Augmentor Documentation

Release 0.2.6

Marcus D. Bloice

Jul 27, 2019

User Guide

Augmentor is a Python package designed to aid the augmentation and artificial generation of image data for machine learning tasks. It is primarily a data augmentation tool, but will also incorporate basic image pre-processing functionality.

Tip: A Julia version of the package is also being actively developed. If you prefer to use Julia, you can find it here.

The documentation is organised as follows:

CHAPTER 1

Main Features

In this section we will describe the main features of Augmentor with example code and output.

Augmentor is software package for image augmentation with an emphasis on providing operations that are typically used in the generation of image data for machine learning problems.

In principle, Augmentor consists of a number of classes for standard image manipulation functions, such as the `Rotate` class or the `Crop` class. You interact and use these classes using a large number of convenience functions, which cover most of the functions you might require when augmenting image datasets for machine learning problems.

Because image augmentation is often a multi-stage procedure, Augmentor uses a **pipeline**-based approach, where **operations** are added sequentially in order to generate a pipeline. Images are then passed through this pipeline, where each operation is applied to the image as it passes through.

Also, Augmentor applies operations to images **stochastically** as they pass through the pipeline, according to a user-defined probability value for each operation.

Therefore every operation has at minimum a probability parameter, which controls how likely the operation will be applied to each image that is seen as the image passes through the pipeline. Take for example a rotate operation, which is defined as follows:

```
rotate(probability=0.5, max_left_rotation=5, max_right_rotation=10)
```

The `probability` parameter controls how often the operation is applied. The `max_left_rotation` and `max_right_rotation` controls the degree by which the image is rotated, **if** the operation is applied. The value, in this case between -5 and 10 degrees, is chosen at random.

Therefore, Augmentor allows you to create an augmentation pipeline, which chains together operations that are applied stochastically, where the parameters of each of these operations are also chosen at random, within a range specified by the user. This means that each time an image is passed through the pipeline, a different image is returned. Depending on the number of operations in the pipeline, and the range of values that each operation has available, a very large amount of new image data can be created in this way.

All functions described in this section are made available by the Pipeline object. To begin using Augmentor, you always create a new Pipeline object by instantiating it with a path to a set of images or image that you wish to augment:

```
>>> import Augmentor
>>> p = Augmentor.Pipeline("/path/to/images")
Initialised with 100 images found in selected directory.
```

You can now add operations to this pipeline using the `p` Pipeline object. For example, to add a rotate operation:

```
>>> p.rotate(probability=1.0, max_left_rotation=5, max_right_rotation=10)
```

All pipeline operations have at least a probability parameter.

To see the status of the current pipeline:

```
>>> p.status()
There are 1 operation(s) in the current pipeline.
Index 0:
    Operation RotateRange (probability: 1):
        Attribute: max_right_rotation (10)
        Attribute: max_left_rotation (-5)
        Attribute: probability (1)

There are 1 image(s) in the source directory.
Dimensions:
    Width: 400 Height: 400
Formats:
    PNG
```

You can remove operations using the `remove_operation(index)` function and the appropriate `index` indicator from above.

Full documentation of all functions and operations can be found in the auto-generated documentation. This guide suffice as a rough guide to the major features of the package, however.

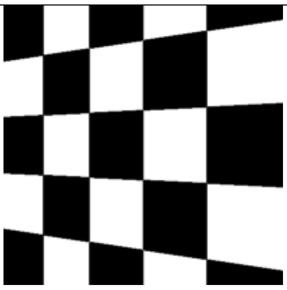
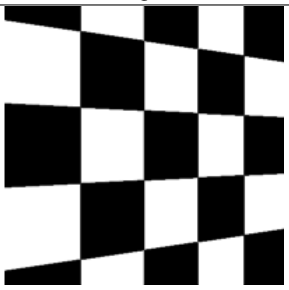
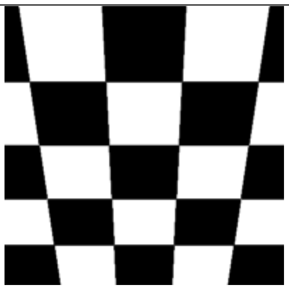
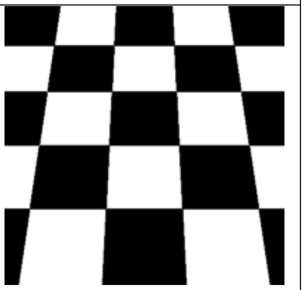
1.1 Perspective Skewing

Perspective skewing involves transforming the image so that it appears that you are looking at the image from a different angle.

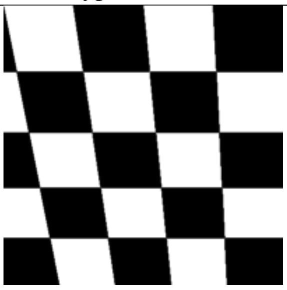
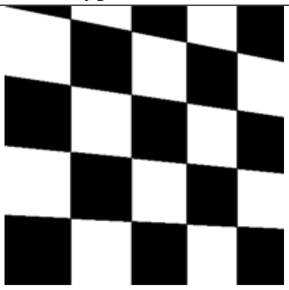
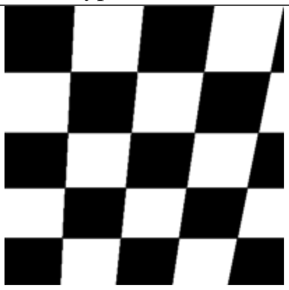
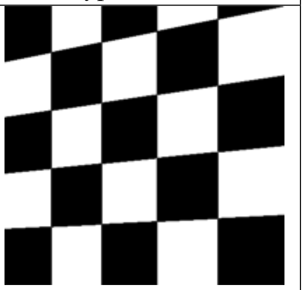
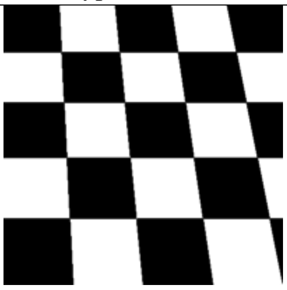
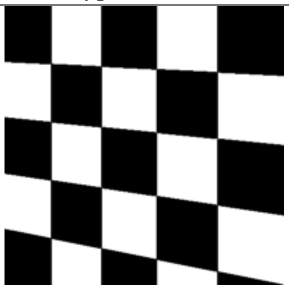
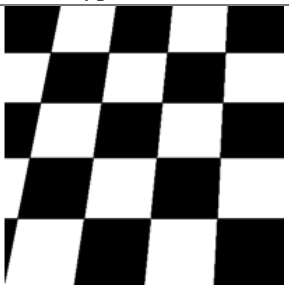
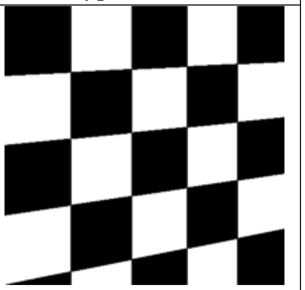
The following main functions are used for skewing:

- `skew_tilt()`
- `skew_left_right()`
- `skew_top_bottom()`
- `skew_corner()`
- `skew()`

To skew or tilt an image either left, right, forwards, or backwards, use the `skew_tilt` function. The image will be skewed by a random amount in the following directions:

Skew Tilt Left	Skew Tilt Right	Skew Tilt Forward	Skew Tilt Backward
			

Or, to skew an image by a random corner, use the `skew_corner()` function. The image will be skewed using one of the following 8 skew types:

Skew Type 0	Skew Type 1	Skew Type 2	Skew Type 3
			
Skew Type 4	Skew Type 5	Skew Type 6	Skew Type 7
			

If you only wish to skew either left or right, use `skew_left_right()`. To skew only forwards or backwards, use `skew_top_bottom()`.

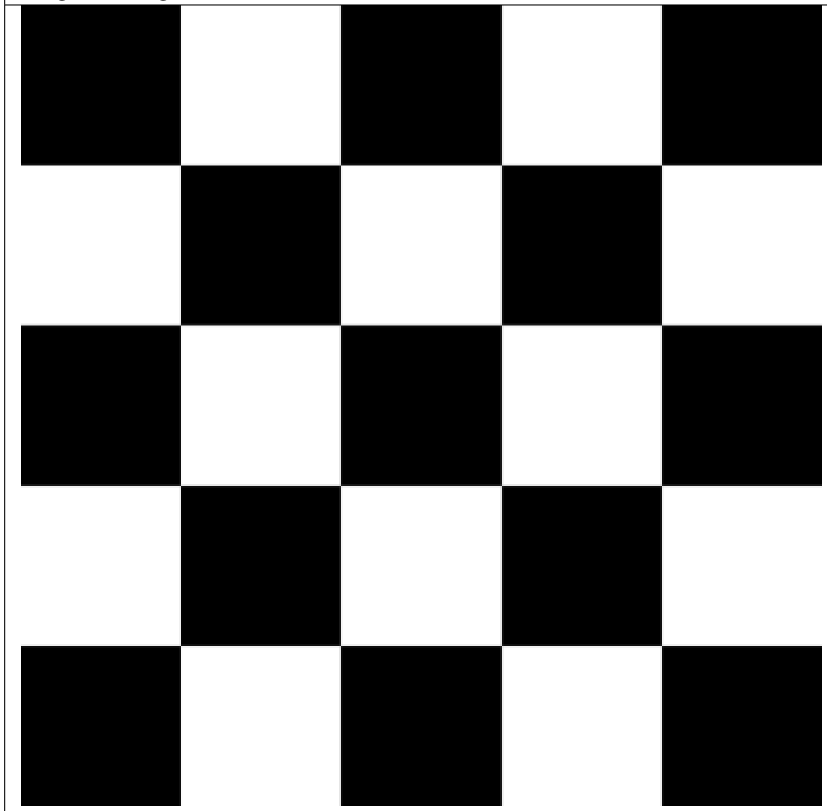
The function `skew()` will skew your image in a random direction of the 12 directions shown above.

1.2 Elastic Distortions

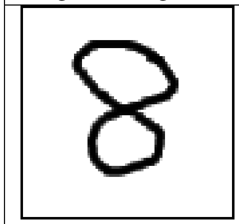
Elastic distortions allow you to make distortions to an image while maintaining the image's aspect ratio.

- `random_distortion()`

Here, we have taken a sample image and generated 50 samples, with a grid size of 16 and a distortion magnitude of 8:

Original Image	Random distortions applied
	

To highlight how this might be useful in a real-world scenario, here is the distort function being applied to a single image of a figure 8.

Original Image	Random distortions applied
	

Realistic new samples can be created using this method.

See the auto-generated documentation for more details regarding this function's parameters.

1.3 Rotating

Rotating can be performed in a number of ways. When rotating by modulo 90, the image is simply rotated and saved. To rotate by arbitrary degrees, then a crop is taken from the centre of the newly rotated image.

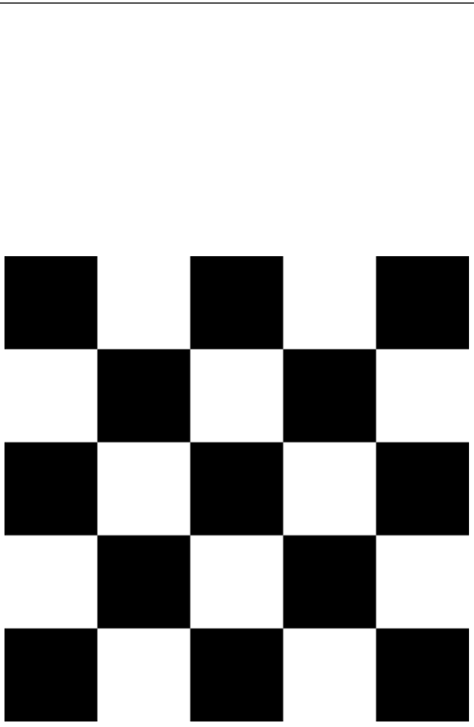
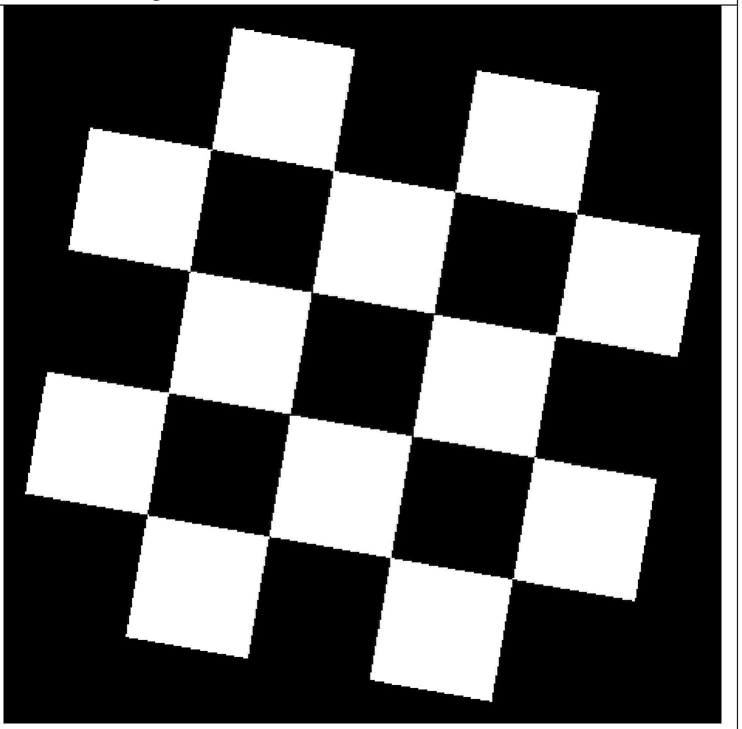
Rotate functions that are available are:

- `rotate()`

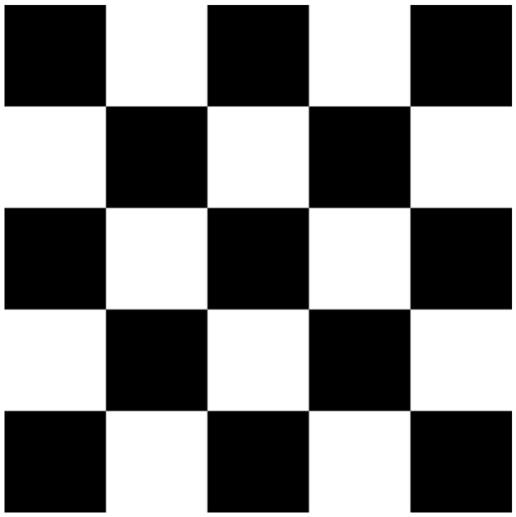
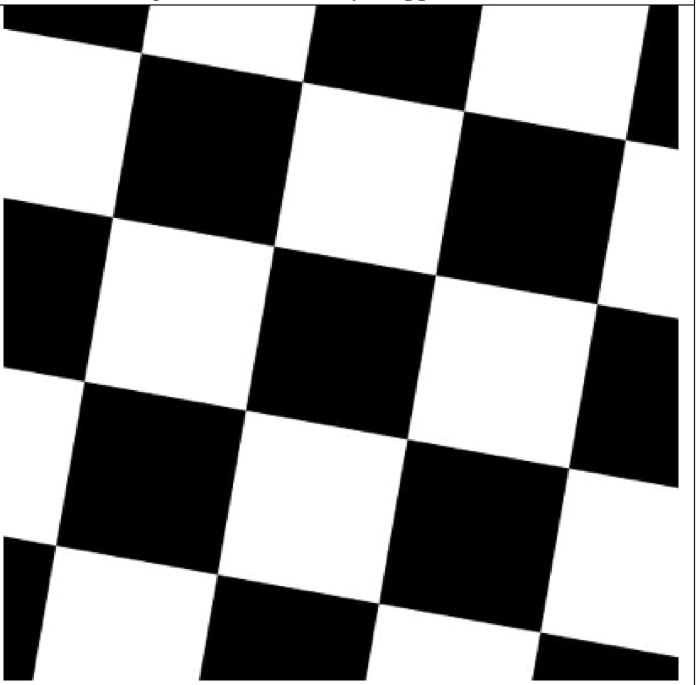
- `rotate90()`
- `rotate180()`
- `rotate270()`
- `rotate_random_90()`

Most of these methods are self-explanatory. The `rotate_random_90()` function will rotate the image by either 90, 180, or 270 degrees.

However, the `rotate()` warrants more discussion and will be described here. When an image is rotated, and it is not a multiple of 90 degrees, the image must either be stretched to accommodate a now larger image, or some of the image must be cut, as demonstrated below:

Original Image	Rotated 10 degrees
	

As can be seen above, an arbitrary, non-modulo 90, rotation will unfortunately result in the image being padded in each corner. To alleviate this, Augmentor's default behaviour is to crop the image and retain the largest crop possible while maintaining the image's aspect ratio:

Original Image	Rotated 10 degrees, automatically cropped
	

This will, of course, result in the image being zoomed in. For smaller rotations of between -5 and 5 degrees, this zoom effect is not particularly drastic.

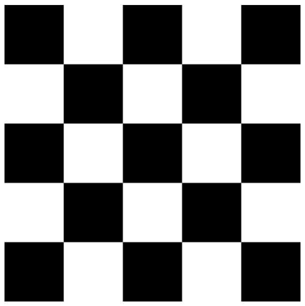
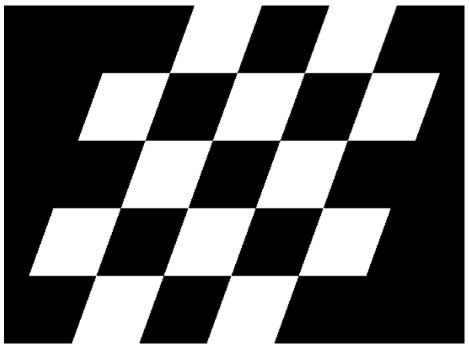
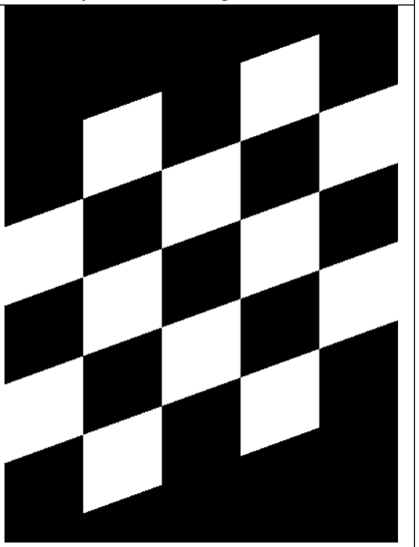
1.4 Shearing

Shearing tilts an image along one of its sides. The can be in the x-axis or y-axis direction.

Functions available for shearing are:

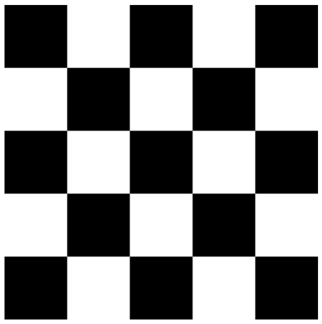
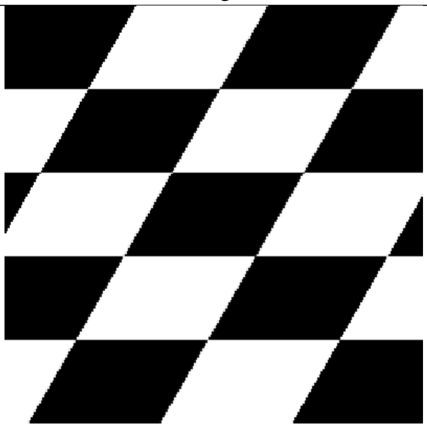
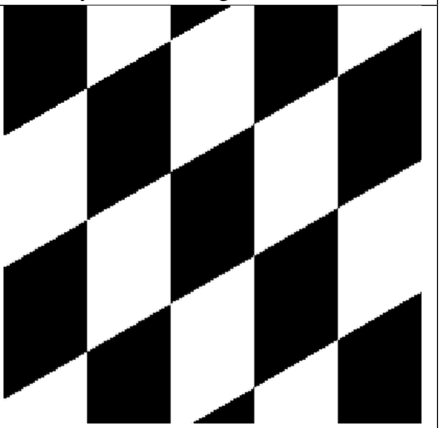
- `shear()`

If you shear in the x or y axis, you will normally get images that look as follows:

Original image	Shear (x-axis) 20 degrees	Shear (y-axis) 20 degrees
		

However, as with rotations, you are left with image that are either larger in size, or are cropped to the original size but contain padding in at the sides of the images.

Augmentor automatically crops the largest area possible before returning the image, as follows:

Original image	Shear (x-axis) 20 degrees	Shear (y-axis) 20 degrees
		

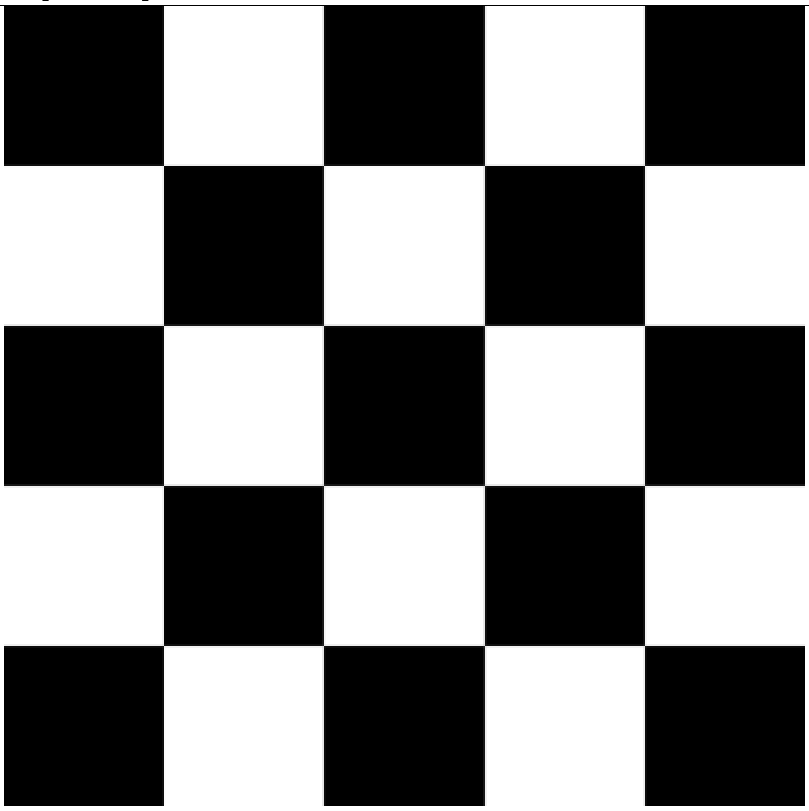
You can shear by random amounts, a fixed amount, in random directions, or in a fixed direction. See the auto-generated documentation for more details.

1.5 Cropping

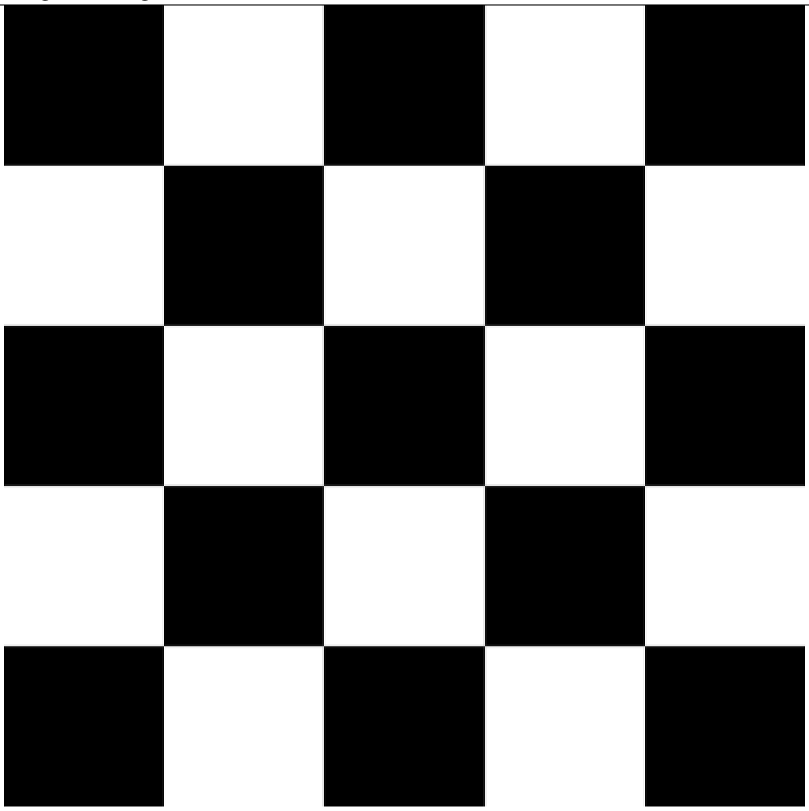
Cropping functions which are available are:

- `crop_centre()`
- `crop_by_size()`
- `crop_random()`

The `crop_random()` function warrants further explanation. Here a region of a size specified by the user is cropped at random from the original image:

Original image	Random crops
	

You could combine this with a resize operation, so that the images returned are the same size as the images of the original, pre-augmented dataset:

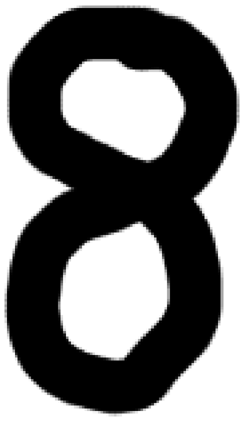
Original image	Random crops + resize operation
	

1.6 Mirroring

The following functions are available for mirroring images (translating them through the x any y axes):

- `flip_left_right()`
- `flip_top_bottom()`
- `flip_random()`

Of these, `flip_random()` can be used in situations where mirroring through both axes may make sense. We may, for example, combine random mirroring, with random distortions, to create new data:

Original image	Random mirroring + random distortions
	

1.7 Notes

Checkerboard image obtained from WikiMedia Commons and is in the public domain. See https://commons.wikimedia.org/wiki/File:Checkerboard_pattern.svg

Skin lesion image obtained from the ISIC Archive:

- Image id: 5436e3adbae478396759f0f1
- Image name: ISIC_0000017.jpg
- Download: https://isic-archive.com:443/api/v1/image/5436e3adbae478396759f0f1/download

See https://isic-archive.com/#images for further details.

CHAPTER 2

Installation

Installation is via `pip`:

```
pip install Augmentor
```

If you have to use `sudo` it is recommended that you use the `-H` flag:

```
sudo -H pip install Augmentor
```

2.1 Requirements

Augmentor requires `Pillow` and `tqdm`. Note that Pillow is a fork of PIL, but both packages cannot exist simultaneously. Uninstall PIL before installing Pillow.

2.2 Building

If you prefer to build the package from source, first clone the repository:

```
git clone https://github.com/mdbloice/Augmentor.git
```

Then enter the `Augmentor` directory and build the package:

```
cd Augmentor
python setup.py install
```

Alternatively you can first run `python setup.py build` followed by `python setup.py install`. This can be useful for debugging.

Attention: If you are compiling from source you may need to compile the dependencies also, including Pillow. On Linux this means having libpng (`libpng-dev`) and zlib (`zlib1g-dev`) installed.

CHAPTER 3

Usage

Here we describe the general usage of Augmentor.

3.1 Getting Started

To use Augmentor, the following general procedure is followed:

1. You instantiate a `Pipeline` object pointing to a directory containing your initial image data set.
2. You define a number of operations to perform on this data set using your `Pipeline` object.
3. You execute these operations by calling the `Pipeline`'s `sample()` method.

We will go through each of these steps in order in the proceeding 3 sub-sections.

3.1.1 Step 1: Create a New Pipeline

Let us first create an empty pipeline. In other words, to begin any augmentation task, you must first initialise a `Pipeline` object, that points to a directory where your original image dataset is stored:

```
>>> import Augmentor
>>> p = Augmentor.Pipeline("/path/to/images")
Initialised with 100 images found in selected directory.
```

The variable `p` now contains a `Pipeline` object, and has been initialised with a list of images found in the source directory.

3.1.2 Step 2: Add Operations to the Pipeline

Once you have created a `Pipeline`, `p`, we can begin by adding operations to `p`. For example, we shall begin by adding a `rotate()` operation:

```
>>> p.rotate(probability=0.7, max_left_rotation=10, max_right_rotation=10)
```

In this case, we have added a *`rotate()`* operation, that will execute with a probability of 70%, and have defined the maximum range by which an image will be rotated from between -10 and 10 degrees.

Next, we add a further operation, in this case a *`zoom()`* operation:

```
>>> p.zoom(probability=0.3, min_factor=1.1, max_factor=1.6)
```

This time, we have specified that we wish the operation to be applied with a probability of 30%, while the scale should be randomly selected from between 1.1 and 1.6

3.1.3 Step 3: Execute and Sample From the Pipeline

Once you have added the operations that you require, you can generate new, augmented data by using the *`sample()`* function and specify the number of images you require, in this case 10,000:

```
>>> p.sample(10000)
```

A progress bar will appear providing a number of metrics while your samples are generated. Newly generated, augmented images will by default be saved into an directory named **output**, relative to the directory which contains your initial image data set.

Hint: A full list of operations can be found in the *`Operations`* module documentation.

CHAPTER 4

Examples

A number of typical usage scenarios are described here.

Note: A full list of operations can be found in the `Operations` module documentation.

4.1 Initialising a pipeline

```
import Augmentor

path_to_data = "/home/user/images/dataset1/"

# Create a pipeline
p = Augmentor.Pipeline(path_to_data)
```

4.2 Adding operations to a pipeline

```
# Add some operations to an existing pipeline.

# First, we add a horizontal flip operation to the pipeline:
p.flip_left_right(probability=0.4)

# Now we add a vertical flip operation to the pipeline:
p.flip_top_bottom(probability=0.8)

# Add a rotate90 operation to the pipeline:
p.rotate90(probability=0.1)
```

4.3 Executing a pipeline

```
# Here we sample 100,000 images from the pipeline.

# It is often useful to use scientific notation for specify
# large numbers with trailing zeros.
num_of_samples = int(1e5)

# Now we can sample from the pipeline:
p.sample(num_of_samples)
```

CHAPTER 5

Extending Augmentor

Extending Augmentor to add new functionality is quite simple, and is performed in two steps:

1) Create a custom class which subclasses from the `Operation` base class, and
2) Add an object of your new class to the pipeline using the `add_operation()` function.

This allows you to add custom functionality and extend Augmentor at run-time. Of course, if you have written an operation that may be of benefit to the community, you can make a pull request on the GitHub repository.

The following sections describe extending Augmentor in two steps. Step 1 involves creating a new `Operation` subclass, and step 2 involves using an object of your new custom operation in a pipeline.

5.1 Step 1: Create a New Operation Subclass

To create a custom operation and extend Augmentor:

1) You create a new class that inherits from the `Operation` base class.
2) You must overload the `perform_operation()` method belonging to the superclass.
3) You must call the superclass's `__init__()` constructor.
4) You must return an object of type `PIL.Image`.

For example, to add a new operation called `FoldImage`, you would write this code:

```
# Create your new operation by inheriting from the Operation superclass:
class FoldImage(Operation):
    # Here you can accept as many custom parameters as required:
    def __init__(self, probability, num_of_folds):
        # Call the superclass's constructor (meaning you must
        # supply a probability value):
        Operation.__init__(self, probability)
        # Set your custom operation's member variables here as required:
        self.num_of_folds = num_of_folds
```

(continues on next page)

(continued from previous page)

```
    # Your class must implement the perform_operation method:
    def perform_operation(self, image):
        # Start of code to perform custom image operation.
        for fold in range(self.num_of_folds):
            pass
        # End of code to perform custom image operation.

        # Return the image so that it can further processed in the pipeline:
        return image
```

You have seen that you need to implement the *perform_operation()* function and you must call the superclass's constructor which requires a `probability` value to be set. Ensure you return a PIL Image as a return value.

If you wish to make these changes permanent, place your code in the *Operations* **module**.

Hint: You can also overload the superclass's `__str__()` function to return a custom string for the object's description text. This is useful for some methods that display information about the operation, such as the *status()* method.

5.2 Step 2: Add an Object to the Pipeline Manually

Once you have a new operation which is of type *Operation*, you can add an object of you new operation to an existing pipeline.

```
# Instantiate a new object of your custom operation
fold = Fold(probability = 0.75, num_of_folds = 4)

# Add this to the current pipeline
p.add_operation(fold)

# Executed the pipeline as normal, and your custom operation will be executed
p.sample(1000)
```

As you can see, adding custom operations at run-time is possible by subclassing the *Operation* class and adding an object of this class to the pipeline manually using the *add_operation()* function.

5.3 Using non-PIL Image Objects

Images can be converted to their raw formats for custom operations, for example by using NumPy:

```
import numpy

# Custom class declaration

def perform_operation(image):

    image_array = numpy.array(image).astype('uint8')

    # Perform your custom operations here
```

(continues on next page)

(continued from previous page)

```
    image = PIL.Image.fromarray(image_array)

    return image
```

CHAPTER 6

Auto Generated Documentation

The Augmentor image augmentation library.

Augmentor is a software package for augmenting image data. It provides a number of utilities that aid augmentation in a automated manner. The aim of the package is to make augmentation for machine learning tasks less prone to error, more reproducible, more efficient, and easier to perform.

6.1 Module by Module Documentation

6.1.1 Documentation of the Pipeline module

The Pipeline module is the user facing API for the Augmentor package. It contains the `Pipeline` class which is used to create pipeline objects, which can be used to build an augmentation pipeline by adding operations to the pipeline object.

For a good overview of how to use Augmentor, along with code samples and example images, can be seen in the *Main Features* section.

class `Augmentor.Pipeline.`**`DataFramePipeline`**(*source_dataframe*, *image_col*, *category_col*, *output_directory=u'output'*, *save_format=None*)

Create a new Pipeline object pointing to dataframe containing the paths to your original image dataset.

Create a new Pipeline object, using the `source_dataframe` and the columns `image_col` for the path of the image and `category_col` for the name of the cateogry

Parameters

- **`source_dataframe`** – A Pandas DataFrame where the images are located
- **`output_directory`** – Specifies where augmented images should be saved to the disk. Default is the absolute path
- **`save_format`** – The file format to use when saving newly created, augmented images. Default is JPEG. Legal options are BMP, PNG, and GIF.

Returns A `Pipeline` object.

class `Augmentor.Pipeline.`**`DataPipeline`**(*images*, *labels=None*)

The DataPipeline used to create augmented data that is not read from or saved to the hard disk. The class is provides beta functionality and will be incorporated into the standard Pipeline class at a later date.

Its main purpose is to provide functionality for augmenting images that have multiple masks.

See https://github.com/mdbloice/Augmentor/blob/master/notebooks/Multiple-Mask-Augmentation.ipynb for example usage.

DataPipeline objects are initialised by passing images and their corresponding masks (grouped as lists) along with an optional list of labels. If labels are provided, the augmented images and its corresponding label are returned, otherwise only the images are returned. Image data is returned in array format.

The images and masks that are passed can be of differing formats and have differing numbers of channels. For example, the ground truth data can be 3 channel RGB, while its mask images can be 1 channel monochrome.

`augmentor_images`

`generator`(*batch_size=1*)

`labels`

`sample`(*n*)

class `Augmentor.Pipeline.`**`Pipeline`**(*source_directory=None*, *output_directory=u'output'*, *save_format=None*)

The Pipeline class handles the creation of augmentation pipelines and the generation of augmented data by applying operations to this pipeline.

Create a new Pipeline object pointing to a directory containing your original image dataset.

Create a new Pipeline object, using the `source_directory` parameter as a source directory where your original images are stored. This folder will be scanned, and any valid file files will be collected and used as the original dataset that should be augmented. The scan will find any image files with the extensions JPEG/JPG, PNG, and GIF (case insensitive).

Parameters

- **`source_directory`** – A directory on your filesystem where your original images are stored.
- **`output_directory`** – Specifies where augmented images should be saved to the disk. Default is the directory **output** relative to the path where the original image set was specified. If it does not exist it will be created.
- **`save_format`** – The file format to use when saving newly created, augmented images. Default is JPEG. Legal options are BMP, PNG, and GIF.

Returns A `Pipeline` object.

`add_further_directory`(*new_source_directory*, *new_output_directory=u'output'*)

Add a further directory containing images you wish to scan for augmentation.

Parameters

- **`new_source_directory`** (`String`) – The directory to scan for images.
- **`new_output_directory`** (`String`) – The directory to use for outputted, augmented images.

Returns None

add_operation(*operation*)

Add an operation directly to the pipeline. Can be used to add custom operations to a pipeline.

To add custom operations to a pipeline, subclass from the Operation abstract base class, overload its methods, and insert the new object into the pipeline using this method.

See also:

The *Operation* class.

Parameters **operation** (`Operation`) – An object of the operation you wish to add to the pipeline. Will accept custom operations written at run-time.

Returns None

black_and_white(*probability*, *threshold=128*)

Convert images to black and white. In other words convert the image to use a 1-bit, binary palette. The threshold defaults to 128, but can be controlled using the `threshold` parameter.

See also:

The *greyscale()* function.

Parameters

- **probability** (*Float*) – A value between 0 and 1 representing the probability that the operation should be performed. For resizing, it is recommended that the probability be set to 1.
- **threshold** (*Integer*) – A value between 0 and 255 which controls the threshold point at which each pixel is converted to either black or white. Any values above this threshold are converted to white, and any values below this threshold are converted to black.

Returns None

static categorical_labels(*numerical_labels*)

Return categorical labels for an array of 0-based numerical labels.

Parameters **numerical_labels** (*Array-like list.*) – The numerical labels.

Returns The categorical labels.

crop_by_size(*probability*, *width*, *height*, *centre=True*)

Crop an image according to a set of dimensions.

Crop each image according to `width` and `height`, by default at the centre of each image, otherwise at a random location within the image.

See also:

See *crop_random()* to crop a random, non-centred area of the image.

If the crop area exceeds the size of the image, this function will crop the entire area of the image.

Parameters

- **probability** (*Float*) – The probability that the function will execute when the image is passed through the pipeline.
- **width** (*Integer*) – The width of the desired crop.
- **height** (*Integer*) – The height of the desired crop.

- **centre** (*Boolean*) – If **True**, crops from the centre of the image, otherwise crops at a random location within the image, maintaining the dimensions specified.

Returns None

crop_centre(*probability*, *percentage_area*, *randomise_percentage_area=False*)

Crops the centre of an image as a percentage of the image's area.

Parameters

- **probability** (*Float*) – The probability that the function will execute when the image is passed through the pipeline.
- **percentage_area** (*Float*) – The area, as a percentage of the current image's area, to crop.
- **randomise_percentage_area** (*Boolean*) – If True, will use `percentage_area` as an upper bound and randomise the crop from between 0 and `percentage_area`.

Returns None

crop_random(*probability*, *percentage_area*, *randomise_percentage_area=False*)

Crop a random area of an image, based on the percentage area to be returned.

This function crops a random area from an image, based on the area you specify using `percentage_area`.

Parameters

- **probability** (*Float*) – The probability that the function will execute when the image is passed through the pipeline.
- **percentage_area** (*Float*) – The area, as a percentage of the current image's area, to crop.
- **randomise_percentage_area** (*Boolean*) – If True, will use `percentage_area` as an upper bound and randomise the crop from between 0 and `percentage_area`.

Returns None

flip_left_right(*probability*)

Flip (mirror) the image along its horizontal axis, i.e. from left to right.

See also:

The *flip_top_bottom()* function.

Parameters **probability** (*Float*) – A value between 0 and 1 representing the probability that the operation should be performed.

Returns None

flip_random(*probability*)

Flip (mirror) the image along **either** its horizontal or vertical axis.

This function mirrors the image along either the horizontal axis or the vertical access. The axis is selected randomly.

Parameters **probability** (*Float*) – A value between 0 and 1 representing the probability that the operation should be performed.

Returns None

flip_top_bottom(*probability*)

Flip (mirror) the image along its vertical axis, i.e. from top to bottom.

See also:

The `flip_left_right()` function.

Parameters **probability** (*Float*) – A value between 0 and 1 representing the probability that the operation should be performed.

Returns None

gaussian_distortion(*probability*, *grid_width*, *grid_height*, *magnitude*, *corner*, *method*, *mex=0.5*, *mey=0.5*, *sdx=0.05*, *sdy=0.05*)

Performs a random, elastic gaussian distortion on an image.

This function performs a randomised, elastic gaussian distortion controlled by the parameters specified. The grid width and height controls how fine the distortions are. Smaller sizes will result in larger, more pronounced, and less granular distortions. Larger numbers will result in finer, more granular distortions. The magnitude of the distortions can be controlled using magnitude. This can be random or fixed.

Good values for parameters are between 2 and 10 for the grid width and height, with a magnitude of between 1 and 10. Using values outside of these approximate ranges may result in unpredictable behaviour.

Parameters

- **probability** (*Float*) – A value between 0 and 1 representing the probability that the operation should be performed.
- **grid_width** (*Integer*) – The number of rectangles in the grid's horizontal axis.
- **grid_height** (*Integer*) – The number of rectangles in the grid's vertical axis.
- **magnitude** (*Integer*) – The magnitude of the distortions.
- **corner** (*String*) – which corner of picture to distort. Possible values: "bell"(circular surface applied), "ul"(upper left), "ur"(upper right), "dl"(down left), "dr"(down right).
- **method** (*String*) – possible values: "in"(apply max magnitude to the chosen corner), "out"(inverse of method in).
- **mex** (*Float*) – used to generate 3d surface for similar distortions. Surface is based on normal distribution.
- **mey** (*Float*) – used to generate 3d surface for similar distortions. Surface is based on normal distribution.
- **sdx** (*Float*) – used to generate 3d surface for similar distortions. Surface is based on normal distribution.
- **sdy** (*Float*) – used to generate 3d surface for similar distortions. Surface is based on normal distribution.

Returns None

For values `mex`, `mey`, `sdx`, and `sdy` the surface is based on the normal distribution:

$$e^{-\left(\frac{(x-\text{mex})^2}{\text{sdx}}+\frac{(y-\text{mey})^2}{\text{sdy}}\right)}$$

generator_threading_tests(*batch_size*)

generator_threading_tests_with_matrix_data(*images*, *label*)

get_ground_truth_paths()

Returns a list of image and ground truth image path pairs. Used for verification purposes to ensure the ground truth images match to the images containing in the pipeline.

Returns A list of tuples containing the image path and ground truth path pairs.

greyscale(*probability*)

Convert images to greyscale. For this operation, setting the `probability` to 1.0 is recommended.

See also:

The *black_and_white()* function.

Parameters probability (*Float*) – A value between 0 and 1 representing the probability that the operation should be performed. For resizing, it is recommended that the probability be set to 1.

Returns None

ground_truth(*ground_truth_directory*)

Specifies a directory containing corresponding images that constitute respective ground truth images for the images in the current pipeline.

This function will search the directory specified by `ground_truth_directory` and will associate each ground truth image with the images in the pipeline by file name.

Therefore, an image titled `cat321.jpg` will match with the image `cat321.jpg` in the `ground_truth_directory`. The function respects each image's label, therefore the image named `cat321.jpg` with the label `cat` will match the image `cat321.jpg` in the subdirectory `cat` relative to `ground_truth_directory`.

Typically used to specify a set of ground truth or gold standard images that should be augmented alongside the original images of a dataset, such as image masks or semantic segmentation ground truth images.

A number of such data sets are openly available, see for example https://arxiv.org/pdf/1704.06857.pdf (Garcia-Garcia et al., 2017).

Parameters ground_truth_directory (*String*) – A directory containing the ground truth images that correspond to the images in the current pipeline.

Returns None.

histogram_equalisation(*probability=1.0*)

Apply histogram equalisation to the image.

Parameters probability (*Float*) – A value between 0 and 1 representing the probability that the operation should be performed. For histogram, equalisation it is recommended that the probability be set to 1.

Returns None

image_generator()

Deprecated. Use the sample function and return a generator. :return: A random image passed through the pipeline.

invert(*probability*)

Invert an image. For this operation, setting the `probability` to 1.0 is recommended.

Warning: This function will cause errors if used on binary, 1-bit palette images (e.g. black and white).

Parameters **probability** – A value between 0 and 1 representing the probability that the operation should be performed. For resizing, it is recommended that the probability be set to 1.

Returns None

keras_generator(*batch_size*, *scaled=True*, *image_data_format=u'channels_last'*)

Returns an image generator that will sample from the current pipeline indefinitely, as long as it is called.

Warning: This function returns images from the current pipeline **with replacement**.

You must configure the generator to provide data in the same format that Keras is configured for. You can use the functions `keras.backend.image_data_format()` and `keras.backend.set_image_data_format()` to get and set Keras' image format at runtime.

```
>>> from keras import backend as K
>>> K.image_data_format()
'channels_first'
>>> K.set_image_data_format('channels_last')
>>> K.image_data_format()
'channels_last'
```

By default, Augmentor uses `'channels_last'`.

Parameters

- **batch_size** (*Integer*) – The number of images to return per batch.
- **scaled** (*Boolean*) – True (default) if pixels are to be converted to float32 values between 0 and 1, or False if pixels should be integer values between 0-255.
- **image_data_format** (*String*) – Either `'channels_last'` (default) or `'channels_first'`.

Returns An image generator.

keras_generator_from_array(*images*, *labels*, *batch_size*, *scaled=True*, *image_data_format=u'channels_last'*)

Returns an image generator that will sample from the current pipeline indefinitely, as long as it is called.

Warning: This function returns images from `images` **with replacement**.

You must configure the generator to provide data in the same format that Keras is configured for. You can use the functions `keras.backend.image_data_format()` and `keras.backend.set_image_data_format()` to get and set Keras' image format at runtime.

```
>>> from keras import backend as K
>>> K.image_data_format()
'channels_first'
>>> K.set_image_data_format('channels_last')
>>> K.image_data_format()
'channels_last'
```

By default, Augmentor uses `'channels_last'`.

Parameters

- **images** (Array-like matrix. For greyscale images they can be in the form `(l, x, y)` or `(l, x, y, 1)`, where `l` is the number of images, `x` is the image width and `y` is the image height. For RGB/A images, the matrix should be in the form `(l, x, y, n)`, where `n` is the number of layers, e.g. 3 for RGB or 4 for RGBA and CMYK.) – The images to augment using the current pipeline.
- **labels** (*List.*) – The label associated with each image in `images`.
- **batch_size** (*Integer*) – The number of images to return per batch.
- **scaled** (*Boolean*) – True (default) if pixels are to be converted to float32 values between 0 and 1, or False if pixels should be integer values between 0-255.
- **image_data_format** – Either `'channels_last'` (default) or `'channels_first'`. When `'channels_last'` is specified the returned batch is in the form `(batch_size, x, y, num_channels)`, while for `'channels_last'` the batch is returned in the form `(batch_size, num_channels, x, y)`.
- **image_data_format** – String

Returns An image generator.

keras_preprocess_func()

Returns the pipeline as a function that can be used with Keras ImageDataGenerator. The image array data fed to the returned function is supposed to have scaled to [0, 1]. It will be once converted to PIL format internally as *Image.fromarray(np.uint8(255 * image))*.

```
>>> import Augmentor
>>> import torchvision
>>> p = Augmentor.Pipeline()
>>> p.rotate(probability=0.7, max_left_rotate=10, max_right_rotate=10)
>>> p.zoom(probability=0.5, min_factor=1.1, max_factor=1.5)
>>> from keras.preprocessing.image import ImageDataGenerator
>>> datagen = ImageDataGenerator(
>>>     ...
>>>     preprocessing_function=p.keras_preprocess_func())
```

Returns The pipeline as a function.

process()

This function is used to process every image in the pipeline exactly once.

This might be useful for resizing a dataset for example, and uses multi-threading for fast execution.

It would make sense to set the probability of every operation in the pipeline to `1` when using this function.

Returns None

random_brightness(*probability*, *min_factor*, *max_factor*)

Random change brightness of an image.

Parameters

- **probability** – A value between 0 and 1 representing the probability that the operation should be performed.
- **min_factor** – The value between 0.0 and max_factor that define the minimum adjustment of image brightness. The value 0.0 gives a black image, value 1.0 gives the original image, value bigger than 1.0 gives more bright image.

- **max_factor** – A value should be bigger than min_factor that define the maximum adjustment of image brightness. The value 0.0 gives a black image, value 1.0 gives the original image, value bigger than 1.0 gives more bright image.

Returns None

random_color (*probability*, *min_factor*, *max_factor*)
Random change saturation of an image.

Parameters

- **probability** – Controls the probability that the operation is performed when it is invoked in the pipeline.
- **min_factor** – The value between 0.0 and max_factor that define the minimum adjustment of image saturation. The value 0.0 gives a black and white image, value 1.0 gives the original image.
- **max_factor** – A value should be bigger than min_factor that define the maximum adjustment of image saturation. The value 0.0 gives a black and white image, value 1.0 gives the original image.

Returns None

random_contrast (*probability*, *min_factor*, *max_factor*)
Random change image contrast.

Parameters

- **probability** – Controls the probability that the operation is performed when it is invoked in the pipeline.
- **min_factor** – The value between 0.0 and max_factor that define the minimum adjustment of image contrast. The value 0.0 gives s solid grey image, value 1.0 gives the original image.
- **max_factor** – A value should be bigger than min_factor that define the maximum adjustment of image contrast. The value 0.0 gives s solid grey image, value 1.0 gives the original image.

Returns None

random_distortion (*probability*, *grid_width*, *grid_height*, *magnitude*)
Performs a random, elastic distortion on an image.

This function performs a randomised, elastic distortion controlled by the parameters specified. The grid width and height controls how fine the distortions are. Smaller sizes will result in larger, more pronounced, and less granular distortions. Larger numbers will result in finer, more granular distortions. The magnitude of the distortions can be controlled using magnitude. This can be random or fixed.

Good values for parameters are between 2 and 10 for the grid width and height, with a magnitude of between 1 and 10. Using values outside of these approximate ranges may result in unpredictable behaviour.

Parameters

- **probability** (*Float*) – A value between 0 and 1 representing the probability that the operation should be performed.
- **grid_width** (*Integer*) – The number of rectangles in the grid's horizontal axis.
- **grid_height** (*Integer*) – The number of rectangles in the grid's vertical axis.
- **magnitude** (*Integer*) – The magnitude of the distortions.

Returns None

`random_erasing` (*probability*, *rectangle_area*)
Work in progress. This operation performs a Random Erasing operation, as described in https://arxiv.org/abs/1708.04896 by Zhong et al.

Its purpose is to make models robust to occlusion, by randomly replacing rectangular regions with random pixel values.

For greyscale images the random pixels values will also be greyscale, and for RGB images the random pixels values will be in RGB.

This operation is subject to change, the original work describes several ways of filling the random regions, including a random solid colour or greyscale value. Currently this operations uses the method which yielded the best results in the tests performed by Zhong et al.

Parameters

- **`probability`** – A value between 0 and 1 representing the probability that the operation should be performed.
- **`rectangle_area`** – The percentage area of the image to occlude with the random rectangle, between 0.1 and 1.

Returns None

`remove_operation` (*operation_index=-1*)
Remove the operation specified by `operation_index`, if supplied, otherwise it will remove the latest operation added to the pipeline.

See also:

Use the *`status()`* function to find an operation's index.

Parameters **`operation_index`** (*`Integer`*) – The index of the operation to remove.

Returns The removed operation. You can reinsert this at end of the pipeline using *`add_operation()`* if required.

`resize` (*probability*, *width*, *height*, *resample_filter=u'BICUBIC'*)
Resize an image according to a set of dimensions specified by the user in pixels.

Parameters

- **`probability`** (*`Float`*) – A value between 0 and 1 representing the probability that the operation should be performed. For resizing, it is recommended that the probability be set to 1.
- **`width`** (*`Integer`*) – The new width that the image should be resized to.
- **`height`** (*`Integer`*) – The new height that the image should be resized to.
- **`resample_filter`** (*`String`*) – The resampling filter to use. Must be one of BICUBIC, BILINEAR, ANTIALIAS, or NEAREST.

Returns None

`rotate` (*probability*, *max_left_rotation*, *max_right_rotation*)
Rotate an image by an arbitrary amount.

The operation will rotate an image by an random amount, within a range specified. The parameters `max_left_rotation` and `max_right_rotation` allow you to control this range. If you wish to rotate the images by an exact number of degrees, set both `max_left_rotation` and `max_right_rotation` to the same value.

Note: This function will rotate **in place**, and crop the largest possible rectangle from the rotated image.

In practice, angles larger than 25 degrees result in images that do not render correctly, therefore there is a limit of 25 degrees for this function.

If this function returns images that are not rendered correctly, then you must reduce the `max_left_rotation` and `max_right_rotation` arguments!

Parameters

- **`probability`** (*`Float`*) – A value between 0 and 1 representing the probability that the operation should be performed.
- **`max_left_rotation`** (*`Integer`*) – The maximum number of degrees the image can be rotated to the left.
- **`max_right_rotation`** (*`Integer`*) – The maximum number of degrees the image can be rotated to the right.

Returns None

`rotate180`(*probability*)

Rotate an image by 180 degrees.

The operation will rotate an image by 180 degrees, and will be performed with a probability of that specified by the `probability` parameter.

Parameters **`probability`** (*`Float`*) – A value between 0 and 1 representing the probability that the operation should be performed.

Returns None

`rotate270`(*probability*)

Rotate an image by 270 degrees.

The operation will rotate an image by 270 degrees, and will be performed with a probability of that specified by the `probability` parameter.

Parameters **`probability`** (*`Float`*) – A value between 0 and 1 representing the probability that the operation should be performed.

Returns None

`rotate90`(*probability*)

Rotate an image by 90 degrees.

The operation will rotate an image by 90 degrees, and will be performed with a probability of that specified by the `probability` parameter.

Parameters **`probability`** (*`Float`*) – A value between 0 and 1 representing the probability that the operation should be performed.

Returns None

`rotate_random_90`(*probability*)

Rotate an image by either 90, 180, or 270 degrees, selected randomly.

This function will rotate by either 90, 180, or 270 degrees. This is useful to avoid scenarios where images may be rotated back to their original positions (such as a *`rotate90()`* and a *`rotate270()`* being performed directly afterwards. The random rotation is chosen uniformly from 90, 180, or 270 degrees. The probability controls the chance of the operation being performed at all, and does not affect the rotation degree.

Parameters **`probability`** (*`Float`*) – A value between 0 and 1 representing the probability that the operation should be performed.

Returns None

`rotate_without_crop`(*probability*, *max_left_rotation*, *max_right_rotation*, *expand=False*, *fillcolor=None*)

Rotate an image without automatically cropping.

The `expand` parameter controls whether the image is enlarged to contain the new rotated images, or if the image size is maintained Defaults to `false` so that images maintain their dimensions when using this function.

Parameters

- **`probability`** (*`Float`*) – A value between 0 and 1 representing the probability that the operation should be performed.
- **`max_left_rotation`** (*`Integer`*) – The maximum number of degrees the image can be rotated to the left.
- **`max_right_rotation`** (*`Integer`*) – The maximum number of degrees the image can be rotated to the right.
- **`expand`** – Controls whether the image's size should be increased to accommodate the rotation. Defaults to `false` so that images maintain their original dimensions after rotation.
- **`fillcolor`** – Specify color to fill points outside the boundaries of the input. Default value is None (points filled with black color). For example, in case of RGB color scheme simply use *(r, g, b)* tuple of int numbers.

Returns None

`sample`(*n*, *multi_threaded=True*)

Generate `n` number of samples from the current pipeline.

This function samples from the pipeline, using the original images defined during instantiation. All images generated by the pipeline are by default stored in an `output` directory, relative to the path defined during the pipeline's instantiation.

By default, Augmentor will use multi-threading to increase the speed of processing the images. However, this may slow down some operations if the images are very small. Set `multi_threaded` to `False` if slowdown is experienced.

Parameters

- **`n`** (*`Integer`*) – The number of new samples to produce.
- **`multi_threaded`** (*`Boolean`*) – Whether to use multi-threading to process the images. Defaults to `True`.

Returns None

`sample_with_array`(*image_array*, *save_to_disk=False*)

Generate images using a single image in array-like format.

See also:

See `keras_image_generator_without_replacement()`

Parameters

- **`image_array`** – The image to pass through the pipeline.
- **`save_to_disk`** – Whether to save to disk or not (default).

Returns

`scale` (*probability*, *scale_factor*)

Scale (enlarge) an image, while maintaining its aspect ratio. This returns an image with larger dimensions than the original image.

Use `resize()` to resize an image to absolute pixel values.

Parameters

- **`probability`** (*`Float`*) – A value between 0 and 1 representing the probability that the operation should be performed.
- **`scale_factor`** (*`Float`*) – The factor to scale by, which must be greater than 1.0.

Returns None

`set_save_format` (*save_format*)

Set the save format for the pipeline. Pass the value `save_format="auto"` to allow Augmentor to choose the correct save format based on each individual image's file extension.

If `save_format` is set to, for example, `save_format="JPEG"` or `save_format="JPG"`, Augmentor will attempt to save the files using the JPEG format, which may result in errors if the file cannot be saved in this format, such as trying to save PNG images with an alpha channel as JPEG.

Parameters **`save_format`** – The save format to save the images when writing to disk.

Returns None

`static set_seed` (*seed*)

Set the seed of Python's internal random number generator.

Parameters **`seed`** (*`Integer`*) – The seed to use. Strings or other objects will be hashed.

Returns None

`shear` (*probability*, *max_shear_left*, *max_shear_right*)

Shear the image by a specified number of degrees.

In practice, shear angles of more than 25 degrees can cause unpredictable behaviour. If you are observing images that are incorrectly rendered (e.g. they do not contain any information) then reduce the shear angles.

Parameters

- **`probability`** – The probability that the operation is performed.
- **`max_shear_left`** – The max number of degrees to shear to the left. Cannot be larger than 25 degrees.
- **`max_shear_right`** – The max number of degrees to shear to the right. Cannot be larger than 25 degrees.

Returns None

`skew` (*probability*, *magnitude=1*)

Skew an image in a random direction, either left to right, top to bottom, or one of 8 corner directions.

To see examples of all the skew types, see *Perspective Skewing*.

Parameters

- **`probability`** (*`Float`*) – A value between 0 and 1 representing the probability that the operation should be performed.
- **`magnitude`** (*`Float`*) – The maximum skew, which must be value between 0.1 and 1.0.

Returns None

`skew_corner` (*probability*, *magnitude=1*)
Skew an image towards one corner, randomly by a random magnitude.

To see examples of the various skews, see *Perspective Skewing*.

Parameters

- **`probability`** – A value between 0 and 1 representing the probability that the operation should be performed.
- **`magnitude`** – The maximum skew, which must be value between 0.1 and 1.0.

Returns

`skew_left_right` (*probability*, *magnitude=1*)
Skew an image by tilting it left or right by a random amount. The magnitude of this skew can be set to a maximum using the magnitude parameter. This can be either a scalar representing the maximum tilt, or vector representing a range.

To see examples of the various skews, see *Perspective Skewing*.

Parameters

- **`probability`** (*`Float`*) – A value between 0 and 1 representing the probability that the operation should be performed.
- **`magnitude`** (*`Float`*) – The maximum tilt, which must be value between 0.1 and 1.0, where 1 represents a tilt of 45 degrees.

Returns None

`skew_tilt` (*probability*, *magnitude=1*)
Skew an image by tilting in a random direction, either forwards, backwards, left, or right, by a random amount. The magnitude of this skew can be set to a maximum using the magnitude parameter. This can be either a scalar representing the maximum tilt, or vector representing a range.

To see examples of the various skews, see *Perspective Skewing*.

Parameters

- **`probability`** (*`Float`*) – A value between 0 and 1 representing the probability that the operation should be performed.
- **`magnitude`** (*`Float`*) – The maximum tilt, which must be value between 0.1 and 1.0, where 1 represents a tilt of 45 degrees.

Returns None

`skew_top_bottom` (*probability*, *magnitude=1*)
Skew an image by tilting it forwards or backwards by a random amount. The magnitude of this skew can be set to a maximum using the magnitude parameter. This can be either a scalar representing the maximum tilt, or vector representing a range.

To see examples of the various skews, see *Perspective Skewing*.

Parameters

- **`probability`** (*`Float`*) – A value between 0 and 1 representing the probability that the operation should be performed.
- **`magnitude`** (*`Float`*) – The maximum tilt, which must be value between 0.1 and 1.0, where 1 represents a tilt of 45 degrees.

Returns None

status()

Prints the status of the pipeline to the console. If you want to remove an operation, use the index shown and the *remove_operation()* method.

See also:

The *remove_operation()* function.

See also:

The *add_operation()* function.

The status includes the number of operations currently attached to the pipeline, each operation's parameters, the number of images in the pipeline, and a summary of the images' properties, such as their dimensions and formats.

Returns None

torch_transform()

Returns the pipeline as a function that can be used with torchvision.

```
>>> import Augmentor
>>> import torchvision
>>> p = Augmentor.Pipeline()
>>> p.rotate(probability=0.7, max_left_rotate=10, max_right_rotate=10)
>>> p.zoom(probability=0.5, min_factor=1.1, max_factor=1.5)
>>> transforms = torchvision.transforms.Compose([
>>>     p.torch_transform(),
>>>     torchvision.transforms.ToTensor(),
>>> ])
```

Returns The pipeline as a function.

zoom(*probability*, *min_factor*, *max_factor*)

Zoom in to an image, while **maintaining its size**. The amount by which the image is zoomed is a randomly chosen value between `min_factor` and `max_factor`.

Typical values may be `min_factor=1.1` and `max_factor=1.5`.

To zoom by a constant amount, set `min_factor` and `max_factor` to the same value.

See also:

See *zoom_random()* for zooming into random areas of the image.

Parameters

- **probability** (*Float*) – A value between 0 and 1 representing the probability that the operation should be performed.
- **min_factor** (*Float*) – The minimum factor by which to zoom the image.
- **max_factor** (*Float*) – The maximum factor by which to zoom the image.

Returns None

zoom_random(*probability*, *percentage_area*, *randomise_percentage_area=False*)

Zooms into an image at a random location within the image.

You can randomise the zoom level by setting the `randomise_percentage_area` argument to true.

See also:

See `zoom()` for zooming into the centre of images.

Parameters

- **`probability`** – The probability that the function will execute when the image is passed through the pipeline.
- **`percentage_area`** – The area, as a percentage of the current image's area, to crop.
- **`randomise_percentage_area`** – If True, will use `percentage_area` as an upper bound and randomise the crop from between 0 and `percentage_area`.

Returns None

6.1.2 Documentation of the Operations module

The Operations module contains classes for all operations used by Augmentor.

The classes contained in this module are not called or instantiated directly by the user, instead the user interacts with the `Pipeline` class and uses the utility functions contained there.

In this module, each operation is a subclass of type `Operation`. The `Pipeline` objects expect `Operation` types, and therefore all operations are of type `Operation`, and provide their own implementation of the `perform_operation()` function.

Hence, the documentation for this module is intended for developers who wish to extend Augmentor or wish to see how operations function internally.

For detailed information on extending Augmentor, see *Extending Augmentor*.

class Augmentor.Operations.**BlackAndWhite**(*probability*, *threshold*)

This class is used to convert images into black and white. In other words, into using a 1-bit, monochrome binary colour palette. This is not to be confused with greyscale, where an 8-bit greyscale pixel intensity range is used.

See also:

The `Greyscale` class.

As well as the required `probability` parameter, a `threshold` can also be defined to define the cut-off point where a pixel is converted to black or white. The `threshold` defaults to 128 at the user-facing `black_and_white()` function.

Parameters

- **`probability`** (*`Float`*) – Controls the probability that the operation is performed when it is invoked in the pipeline.
- **`threshold`** (*`Integer`*) – A value between 0 and 255 that defines the cut off point where an individual pixel is converted into black or white.

perform_operation(*images*)

Convert the image passed as an argument to black and white, 1-bit monochrome. Uses the `threshold` passed to the constructor to control the cut-off point where a pixel is converted to black or white.

Parameters **`images`** (*`List containing PIL.Image object(s)`*) – The image to convert into monochrome.

Returns The transformed image(s) as a list of object(s) of type PIL.Image.

class Augmentor.Operations.**Crop**(*probability*, *width*, *height*, *centre*)

This class is used to crop images by absolute values passed as parameters.

As well as the always required `probability` parameter, the constructor requires a `width` to control the width of of the area to crop as well as a `height` parameter to control the height of the area to crop. Also, whether the area to crop should be taken from the centre of the image or from a random location within the image is toggled using `centre`.

Parameters

- **`probability`** (*`Float`*) – Controls the probability that the operation is performed when it is invoked in the pipeline.
- **`width`** (*`Integer`*) – The width in pixels of the area to crop from the image.
- **`height`** (*`Integer`*) – The height in pixels of the area to crop from the image.
- **`centre`** (*`Boolean`*) – Whether to crop from the centre of the image or a random location within the image, while maintaining the size of the crop without cropping out of the original image's area.

`perform_operation`(*images*)

Crop an area from an image, either from a random location or centred, using the dimensions supplied during instantiation.

Parameters **`images`** (*`List containing PIL.Image object(s)`*) – The image(s) to crop the area from.

Returns The transformed image(s) as a list of object(s) of type PIL.Image.

class `Augmentor.Operations.`**`CropPercentage`**(*probability*, *percentage_area*, *centre*, *randomise_percentage_area*)

This class is used to crop images by a percentage of their area.

As well as the always required `probability` parameter, the constructor requires a `percentage_area` to control the area of the image to crop in terms of its percentage of the original image, and a `centre` parameter toggle whether a random area or the centre of the images should be cropped.

Parameters

- **`probability`** (*`Float`*) – Controls the probability that the operation is performed when it is invoked in the pipeline.
- **`percentage_area`** (*`Float`*) – The percentage area of the original image to crop. A value of 0.5 would crop an area that is 50% of the area of the original image's size.
- **`centre`** (*`Boolean`*) – Whether to crop from the centre of the image or crop a random location within the image.

`perform_operation`(*images*)

Crop the passed `images` by percentage area, returning the crop as an image.

Parameters **`images`** (*`List containing PIL.Image object(s)`*) – The image(s) to crop an area from.

Returns The transformed image(s) as a list of object(s) of type PIL.Image.

class `Augmentor.Operations.`**`CropRandom`**(*probability*, *percentage_area*)

Warning: This *`CropRandom`* class is currently not used by any of the user-facing functions in the *`Pipeline`* class.

Parameters

- **`probability`** – Controls the probability that the operation is performed when it is invoked in the pipeline.
- **`percentage_area`** – The percentage area of the original image to crop. A value of 0.5 would crop an area that is 50% of the area of the original image's size.

`perform_operation`(*images*)
Randomly crop the passed image, returning the crop as a new image.

Parameters **`images`** (*`List containing PIL.Image object(s)`*) – The image to crop.

Returns The transformed image(s) as a list of object(s) of type PIL.Image.

class `Augmentor.Operations.`**`Custom`**(*probability*, *custom_function*, ***function_arguments*)
Class that allows for a custom operation to be performed using Augmentor's standard *`Pipeline`* object.

Creates a custom operation that can be added to a pipeline.

To add a custom operation you can instantiate this class, passing a function pointer, `custom_function`, followed by an arbitrarily long list keyword arguments, `**function_arguments`.

See also:

The *`add_operation()`* function.

Parameters

- **`custom_function`** is listed below.

- **`probability`** (*`Float`*) – The probability that the operation will be performed.
- **`custom_function`** (*`*Function`*) – The name of the function that performs your custom code. Must return an Image object and accept an Image object as its first parameter.
- **`function_arguments`** (*`dict`*) – The arguments for your custom operation's code.

`perform_operation`(*images*)
Perform the custom operation on the passed image(s), returning the transformed image(s).

Parameters **`images`** – The image to perform the custom operation on.

Returns The transformed image(s) (other functions in the pipeline will expect an image of type PIL.Image)

class `Augmentor.Operations.`**`Distort`**(*probability*, *grid_width*, *grid_height*, *magnitude*)
This class performs randomised, elastic distortions on images.

As well as the probability, the granularity of the distortions produced by this class can be controlled using the width and height of the overlaying distortion grid. The larger the height and width of the grid, the smaller the distortions. This means that larger grid sizes can result in finer, less severe distortions. As well as this, the magnitude of the distortions vectors can also be adjusted.

Parameters

- **`probability`** (*`Float`*) – Controls the probability that the operation is performed when it is invoked in the pipeline.
- **`grid_width`** (*`Integer`*) – The width of the gird overlay, which is used by the class to apply the transformations to the image.
- **`grid_height`** (*`Integer`*) – The height of the gird overlay, which is used by the class to apply the transformations to the image.
- **`magnitude`** (*`Integer`*) – Controls the degree to which each distortion is applied to the overlaying distortion grid.

`perform_operation`(*images*)

Distorts the passed image(s) according to the parameters supplied during instantiation, returning the newly distorted image.

Parameters **`images`** (*`List containing PIL.Image object(s)`*) – The image(s) to be distorted.

Returns The transformed image(s) as a list of object(s) of type PIL.Image.

class `Augmentor.Operations.`**`Flip`**(*probability*, *top_bottom_left_right*)

This class is used to mirror images through the x or y axes.

The class allows an image to be mirrored along either its x axis or its y axis, or randomly.

The direction of the flip, or whether it should be randomised, is controlled using the `top_bottom_left_right` parameter.

Parameters

- **`probability`** – Controls the probability that the operation is performed when it is invoked in the pipeline.
- **`top_bottom_left_right`** – Controls the direction the image should be mirrored. Must be one of `LEFT_RIGHT`, `TOP_BOTTOM`, or `RANDOM`.
  - `LEFT_RIGHT` defines that the image is mirrored along its x axis.
  - `TOP_BOTTOM` defines that the image is mirrored along its y axis.
  - `RANDOM` defines that the image is mirrored randomly along either the x or y axis.

`perform_operation`(*images*)

Mirror the image according to the *attr*:top_bottom_left_right‘ argument passed to the constructor and return the mirrored image.

Parameters **`images`** (*`List containing PIL.Image object(s)`*) – The image(s) to mirror.

Returns The transformed image(s) as a list of object(s) of type PIL.Image.

class `Augmentor.Operations.`**`GaussianDistortion`**(*probability*, *grid_width*, *grid_height*, *magnitude*, *corner*, *method*, *mex*, *mey*, *sdx*, *sdy*)

This class performs randomised, elastic gaussian distortions on images.

As well as the probability, the granularity of the distortions produced by this class can be controlled using the width and height of the overlaying distortion grid. The larger the height and width of the grid, the smaller the distortions. This means that larger grid sizes can result in finer, less severe distortions. As well as this, the magnitude of the distortions vectors can also be adjusted.

Parameters

- **`probability`** (*`Float`*) – Controls the probability that the operation is performed when it is invoked in the pipeline.
- **`grid_width`** (*`Integer`*) – The width of the gird overlay, which is used by the class to apply the transformations to the image.
- **`grid_height`** (*`Integer`*) – The height of the gird overlay, which is used by the class to apply the transformations to the image.
- **`magnitude`** (*`Integer`*) – Controls the degree to which each distortion is applied to the overlaying distortion grid.

- **corner** (*String*) – which corner of picture to distort. Possible values: "bell"(circular surface applied), "ul"(upper left), "ur"(upper right), "dl"(down left), "dr"(down right).
- **method** (*String*) – possible values: "in"(apply max magnitude to the chosen corner), "out"(inverse of method in).
- **mex** (*Float*) – used to generate 3d surface for similar distortions. Surface is based on normal distribution.
- **mey** (*Float*) – used to generate 3d surface for similar distortions. Surface is based on normal distribution.
- **sdx** (*Float*) – used to generate 3d surface for similar distortions. Surface is based on normal distribution.
- **sdy** (*Float*) – used to generate 3d surface for similar distortions. Surface is based on normal distribution.

For values `mex`, `mey`, `sdx`, and `sdy` the surface is based on the normal distribution:

$$e^{-\left(\frac{(x-\text{mex})^2}{\text{sdx}}+\frac{(y-\text{mey})^2}{\text{sdy}}\right)}$$

perform_operation(*images*)

Distorts the passed image(s) according to the parameters supplied during instantiation, returning the newly distorted image.

Parameters images (*List containing PIL.Image object(s)*) – The image(s) to be distorted.

Returns The transformed image(s) as a list of object(s) of type PIL.Image.

class Augmentor.Operations.**Greyscale**(*probability*)

This class is used to convert images into greyscale. That is, it converts images into having only shades of grey (pixel value intensities) varying from 0 to 255 which represent black and white respectively.

See also:

The *BlackAndWhite* class.

As there are no further user definable parameters, the class is instantiated using only the `probability` argument.

Parameters probability (*Float*) – Controls the probability that the operation is performed when it is invoked in the pipeline.

perform_operation(*images*)

Converts the passed image to greyscale and returns the transformed image. There are no user definable parameters for this method.

Parameters images (*List containing PIL.Image object(s)*) – The image to convert to greyscale.

Returns The transformed image(s) as a list of object(s) of type PIL.Image.

class Augmentor.Operations.**HSVShifting**(*probability*, *hue_shift*, *saturation_scale*, *saturation_shift*, *value_scale*, *value_shift*)

CURRENTLY NOT IMPLEMENTED.

perform_operation(*images*)

class `Augmentor.Operations.`**`HistogramEqualisation`**(*probability*)

The class `HistogramEqualisation` is used to perform histogram equalisation on images passed to its `perform_operation()` function.

As there are no further user definable parameters, the class is instantiated using only the `probability` argument.

Parameters **`probability`** (*`Float`*) – Controls the probability that the operation is performed when it is invoked in the pipeline.

`perform_operation`(*images*)

Performs histogram equalisation on the images passed as an argument and returns the equalised images. There are no user definable parameters for this method.

Parameters **`images`** (*`List containing PIL.Image object(s)`*) – The image(s) on which to perform the histogram equalisation.

Returns The transformed image(s) as a list of object(s) of type PIL.Image.

class `Augmentor.Operations.`**`Invert`**(*probability*)

This class is used to negate images. That is to reverse the pixel values for any image processed by it.

As there are no further user definable parameters, the class is instantiated using only the `probability` argument.

Parameters **`probability`** (*`Float`*) – Controls the probability that the operation is performed when it is invoked in the pipeline.

`perform_operation`(*images*)

Negates the image passed as an argument. There are no user definable parameters for this method.

Parameters **`images`** (*`List containing PIL.Image object(s)`*) – The image(s) to negate.

Returns The transformed image(s) as a list of object(s) of type PIL.Image.

class `Augmentor.Operations.`**`Mixup`**(*probability*, *alpha=0.4*)

Implements the *mixup* augmentation method, as described in:

Zhang et al. (2018), *mixup*: Beyond Empirical Risk Minimization, arXiv:1710.09412

See http://arxiv.org/abs/1710.09412 for details.

Also see https://github.com/facebookresearch/mixup-cifar10 for code which was followed to create this functionality in Augmentor.

The *mixup* augmentation technique trains a neural network on "*combinations of pairs of examples and their labels*" (Zhang et al., 2018).

In summary, *mixup* constructs training samples as follows:

$$\tilde{x} = \lambda x_i + (1 - \lambda)x_j$$

$$\tilde{y} = \lambda y_i + (1 - \lambda)y_j$$

where (x_i, y_i) and (x_j, y_j) are two samples from the training data, x_i, x_j are raw input vectors, y_i, y_j are one-hot label encodings (such as [0.0, 1.0]), and $\lambda \in [0, 1]$ where λ is sampled randomly from the Beta distribution, $\beta(\alpha, \alpha)$.

The α hyper-parameter controls the strength of the interpolation between image-label pairs, where $\alpha \in \{0, \infty\}$

According to the paper referenced above, values for α between 0.1 and 0.4 led to best performance. Smaller values for α result in less *mixup* effect where larger values would tend to result in overfitting.

Performs the *mixit* augmentation technique.

Note: Not yet enabled! This function is currently implemented but not **enabled**, as it requires each image's label in order to operate - something which Augmentor was not designed to handle.

Parameters

- **`probability`** (*`Float`*) – Controls the probability that the operation is performed when it is invoked in the pipeline.
- **`alpha`** (*`Float`*) – The alpha parameter controls the strength of the interpolation between image-label pairs. It's value can be any value greater than 0. A smaller value for `alpha` results in more values closer to 0 or 1, meaning the *mixup* is more often closer to either of the images in the pair. Its value is set to 0.4 by default.

`perform_operation`(*images*)

This function is currently implemented but not **enabled**, as it requires each image's label in order to operate - something which Augmentor was not designed to handle.

This is therefore future work, and may only be possible when used in combination with generators.

`class` `Augmentor.Operations.`**`Operation`**(*probability*)

The class *`Operation`* represents the base class for all operations that can be performed. Inherit from *`Operation`*, overload its methods, and instantiate super to create a new operation. See the section on extending Augmentor with custom operations at *Extending Augmentor*.

All operations must at least have a `probability` which is initialised when creating the operation's object.

Parameters **`probability`** (*`Float`*) – Controls the probability that the operation is performed when it is invoked in the pipeline.

`perform_operation`(*images*)

Perform the operation on the passed images. Each operation must at least have this function, which accepts a list containing objects of type PIL.Image, performs its operation, and returns a new list containing objects of type PIL.Image.

Parameters **`images`** (*`List containing PIL.Image object(s)`*) – The image(s) to transform.

Returns The transformed image(s) as a list of object(s) of type PIL.Image.

`class` `Augmentor.Operations.`**`RandomBrightness`**(*probability*, *min_factor*, *max_factor*)

This class is used to random change image brightness.

required `probability` parameter

`random_brightness()` function.

Parameters

- **`probability`** (*`Float`*) – Controls the probability that the operation is performed when it is invoked in the pipeline.
- **`min_factor`** – The value between 0.0 and max_factor that define the minimum adjustment of image brightness. The value 0.0 gives a black image,The value 1.0 gives the original image, value bigger than 1.0 gives more bright image.
- **`max_factor`** (*`Float`*) – A value should be bigger than min_factor. that define the maximum adjustment of image brightness. The value 0.0 gives a black image, value 1.0 gives the original image, value bigger than 1.0 gives more bright image.

perform_operation(*images*)
Random change the passed image brightness.

Parameters images (*List containing PIL.Image object(s)*) – The image to convert into monochrome.

Returns The transformed image(s) as a list of object(s) of type PIL.Image.

class Augmentor.Operations.**RandomColor**(*probability*, *min_factor*, *max_factor*)
This class is used to random change saturation of an image.

required `probability` parameter

random_color() function.

Parameters

- **probability** (*Float*) – Controls the probability that the operation is performed when it is invoked in the pipeline.
- **min_factor** – The value between 0.0 and max_factor that define the minimum adjustment of image saturation. The value 0.0 gives a black and white image, value 1.0 gives the original image.
- **max_factor** (*Float*) – A value should be bigger than min_factor. that define the maximum adjustment of image saturation. The value 0.0 gives a black and white image, value 1.0 gives the original image.

perform_operation(*images*)
Random change the passed image saturation.

Parameters images (*List containing PIL.Image object(s)*) – The image to convert into monochrome.

Returns The transformed image(s) as a list of object(s) of type PIL.Image.

class Augmentor.Operations.**RandomContrast**(*probability*, *min_factor*, *max_factor*)
This class is used to random change contrast of an image.

required `probability` parameter

random_contrast() function.

Parameters

- **probability** (*Float*) – Controls the probability that the operation is performed when it is invoked in the pipeline.
- **min_factor** – The value between 0.0 and max_factor that define the minimum adjustment of image contrast. The value 0.0 gives s solid grey image, value 1.0 gives the original image.
- **max_factor** (*Float*) – A value should be bigger than min_factor. that define the maximum adjustment of image contrast. The value 0.0 gives s solid grey image, value 1.0 gives the original image.

perform_operation(*images*)
Random change the passed image contrast.

Parameters images (*List containing PIL.Image object(s)*) – The image to convert into monochrome.

Returns The transformed image(s) as a list of object(s) of type PIL.Image.

class `Augmentor.Operations.`**`RandomErasing`**(*probability*, *rectangle_area*)

Class that performs Random Erasing, an augmentation technique described in https://arxiv.org/abs/1708.04896 by Zhong et al. To quote the authors, random erasing:

"*. . . randomly selects a rectangle region in an image, and erases its pixels with random values.*"

Exactly this is provided by this class.

Random Erasing can make a trained neural network more robust to occlusion.

The size of the random rectangle is controlled using the `rectangle_area` parameter. This area is random in its width and height.

Parameters

- **`probability`** – The probability that the operation will be performed.
- **`rectangle_area`** – The percentage are of the image to occlude.

`perform_operation`(*images*)

Adds a random noise rectangle to a random area of the passed image, returning the original image with this rectangle superimposed.

Parameters **`images`** (*`List containing PIL.Image object(s)`*) – The image(s) to add a random noise rectangle to.

Returns The transformed image(s) as a list of object(s) of type PIL.Image.

class `Augmentor.Operations.`**`Resize`**(*probability*, *width*, *height*, *resample_filter*)

This class is used to resize images by absolute values passed as parameters.

Accepts the required probability parameter as well as parameters to control the size of the transformed image.

Parameters

- **`probability`** (*`Float`*) – Controls the probability that the operation is performed when it is invoked in the pipeline.
- **`width`** (*`Integer`*) – The width in pixels to resize the image to.
- **`height`** (*`Integer`*) – The height in pixels to resize the image to.
- **`resample_filter`** (*`String`*) – The resample filter to use. Must be one of the standard PIL types, i.e. `NEAREST`, `BICUBIC`, `ANTIALIAS`, or `BILINEAR`.

`perform_operation`(*images*)

Resize the passed image and returns the resized image. Uses the parameters passed to the constructor to resize the passed image.

Parameters **`images`** (*`List containing PIL.Image object(s)`*) – The image to resize.

Returns The transformed image(s) as a list of object(s) of type PIL.Image.

class `Augmentor.Operations.`**`Rotate`**(*probability*, *rotation*)

This class is used to perform rotations on images in multiples of 90 degrees. Arbitrary rotations are handled by the *`RotateRange`* class.

As well as the required `probability` parameter, the `rotation` parameter controls the rotation to perform, which must be one of `90`, `180`, `270` or `-1` (see below).

Parameters

- **`probability`** – Controls the probability that the operation is performed when it is invoked in the pipeline.

- **rotation** – Controls the rotation to perform. Must be one of `90`, `180`, `270` or `-1`.
  - `90` rotate the image by 90 degrees.
  - `180` rotate the image by 180 degrees.
  - `270` rotate the image by 270 degrees.
  - `-1` rotate the image randomly by either 90, 180, or 270 degrees.

See also:

For arbitrary rotations, see the *RotateRange* class.

perform_operation(*images*)

Rotate an image by either 90, 180, or 270 degrees, or randomly from any of these.

Parameters images (*List containing PIL.Image object(s)*) – The image(s) to rotate.

Returns The transformed image(s) as a list of object(s) of type PIL.Image.

class Augmentor.Operations.**RotateRange**(*probability*, *max_left_rotation*, *max_right_rotation*)

This class is used to perform rotations on images by arbitrary numbers of degrees.

Images are rotated **in place** and an image of the same size is returned by this function. That is to say, that after a rotation has been performed, the largest possible area of the same aspect ratio of the original image is cropped from the skewed image, and this is then resized to match the original image size.

The method by which this is performed is described as follows:

$$E = \frac{\frac{\sin\theta_a}{\sin\theta_b}\left(X - \frac{\sin\theta_a}{\sin\theta_b}Y\right)}{1 - \frac{(\sin\theta_a)^2}{(\sin\theta_b)^2}}$$

which describes how E is derived, and then follows $B = Y - E$ and $A = \frac{\sin\theta_a}{\sin\theta_b}B$.

The *Rotating* section describes this in detail and has example images to demonstrate this.

As well as the required `probability` parameter, the `max_left_rotation` parameter controls the maximum number of degrees by which to rotate to the left, while the `max_right_rotation` controls the maximum number of degrees to rotate to the right.

Parameters

- **probability** (*Float*) – Controls the probability that the operation is performed when it is invoked in the pipeline.
- **max_left_rotation** (*Integer*) – The maximum number of degrees to rotate the image anti-clockwise.
- **max_right_rotation** (*Integer*) – The maximum number of degrees to rotate the image clockwise.

perform_operation(*images*)

Perform the rotation on the passed `image` and return the transformed image. Uses the `max_left_rotation` and `max_right_rotation` passed into the constructor to control the amount of degrees to rotate by. Whether the image is rotated clockwise or anti-clockwise is chosen at random.

Parameters images (*List containing PIL.Image object(s)*) – The image(s) to rotate.

Returns The transformed image(s) as a list of object(s) of type PIL.Image.

class `Augmentor.Operations.`**`RotateStandard`**(*probability*, *max_left_rotation*, *max_right_rotation*, *expand=False*, *fillcolor=None*)

Class to perform rotations without automatically cropping the images, as opposed to the `RotateRange` class.

See also:

For arbitrary rotations with automatic cropping, see the `RotateRange` class.

See also:

For 90 degree rotations, see the `Rotate` class.

Documentation to appear.

`perform_operation`(*images*)

Documentation to appear.

Returns The transformed image(s) as a list of object(s) of type PIL.Image.

class `Augmentor.Operations.`**`Scale`**(*probability*, *scale_factor*)

This class is used to increase or decrease images in size by a certain factor, while maintaining the aspect ratio of the original image.

See also:

The `Resize` class for resizing images by **dimensions**, and hence will not necessarily maintain the aspect ratio.

This function will return images that are **larger** than the input images.

As the aspect ratio is always kept constant, only a `scale_factor` is required for scaling the image.

Parameters

- **`probability`** (*`Float`*) – Controls the probability that the operation is performed when it is invoked in the pipeline.
- **`scale_factor`** (*`Float`*) – The factor by which to scale, where 1.5 would result in an image scaled up by 150%.

`perform_operation`(*images*)

Scale the passed `images` by the factor specified during instantiation, returning the scaled image.

Parameters **`images`** (*`List containing PIL.Image object(s)`*) – The image to scale.

Returns The transformed image(s) as a list of object(s) of type PIL.Image.

class `Augmentor.Operations.`**`Shear`**(*probability*, *max_shear_left*, *max_shear_right*)

This class is used to shear images, that is to tilt them in a certain direction. Tilting can occur along either the x- or y-axis and in both directions (i.e. left or right along the x-axis, up or down along the y-axis).

Images are sheared **in place** and an image of the same size as the input image is returned by this class. That is to say, that after a shear has been performed, the largest possible area of the same aspect ratio of the original image is cropped from the sheared image, and this is then resized to match the original image size. The *Shearing* section describes this in detail.

For sample code with image examples see *Shearing*.

The shearing is randomised in magnitude, from 0 to the `max_shear_left` or 0 to `max_shear_right` where the direction is randomised. The shear axis is also randomised i.e. if it shears up/down along the y-axis or left/right along the x-axis.

Parameters

- **probability** (*Float*) – Controls the probability that the operation is performed when it is invoked in the pipeline.
- **max_shear_left** (*Integer*) – The maximum shear to the left.
- **max_shear_right** (*Integer*) – The maximum shear to the right.

perform_operation(*images*)

Shears the passed image according to the parameters defined during instantiation, and returns the sheared image.

Parameters **images** (*List containing PIL.Image object(s)*) – The image to shear.

Returns The transformed image(s) as a list of object(s) of type PIL.Image.

class `Augmentor.Operations.`**Skew**(*probability*, *skew_type*, *magnitude*)

This class is used to perform perspective skewing on images. It allows for skewing from a total of 12 different perspectives.

As well as the required `probability` parameter, the type of skew that is performed is controlled using a `skew_type` and a `magnitude` parameter. The `skew_type` controls the direction of the skew, while `magnitude` controls the degree to which the skew is performed.

To see examples of the various skews, see *Perspective Skewing*.

Images are skewed **in place** and an image of the same size is returned by this function. That is to say, that after a skew has been performed, the largest possible area of the same aspect ratio of the original image is cropped from the skewed image, and this is then resized to match the original image size. The *Perspective Skewing* section describes this in detail.

Parameters

- **probability** (*Float*) – Controls the probability that the operation is performed when it is invoked in the pipeline.
- **skew_type** (*String*) – Must be one of `TILT`, `TILT_TOP_BOTTOM`, `TILT_LEFT_RIGHT`, or `CORNER`.
  - `TILT` will randomly skew either left, right, up, or down. Left or right means it skews on the x-axis while up and down means that it skews on the y-axis.
  - `TILT_TOP_BOTTOM` will randomly skew up or down, or in other words skew along the y-axis.
  - `TILT_LEFT_RIGHT` will randomly skew left or right, or in other words skew along the x-axis.
  - `CORNER` will randomly skew one **corner** of the image either along the x-axis or y-axis. This means in one of 8 different directions, randomly.

  To see examples of the various skews, see *Perspective Skewing*.
- **magnitude** (*Integer*) – The degree to which the image is skewed.

perform_operation(*images*)

Perform the skew on the passed image(s) and returns the transformed image(s). Uses the `skew_type` and `magnitude` parameters to control the type of skew to perform as well as the degree to which it is performed.

If a list of images is passed, they must have identical dimensions. This is checked when we add the ground truth directory using `Pipeline.:func:`~Augmentor.Pipeline.Pipeline.ground_truth()` function.

However, if this check fails, the skew function will be skipped and a warning thrown, in order to avoid an exception.

Parameters **`images`** (*`List containing PIL.Image object(s)`*) – The image(s) to skew.

Returns The transformed image(s) as a list of object(s) of type PIL.Image.

class `Augmentor.Operations.`**`Zoom`**(*probability*, *min_factor*, *max_factor*)

This class is used to enlarge images (to zoom) but to return a cropped region of the zoomed image of the same size as the original image.

The amount of zoom applied is randomised, from between `min_factor` and `max_factor`. Set these both to the same value to always zoom by a constant factor.

Parameters

- **`probability`** (*`Float`*) – Controls the probability that the operation is performed when it is invoked in the pipeline.
- **`min_factor`** (*`Float`*) – The minimum amount of zoom to apply. Set both the `min_factor` and `min_factor` to the same values to zoom by a constant factor.
- **`max_factor`** (*`Float`*) – The maximum amount of zoom to apply. Set both the `min_factor` and `min_factor` to the same values to zoom by a constant factor.

`perform_operation`(*images*)

Zooms/scales the passed image(s) and returns the new image.

Parameters **`images`** (*`List containing PIL.Image object(s)`*) – The image(s) to be zoomed.

Returns The transformed image(s) as a list of object(s) of type PIL.Image.

class `Augmentor.Operations.`**`ZoomGroundTruth`**(*probability*, *min_factor*, *max_factor*)

This class is used to enlarge images (to zoom) but to return a cropped region of the zoomed image of the same size as the original image.

The amount of zoom applied is randomised, from between `min_factor` and `max_factor`. Set these both to the same value to always zoom by a constant factor.

Parameters

- **`probability`** (*`Float`*) – Controls the probability that the operation is performed when it is invoked in the pipeline.
- **`min_factor`** (*`Float`*) – The minimum amount of zoom to apply. Set both the `min_factor` and `min_factor` to the same values to zoom by a constant factor.
- **`max_factor`** (*`Float`*) – The maximum amount of zoom to apply. Set both the `min_factor` and `min_factor` to the same values to zoom by a constant factor.

`perform_operation`(*images*)

Zooms/scales the passed images and returns the new images.

Parameters **`images`** (*`List containing PIL.Image object(s)`*) – An arbitrarily long list of image(s) to be zoomed.

Returns The zoomed in image(s) as a list of PIL.Image object(s).

class `Augmentor.Operations.`**`ZoomRandom`**(*probability*, *percentage_area*, *randomise*)

This class is used to zoom into random areas of the image.

Zooms into a random area of the image, rather than the centre of the image, as is done by *`Zoom`*. The zoom factor is fixed unless `randomise` is set to `True`.

Parameters

- **`probability`** – Controls the probability that the operation is performed when it is invoked in the pipeline.
- **`percentage_area`** – A value between 0.1 and 1 that represents the area that will be cropped, with 1 meaning the entire area of the image will be cropped and 0.1 mean 10% of the area of the image will be cropped, before zooming.
- **`randomise`** – If `True`, uses the `percentage_area` as an upper bound, and randomises the zoom level from between 0.1 and `percentage_area`.

`perform_operation`(*images*)

Randomly zoom into the passed `images` by first cropping the image based on the `percentage_area` argument, and then resizing the image to match the size of the input area.

Effectively, you are zooming in on random areas of the image.

Parameters **`images`** (*`List containing PIL.Image object(s)`*) – The image to crop an area from.

Returns The transformed image(s) as a list of object(s) of type PIL.Image.

6.1.3 Documentation of the ImageUtilities module

The ImageUtilities module provides a number of helper functions, as well as the main *`AugmentorImage`* class, that is used throughout the package as a container class for images to be augmented.

`class` `Augmentor.ImageUtilities.`**`AugmentorImage`**(*image_path*, *output_directory*, *pil_images=None*, *array_images=None*, *path_images=None*, *class_label_int=None*)

Wrapper class containing paths to images, as well as a number of other parameters, that are used by the Pipeline and Operation modules to perform augmentation.

Each image that is found by Augmentor during the initialisation of a Pipeline object is contained with a new AugmentorImage object.

To initialise an AugmentorImage object for any image, the image's file path is required, as well as that image's output directory, which defines where any augmented images are stored.

Parameters

- **`image_path`** – The full path to an image.
- **`output_directory`** – The directory where augmented images for this image should be saved.

`categorical_label`

`class_label`

`class_label_int`

`file_format`

`ground_truth`

The *`ground_truth`* property contains an absolute path to the ground truth file for an image.

Getter Returns this image's ground truth file path.

Setter Sets this image's ground truth file path.

Type String

image_arrays

image_file_name

The *image_file_name* property contains the **file name** of the image contained in this instance. **There is no setter for this property.**

Getter Returns this image's file name.

Type String

image_path

The *image_path* property contains the absolute file path to the image.

Getter Returns this image's image path.

Setter Sets this image's image path

Type String

label

label_pair

output_directory

The *output_directory* property contains a path to the directory to which augmented images will be saved for this instance.

Getter Returns this image's output directory.

Setter Sets this image's output directory.

Type String

pil_images

`Augmentor.ImageUtilities.`**`extract_paths_and_extensions`**(*image_path*)

Extract an image's file name, its extension, and its root path (the image's absolute path without the file name).

Parameters **`image_path`** (*`String`*) – The path to the image.

Returns A 3-tuple containing the image's file name, extension, and root path.

`Augmentor.ImageUtilities.`**`parse_user_parameter`**(*user_param*)

`Augmentor.ImageUtilities.`**`scan`**(*source_directory*, *output_directory*)

`Augmentor.ImageUtilities.`**`scan_dataframe`**(*source_dataframe*, *image_col*, *category_col*, *output_directory*)

`Augmentor.ImageUtilities.`**`scan_directory`**(*source_directory*)

Scan a directory for images, returning any images found with the extensions `.jpg`, `.JPG`, `.jpeg`, `.JPEG`, `.gif`, `.GIF`, `.img`, `.IMG`, `.png`, `.PNG`, `.tif`, `.TIF`, `.tiff`, or `.TIFF`.

Parameters **`source_directory`** (*`String`*) – The directory to scan for images.

Returns A list of images found in the `source_directory`

`Augmentor.ImageUtilities.`**`scan_directory_with_classes`**(*source_directory*)

CHAPTER 7

Licence

7.1 Augmentor Licence

The Augmentor package is licenced under the terms of the MIT licence.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index