**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 shoud be subclassed when implementing new criteria. Subclasses must implement is_adversarial.

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

**name**(*self*)
Returns a human readable name.

**class** perceptron.utils.criteria.classification.**Misclassification**
Defines adversarials as images for which the predicted class is not the original class.

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

**name**(*self*)
Return criterion name.

**class** perceptron.utils.criteria.classification.**ConfidentMisclassification**(*threshold*)
Defines adversarials as images for which the probability of any class other than the original is above a given threshold.

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

**name**(*self*)
Return criterion name.

**class** perceptron.utils.criteria.classification.**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.

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

**name**(*self*)
Return criterion name.

**class** perceptron.utils.criteria.classification.**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.

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

**name**(*self*)
Return criterion name.

**target_class**(*self*)
Return target class.

**class** perceptron.utils.criteria.classification.**OriginalClassProbability**(*p*)
Defines adversarials as images for which the probability of original class is below a given threshold.

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

**name**(*self*)
Return criterion name.

**class** perceptron.utils.criteria.classification.**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.

**name**(*self*)
Return criterion name.

**target_class**(*self*)
Return target class

### Criterial for Object Detection Models

Provide base classes that define what is an adversarial for object detection models.

**class** perceptron.utils.criteria.detection.**TargetClassMiss**(*target_class*)
Defines adversarials as images for which the target class is not in the detection result.

**__init__**(*self*, *target_class*)
Initialize self. See help(type(self)) for accurate signature.

**is_adversarial**(*self*, *predictions*, *annotation*)
Decides if predictions for an image are adversarial.

**name**(*self*)
Return ctiterion name.

**target_class**(*self*)
Return target class.

## 2.8 perceptron.utils.distances

Provides classes to measure the distance between two images.

| | |
|---|---|
| *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. |
| *MSE* | alias of *perceptron.utils.distances.MeanSquaredDistance* |
| *MAE* | alias of *perceptron.utils.distances.MeanAbsoluteDistance* |
| *Linf* | alias of *perceptron.utils.distances.Linfinity* |

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

Calculates the mean squared error between two images.

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

Calculates the mean absolute error between two images.

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

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

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

Calculates the L0 norm of the difference between two images.

perceptron.utils.distances.**MSE**

alias of *perceptron.utils.distances.MeanSquaredDistance*

perceptron.utils.distances.**MAE**

alias of *perceptron.utils.distances.MeanAbsoluteDistance*

perceptron.utils.distances.**Linf**

alias of *perceptron.utils.distances.Linfinity*

CHAPTER 3

# Indices and tables

- genindex
- modindex
- search

# Python Module Index

## p

# Index

## Symbols



## A



## B

## C



## D



## G



## H



## I



## K



## L

## R



## S



## T



## V