aleju / **imgaug**

Image augmentation for machine learning experiments. http://imgaug.readthedocs.io

#image-augmentation #machine-learning #augmentation #deep-learning #images #affine-transformation

198 commits | **1** branch | **6** releases | **4** contributors | MIT

Branch: **master** | **New pull request** | **Create new file** | **Upload files** | **Find file** | **Clone or download**

aleju Merge branch 'master' of https://github.com/aleju/imgaug — Latest commit `412500a` 4 days ago

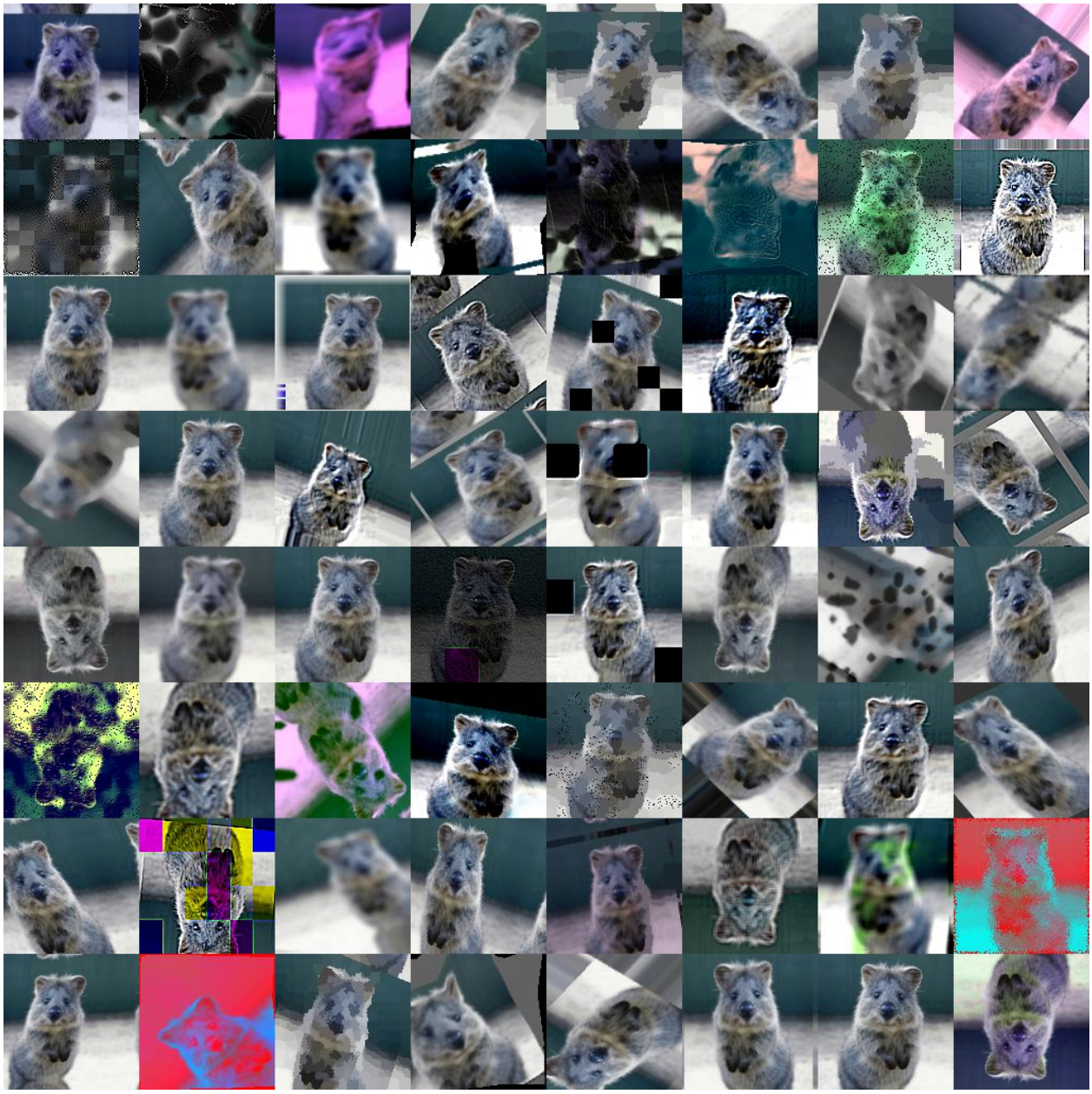
docs	Add documentation for alpha augmenters	8 days ago
images	Add new augmenters to readme	8 days ago
imgaug	Merge branch 'master' of https://github.com/aleju/imgaug	4 days ago
old_version	Move old version to subdirectory	10 months ago
tests	Fix PiecewiseAffine keypoint aug and contraint regular grid ; Fix #63	4 days ago
.gitignore	Add sphinx/readthedocs documentation	a month ago
.travis.yml	Refactor .travis.yml by spliting the sections	a month ago
LICENSE	Initial commit	2 years ago
MANIFEST.in	Fix bg aug errors; add bg aug tests; fix image not installed; change …	a month ago
README.md	Add new augmenters to readme	8 days ago
__init__.py	Add init file	10 months ago
examples.jpg	add PiecewiseAffine, improve example image generation plotting	4 months ago
examples_grid.jpg	Add new augmenters to readme	8 days ago
generate_documentation_images.py	Add documentation for alpha augmenters	8 days ago
generate_example_images.py	Add new augmenters to readme	8 days ago
setup.cfg	Add code for PyPI version	6 months ago
setup.py	Fix bg aug errors; add bg aug tests; fix image not installed; change …	a month ago

README.md

imgaug

This python library helps you with augmenting images for your machine learning projects. It converts a set of input images into a new, much larger set of slightly altered images.

build passing



Features:

- Most standard augmentation techniques available.
- Techniques can be applied to both images and keypoints/landmarks on images.
- Define your augmentation sequence once at the start of the experiment, then apply it many times.
- Define flexible stochastic ranges for each augmentation, e.g. "rotate each image by a value between -45 and 45 degrees" or "rotate each image by a value sampled from the normal distribution N(0, 5.0)".
- Easily convert all stochastic ranges to deterministic values in order to augment different batches of images in exactly the same way. (E.g. images and their respective heatmaps. If an image is rotated, you want its heatmap to be rotated by the exactly same amount.)
- Optionally run the augmentations in background processes, improving performance of experiments.

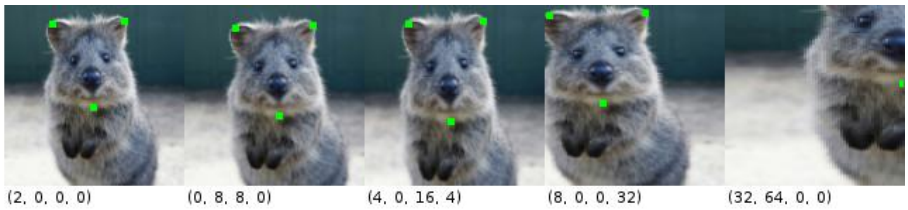
Documentation:

- http://imgaug.readthedocs.io/en/latest/source/examples_basics.html - Quick example code to use the library.
- http://imgaug.readthedocs.io/en/latest/source/augmenters.html - Example code for each augmentation technique.
- http://imgaug.readthedocs.io/en/latest/source/modules.html - API.
- This README contains more examples. See further below.

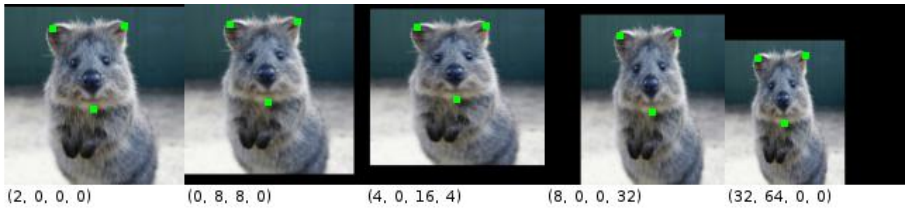
The images below show examples for most augmentation techniques (values written in the form `(a, b)` mean that a value was randomly picked from the range `a <= x <= b`):

Noop

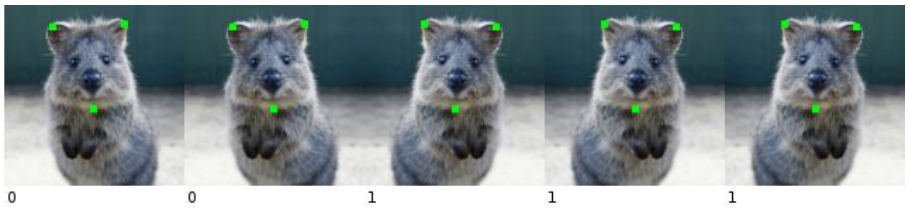
Crop
(top, right,
bottom, left)



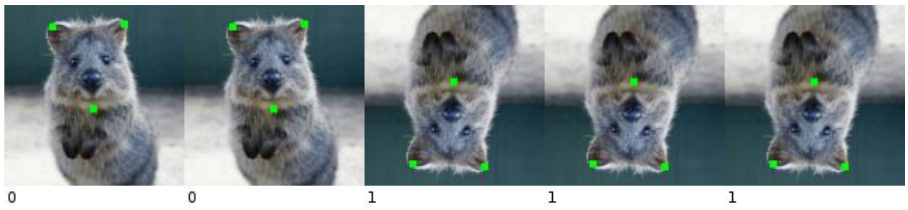
Pad
(top, right,
bottom, left)



Fliplr

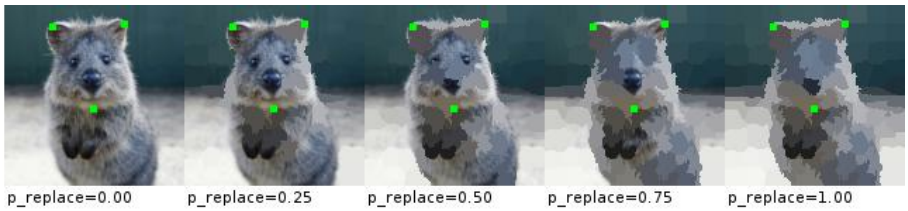


Flipud

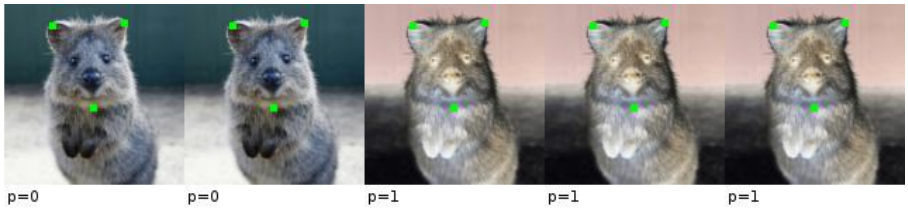


Superpixels
p_replace=1

Superpixels
n_segments=100

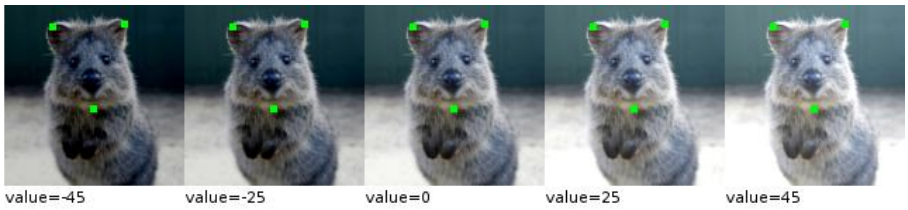


Invert



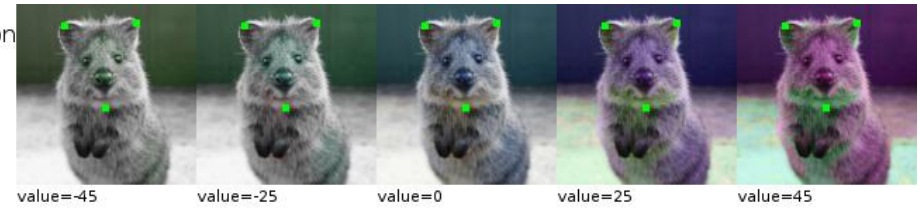
Invert (per_channel)

Add



Add (per channel)

AddToHueAndSaturation

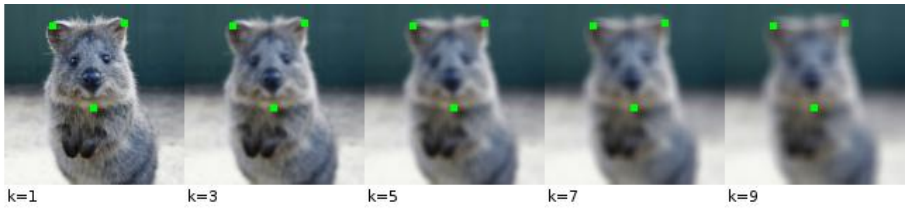


Multiply

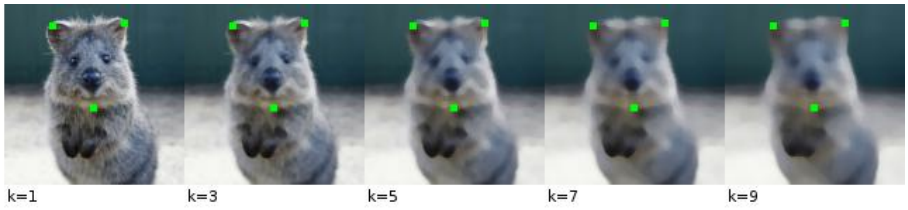
Multiply (per channel)

GaussianBlur

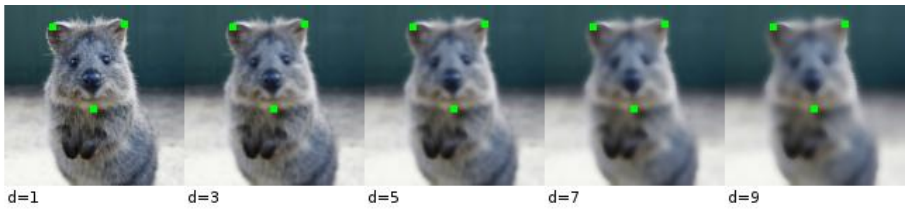
AverageBlur



MedianBlur

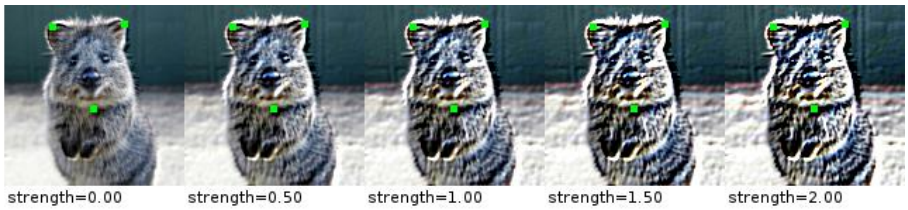


BilateralBlur
sigma_color=250,
sigma_space=250

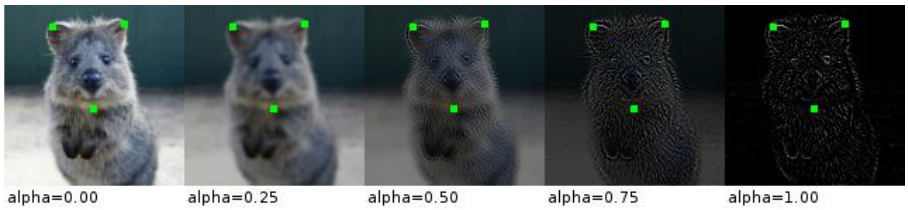


Sharpen
(alpha=1)

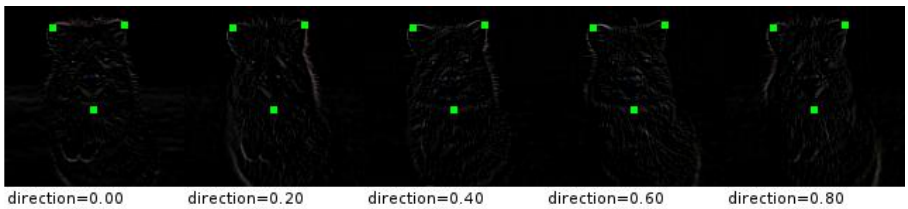
Emboss
(alpha=1)



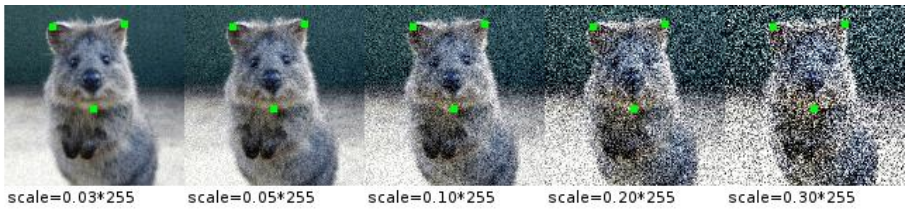
EdgeDetect



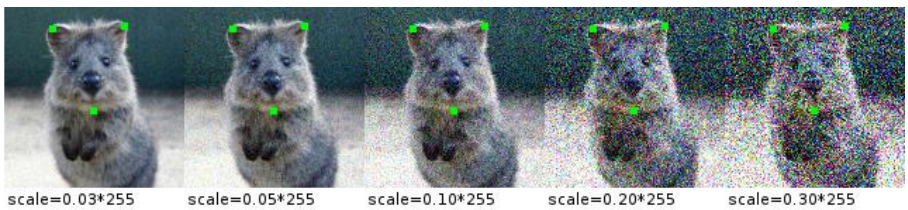
DirectedEdgeDetect
(alpha=1)



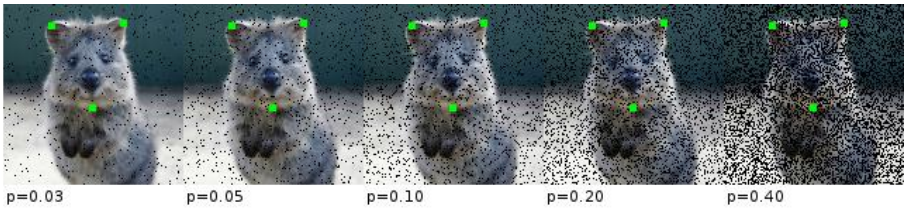
AdditiveGaussianNoise



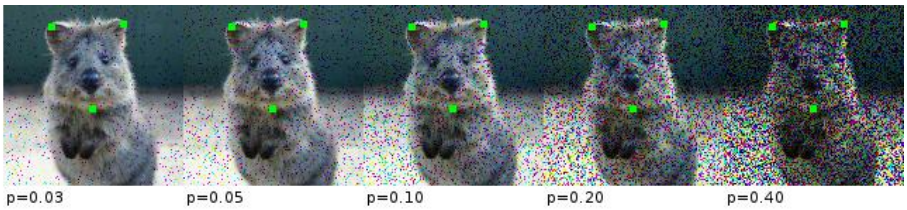
AdditiveGaussianNoise
(per channel)



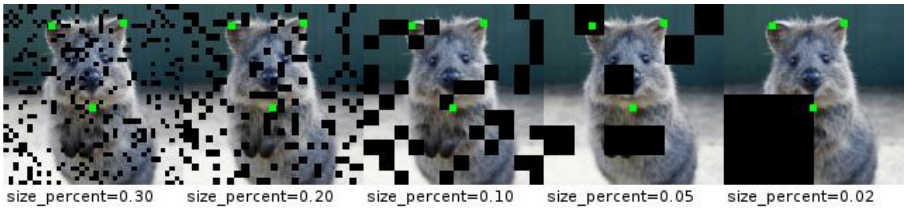
Dropout



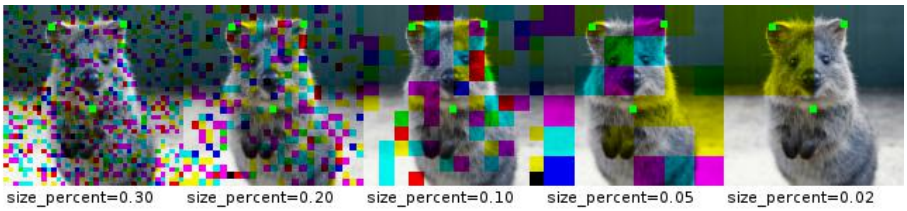
Dropout
(per channel)



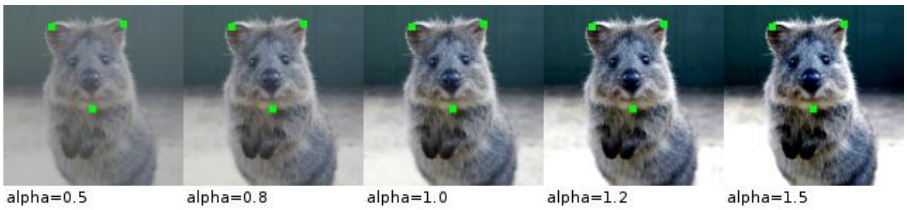
CoarseDropout
(p=0.2)



CoarseDropout
(p=0.2, per channel)



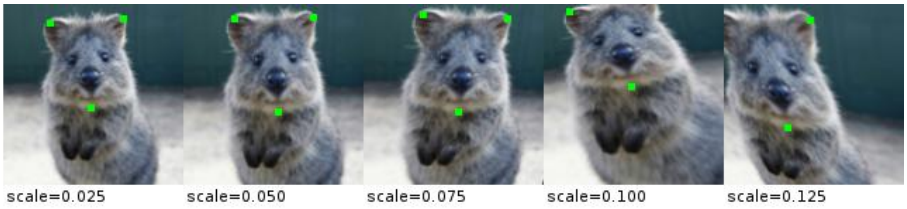
ContrastNormalization



ContrastNormalization
(per channel)

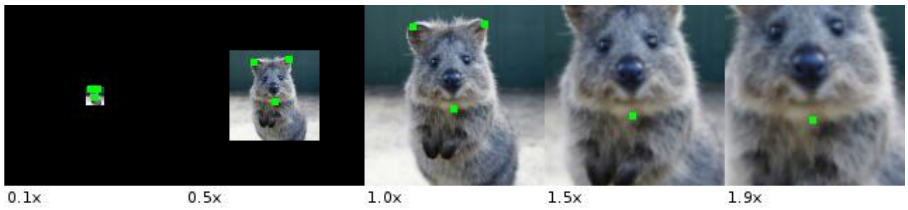
Grayscale

PerspectiveTransform

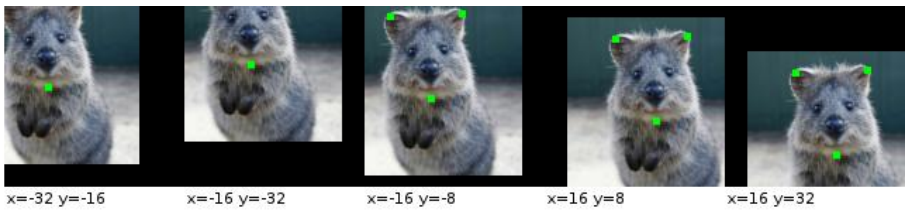


PiecewiseAffine

Affine: Scale



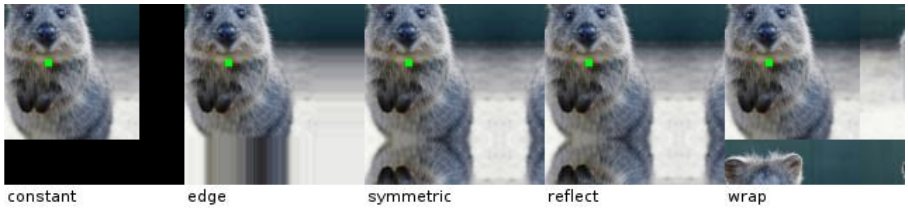
Affine: Translate



Affine: Rotate

Affine: Shear

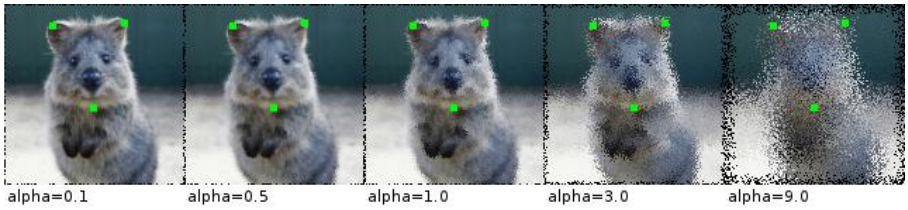
Affine: Modes



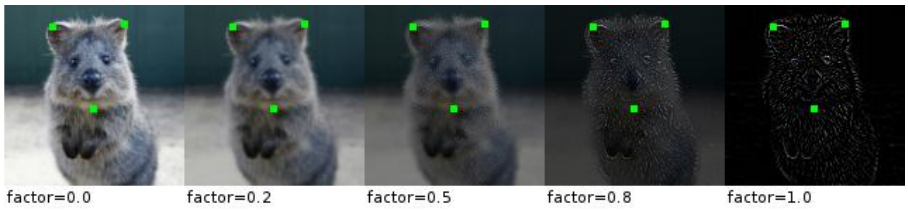
Affine: cval

Affine: all

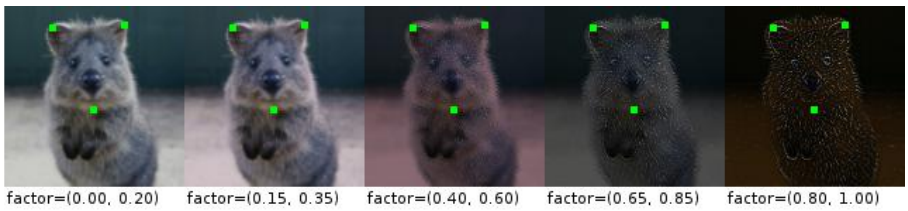
ElasticTransformation (sigma=0.2)



Alpha with EdgeDetect(1.0)

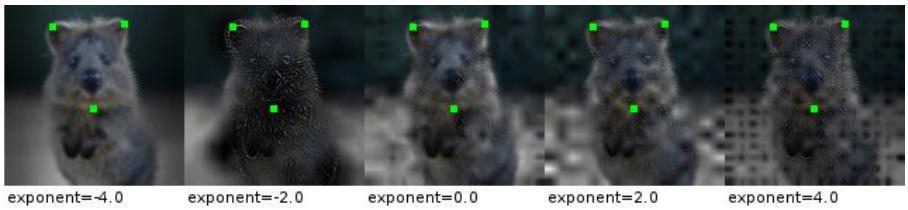


Alpha with EdgeDetect(1.0) (per channel)



SimplexNoiseAlpha with EdgeDetect(1.0)

FrequencyNoiseAlpha with EdgeDetect(1.0)



Requirements and installation

Required packages:

- six
- numpy
- scipy
- scikit-image (`pip install -U scikit-image`)
- OpenCV (i.e. `cv2`)

OpenCV has to be manually installed. The other package should auto-install themselves.

To install, simply use `sudo pip install imgaug` . That version might be outdated though. To always get the newest version directly from github use `sudo pip install git+https://github.com/aleju/imgaug` . Alternatively, you can download the repository via `git clone https://github.com/aleju/imgaug` and install by using `python setup.py sdist && sudo pip install dist/imgaug-0.2.4.tar.gz` .

To deinstall the library, just execute `sudo pip uninstall imgaug` .

The library is currently only tested in python2.7, but the code is written so that it *should* run in python3 too.

Examples

A standard machine learning situation. Train on batches of images and augment each batch via crop, horizontal flip ("Fliplr") and gaussian blur:

```
from imgaug import augmenters as iaa

seq = iaa.Sequential([
    iaa.Crop(px=(0, 16)), # crop images from each side by 0 to 16px (randomly chosen)
    iaa.Fliplr(0.5), # horizontally flip 50% of the images
    iaa.GaussianBlur(sigma=(0, 3.0)) # blur images with a sigma of 0 to 3.0
])

for batch_idx in range(1000):
    # 'images' should be either a 4D numpy array of shape (N, height, width, channels)
    # or a list of 3D numpy arrays, each having shape (height, width, channels).
    # Grayscale images must have shape (height, width, 1) each.
    # All images must have numpy's dtype uint8. Values are expected to be in
    # range 0-255.
    images = load_batch(batch_idx)
    images_aug = seq.augment_images(images)
    train_on_images(images_aug)
```

Apply heavy augmentations to images (used to create the image at the very top of this readme):

```
import imgaug as ia
from imgaug import augmenters as iaa
import numpy as np

# random example images
images = np.random.randint(0, 255, (16, 128, 128, 3), dtype=np.uint8)

# Sometimes(0.5, ...) applies the given augmenter in 50% of all cases,
# e.g. Sometimes(0.5, GaussianBlur(0.3)) would blur roughly every second image.
sometimes = lambda aug: iaa.Sometimes(0.5, aug)

# Define our sequence of augmentation steps that will be applied to every image
# All augmenters with per_channel=0.5 will sample one value _per image_
# in 50% of all cases. In all other cases they will sample new values
# _per channel_.
seq = iaa.Sequential(
    [
        # apply the following augmenters to most images
        iaa.Fliplr(0.5), # horizontally flip 50% of all images
        iaa.Flipud(0.2), # vertically flip 20% of all images
        # crop images by -5% to 10% of their height/width
        sometimes(iaa.CropAndPad(
            percent=(-0.05, 0.1),
            pad_mode=ia.ALL,
            pad_cval=(0, 255)
        )),
        sometimes(iaa.Affine(
            scale={"x": (0.8, 1.2), "y": (0.8, 1.2)}, # scale images to 80-120% of their size, individually
            translate_percent={"x": (-0.2, 0.2), "y": (-0.2, 0.2)}, # translate by -20 to +20 percent (per
            rotate=(-45, 45), # rotate by -45 to +45 degrees
            shear=(-16, 16), # shear by -16 to +16 degrees
            order=[0, 1], # use nearest neighbour or bilinear interpolation (fast)
            cval=(0, 255), # if mode is constant, use a cval between 0 and 255
            mode=ia.ALL # use any of scikit-image's warping modes (see 2nd image from the top for examples)
        )),
        # execute 0 to 5 of the following (less important) augmenters per image
        # don't execute all of them, as that would often be way too strong
        iaa.SomeOf((0, 5),
            [
                sometimes(iaa.Superpixels(p_replace=(0, 1.0), n_segments=(20, 200))), # convert images into
                iaa.OneOf([
                    iaa.GaussianBlur((0, 3.0)), # blur images with a sigma between 0 and 3.0
                    iaa.AverageBlur(k=(2, 7)), # blur image using local means with kernel sizes between 2 a
                    iaa.MedianBlur(k=(3, 11)), # blur image using local medians with kernel sizes between 2
                ]),
                iaa.Sharpen(alpha=(0, 1.0), lightness=(0.75, 1.5)), # sharpen images
                iaa.Emboss(alpha=(0, 1.0), strength=(0, 2.0)), # emboss images
                # search either for all edges or for directed edges,
                # blend the result with the original image using a blobby mask
                iaa.SimplexNoiseAlpha(iaa.OneOf([
                    iaa.EdgeDetect(alpha=(0.5, 1.0)),
                    iaa.DirectedEdgeDetect(alpha=(0.5, 1.0), direction=(0.0, 1.0)),
                ])),
                iaa.AdditiveGaussianNoise(loc=0, scale=(0.0, 0.05*255), per_channel=0.5), # add gaussian no
                iaa.OneOf([
```

```
                    iaa.Dropout((0.01, 0.1), per_channel=0.5), # randomly remove up to 10% of the pixels
                    iaa.CoarseDropout((0.03, 0.15), size_percent=(0.02, 0.05), per_channel=0.2),
                ]),
                iaa.Invert(0.05, per_channel=True), # invert color channels
                iaa.Add((-10, 10), per_channel=0.5), # change brightness of images (by -10 to 10 of origina
                iaa.AddToHueAndSaturation((-20, 20)), # change hue and saturation
                # either change the brightness of the whole image (sometimes
                # per channel) or change the brightness of subareas
                iaa.OneOf([
                    iaa.Multiply((0.5, 1.5), per_channel=0.5),
                    iaa.FrequencyNoiseAlpha(
                        exponent=(-4, 0),
                        first=iaa.Multiply((0.5, 1.5), per_channel=True),
                        second=iaa.ContrastNormalization((0.5, 2.0))
                    )
                ]),
                iaa.ContrastNormalization((0.5, 2.0), per_channel=0.5), # improve or worsen the contrast
                iaa.Grayscale(alpha=(0.0, 1.0)),
                sometimes(iaa.ElasticTransformation(alpha=(0.5, 3.5), sigma=0.25)), # move pixels locally a
                sometimes(iaa.PiecewiseAffine(scale=(0.01, 0.05))), # sometimes move parts of the image aro
                sometimes(iaa.PerspectiveTransform(scale=(0.01, 0.1)))
            ],
            random_order=True
        )
    ],
    random_order=True
)

images_aug = seq.augment_images(images)
```

Quickly show example results of your augmentation sequence:

```
from imgaug import augmenters as iaa
import numpy as np

images = np.random.randint(0, 255, (16, 128, 128, 3), dtype=np.uint8)
seq = iaa.Sequential([iaa.Fliplr(0.5), iaa.GaussianBlur((0, 3.0))])

# show an image with 8*8 augmented versions of image 0
seq.show_grid(images[0], cols=8, rows=8)

# Show an image with 8*8 augmented versions of image 0 and 8*8 augmented
# versions of image 1. The identical augmentations will be applied to
# image 0 and 1.
seq.show_grid([images[0], images[1]], cols=8, rows=8)
```

Augment two batches of images in *exactly the same way* (e.g. horizontally flip 1st, 2nd and 5th images in both batches, but do not alter 3rd and 4th images):

```
from imgaug import augmenters as iaa

# Standard scenario: You have N RGB-images and additionally 21 heatmaps per image.
# You want to augment each image and its heatmaps identically.
images = np.random.randint(0, 255, (16, 128, 128, 3), dtype=np.uint8)
heatmaps = np.random.randint(0, 255, (16, 128, 128, 21), dtype=np.uint8)

seq = iaa.Sequential([iaa.GaussianBlur((0, 3.0)), iaa.Affine(translate_px={"x": (-40, 40)})])

# Convert the stochastic sequence of augmenters to a deterministic one.
# The deterministic sequence will always apply the exactly same effects to the images.
seq_det = seq.to_deterministic() # call this for each batch again, NOT only once at the start
images_aug = seq_det.augment_images(images)
heatmaps_aug = seq_det.augment_images(heatmaps)
```

Augment images *and* **landmarks/keypoints** on these images:

```
import imgaug as ia
from imgaug import augmenters as iaa
from scipy import misc
import random
import numpy as np
images = np.random.randint(0, 50, (4, 128, 128, 3), dtype=np.uint8)
```

```
# Generate random keypoints.
# The augmenters expect a list of imgaug.KeypointsOnImage.
keypoints_on_images = []
for image in images:
    height, width = image.shape[0:2]
    keypoints = []
    for _ in range(4):
        x = random.randint(0, width-1)
        y = random.randint(0, height-1)
        keypoints.append(ia.Keypoint(x=x, y=y))
    keypoints_on_images.append(ia.KeypointsOnImage(keypoints, shape=image.shape))

seq = iaa.Sequential([iaa.GaussianBlur((0, 3.0)), iaa.Affine(scale=(0.5, 0.7))])
seq_det = seq.to_deterministic() # call this for each batch again, NOT only once at the start

# augment keypoints and images
images_aug = seq_det.augment_images(images)
keypoints_aug = seq_det.augment_keypoints(keypoints_on_images)

# Example code to show each image and print the new keypoints coordinates
for img_idx, (image_before, image_after, keypoints_before, keypoints_after) in enumerate(zip(images, images
    image_before = keypoints_before.draw_on_image(image_before)
    image_after = keypoints_after.draw_on_image(image_after)
    misc.imshow(np.concatenate((image_before, image_after), axis=1)) # before and after
    for kp_idx, keypoint in enumerate(keypoints_after.keypoints):
        keypoint_old = keypoints_on_images[img_idx].keypoints[kp_idx]
        x_old, y_old = keypoint_old.x, keypoint_old.y
        x_new, y_new = keypoint.x, keypoint.y
        print("[Keypoints for image #%d] before aug: x=%d y=%d | after aug: x=%d y=%d" % (img_idx, x_old, y
```

Apply single augmentations to images:

```
from imgaug import augmenters as iaa
import numpy as np
images = np.random.randint(0, 255, (16, 128, 128, 3), dtype=np.uint8)

flipper = iaa.Fliplr(1.0) # always horizontally flip each input image
images[0] = flipper.augment_image(images[0]) # horizontally flip image 0

vflipper = iaa.Flipud(0.9) # vertically flip each input image with 90% probability
images[1] = vflipper.augment_image(images[1]) # probably vertically flip image 1

blurer = iaa.GaussianBlur(3.0)
images[2] = blurer.augment_image(images[2]) # blur image 2 by a sigma of 3.0
images[3] = blurer.augment_image(images[3]) # blur image 3 by a sigma of 3.0 too

translater = iaa.Affine(translate_px={"x": -16}) # move each input image by 16px to the left
images[4] = translater.augment_image(images[4]) # move image 4 to the left

scaler = iaa.Affine(scale={"y": (0.8, 1.2)}) # scale each input image to 80-120% on the y axis
images[5] = scaler.augment_image(images[5]) # scale image 5 by 80-120% on the y axis
```

Apply an augmenter to only specific image channels:

```
from imgaug import augmenters as iaa
import numpy as np

# fake RGB images
images = np.random.randint(0, 255, (16, 128, 128, 3), dtype=np.uint8)

# add a random value from the range (-30, 30) to the first two channels of
# input images (e.g. to the R and G channels)
aug = iaa.WithChannels(
  channels=[0, 1],
  children=iaa.Add((-30, 30))
)

images_aug = aug.augment_images(images)
```

You can use more unusual distributions for the stochastic parameters of each augmenter:

```
from imgaug import augmenters as iaa
from imgaug import parameters as iap
import numpy as np
images = np.random.randint(0, 255, (16, 128, 128, 3), dtype=np.uint8)

# Blur by a value sigma which is sampled from a uniform distribution
# of range 0.1 <= x < 3.0.
# The convenience shortcut for this is: iaa.GaussianBlur((0.1, 3.0))
blurer = iaa.GaussianBlur(iap.Uniform(0.1, 3.0))
images_aug = blurer.augment_images(images)

# Blur by a value sigma which is sampled from a normal distribution N(1.0, 0.1),
# i.e. sample a value that is usually around 1.0.
# Clip the resulting value so that it never gets below 0.1 or above 3.0.
blurer = iaa.GaussianBlur(iap.Clip(iap.Normal(1.0, 0.1), 0.1, 3.0))
images_aug = blurer.augment_images(images)

# Same again, but this time the mean of the normal distribution is not constant,
# but comes itself from a uniform distribution between 0.5 and 1.5.
blurer = iaa.GaussianBlur(iap.Clip(iap.Normal(iap.Uniform(0.5, 1.5), 0.1), 0.1, 3.0))
images_aug = blurer.augment_images(images)

# Use for sigma one of exactly three allowed values: 0.5, 1.0 or 1.5.
blurer = iaa.GaussianBlur(iap.Choice([0.5, 1.0, 1.5]))
images_aug = blurer.augment_images(images)

# Sample sigma from a discrete uniform distribution of range 1 <= sigma <= 5,
# i.e. sigma will have any of the following values: 1, 2, 3, 4, 5.
blurer = iaa.GaussianBlur(iap.DiscreteUniform(1, 5))
images_aug = blurer.augment_images(images)
```

You can **dynamically deactivate augmenters** in an already defined sequence:

```
import imgaug as ia
from imgaug import augmenters as iaa
import numpy as np

# images and heatmaps, just arrays filled with value 30
images = np.ones((16, 128, 128, 3), dtype=np.uint8) * 30
heatmaps = np.ones((16, 128, 128, 21), dtype=np.uint8) * 30

# add vertical lines to see the effect of flip
images[:, 16:128-16, 120:124, :] = 120
heatmaps[:, 16:128-16, 120:124, :] = 120

seq = iaa.Sequential([
  iaa.Fliplr(0.5, name="Flipper"),
  iaa.GaussianBlur((0, 3.0), name="GaussianBlur"),
  iaa.Dropout(0.02, name="Dropout"),
  iaa.AdditiveGaussianNoise(scale=0.01*255, name="MyLittleNoise"),
  iaa.AdditiveGaussianNoise(loc=32, scale=0.0001*255, name="SomeOtherNoise"),
  iaa.Affine(translate_px={"x": (-40, 40)}, name="Affine")
])

# change the activated augmenters for heatmaps,
# we only want to execute horizontal flip, affine transformation and one of
# the gaussian noises
def activator_heatmaps(images, augmenter, parents, default):
    if augmenter.name in ["GaussianBlur", "Dropout", "MyLittleNoise"]:
        return False
    else:
        # default value for all other augmenters
        return default
hooks_heatmaps = ia.HooksImages(activator=activator_heatmaps)

seq_det = seq.to_deterministic() # call this for each batch again, NOT only once at the start
images_aug = seq_det.augment_images(images)
heatmaps_aug = seq_det.augment_images(heatmaps, hooks=hooks_heatmaps)
```

Images can be augmented in **background processes** using the method `augment_batches(batches, background=True)`, where `batches` is expected to be a list of image batches or a list of batches/lists of `imgaug.KeypointsOnImage` or a list of `imgaug.Batch`. The following example augments a list of image batches in the background:

```
import imgaug as ia
from imgaug import augmenters as iaa
import numpy as np
from skimage import data

# Number of batches and batch size for this example
nb_batches = 10
batch_size = 32

# Example augmentation sequence to run in the background
augseq = iaa.Sequential([
    iaa.Fliplr(0.5),
    iaa.CoarseDropout(p=0.1, size_percent=0.1)
])

# For simplicity, we use the same image here many times
astronaut = data.astronaut()
astronaut = ia.imresize_single_image(astronaut, (64, 64))

# Make batches out of the example image (here: 10 batches, each 32 times
# the example image)
batches = []
for _ in range(nb_batches):
    batches.append(
        np.array(
            [astronaut for _ in range(batch_size)],
            dtype=np.uint8
        )
    )

# Show the augmented images.
# Note that augment_batches() returns a generator.
for images_aug in augseq.augment_batches(batches, background=True):
    misc.imshow(ia.draw_grid(images_aug, cols=8))
```

Images can also be augmented in background processes using the classes `imgaug.BatchLoader` and `imgaug.BackgroundAugmenter`, which offer a bit more flexibility. (`augment_batches()` is a wrapper around these.) Using these classes is good practice, when you have a lot of images that you don't want to load at the same time.

```
import imgaug as ia
from imgaug import augmenters as iaa
import numpy as np
from skimage import data

# Example augmentation sequence to run in the background.
augseq = iaa.Sequential([
    iaa.Fliplr(0.5),
    iaa.CoarseDropout(p=0.1, size_percent=0.1)
])

# A generator that loads batches from the hard drive.
def load_batches():
    # Here, load 10 batches of size 4 each.
    # You can also load an infinite amount of batches, if you don't train
    # in epochs.
    batch_size = 4
    nb_batches = 10

    # Here, for simplicity we just always use the same image.
    astronaut = data.astronaut()
    astronaut = ia.imresize_single_image(astronaut, (64, 64))

    for i in range(nb_batches):
        # A list containing all images of the batch.
        batch_images = []
        # A list containing IDs of images in the batch. This is not necessary
        # for the background augmentation and here just used to showcase that
        # you can transfer additional information.
        batch_data = []

        # Add some images to the batch.
        for b in range(batch_size):
            batch_images.append(astronaut)
            batch_data.append((i, b))
```

```
            # Create the batch object to send to the background processes.
            batch = ia.Batch(
                images=np.array(batch_images, dtype=np.uint8),
                data=batch_data
            )

            yield batch

    # background augmentation consists of two components:
    #  (1) BatchLoader, which runs in a Thread and calls repeatedly a user-defined
    #      function (here: load_batches) to load batches (optionally with keypoints
    #      and additional information) and sends them to a queue of batches.
    #  (2) BackgroundAugmenter, which runs several background processes (on other
    #      CPU cores). Each process takes batches from the queue defined by (1),
    #      augments images/keypoints and sends them to another queue.
    # The main process can then read augmented batches from the queue defined
    # by (2).
    batch_loader = ia.BatchLoader(load_batches)
    bg_augmenter = ia.BackgroundAugmenter(batch_loader, augseq)

    # Run until load_batches() returns nothing anymore. This also allows infinite
    # training.
    while True:
        print("Next batch...")
        batch = bg_augmenter.get_batch()
        if batch is None:
            print("Finished epoch.")
            break
        images_aug = batch.images_aug

        print("Image IDs: ", batch.data)

        misc.imshow(np.hstack(list(images_aug)))

    batch_loader.terminate()
    bg_augmenter.terminate()
```

List of augmenters

The following is a list of available augmenters. Note that most of the below mentioned variables can be set as ranges, e.g. `A=(0.0, 1.0)` to sample a random value between 0 and 1.0 per image.

Augmenter	Description
Sequential(C, R)	Takes a list of child augmenters `C` and applies them in that order to images. If `R` is true (default: false), then the order is random (chosen once per batch).
SomeOf(N, C, R)	Applies `N` randomly selected augmenters from from a list of augmenters `C` to each image. The augmenters are chosen per image. `R` is the same as for `Sequential`. `N` can be a range, e.g. `(1, 3)` in order to pick 1 to 3.
OneOf(C)	Identical to `SomeOf(1, C)`.
Sometimes(P, C, D)	Augments images with probability `P` by using child augmenters `C`, otherwise uses `D`. `D` can be None, then only `P` percent of all images are augmented via `C`.
WithColorspace(T, F, C)	Transforms images from colorspace `F` (default: RGB) to colorspace `T`, applies augmenters `C` and then converts back to `F`.
WithChannels(H, C)	Selects from each image channels `H` (e.g. `[0,1]` for red and green in RGB images), applies child augmenters `C` to these channels and merges the result back into the original images.
Noop()	Does nothing. (Useful for validation/test.)
Lambda(I, K)	Applies lambda function `I` to images and `K` to keypoints.
AssertLambda(I, K)	Checks images via lambda function `I` and keypoints via `K` and raises an error if false is returned by either of them.
AssertShape(S)	Raises an error if input images are not of shape `S`.

Augmenter	Description
Scale(S, I)	Resizes images to size `S`. Common use case would be to use `S={"height":H, "width":W}` to resize all images to shape `HxW`. `H` and `W` may be floats (e.g. resize to `50%` of original size). Either `H` or `W` may be `"keep-aspect-ratio"` to define only one side's new size and resize the other side correspondingly. `I` is the interpolation to use (default: `cubic`).
CropAndPad(PX, PC, PM, PCV, KS)	Crops away or pads `PX` pixels or `PC` percent of pixels at top/right/bottom/left of images. Negative values result in cropping, positive in padding. `PM` defines the pad mode (e.g. use uniform color for all added pixels). `PCV` controls the color of added pixels if `PM=constant`. If `KS` is true (default), the resulting image is resized back to the original size.
Pad(PX, PC, PM, PCV, KS)	Shortcut for CropAndPad(), which only adds pixels. Only positive values are allowed for `PX` and `PC`.
Crop(PX, PC, KS)	Shortcut for CropAndPad(), which only crops away pixels. Only positive values are allowed for `PX` and `PC` (e.g. a value of 5 results in 5 pixels cropped away).
Fliplr(P)	Horizontally flips images with probability `P`.
Flipud(P)	Vertically flips images with probability `P`.
Superpixels(P, N, M)	Generates N superpixels of the image at (max) resolution M and resizes back to the original size. Then `P` percent of all superpixel areas in the original image are replaced by the superpixel. (1-P) percent remain unaltered.
ChangeColorspace(T, F, A)	Converts images from colorspace `F` to `T` and mixes with the original image using alpha `A`. Grayscale remains at three channels. (Fairly untested augmenter, use at own risk.)
Grayscale(A, F)	Converts images from colorspace F (default: RGB) to grayscale and mixes with the original image using alpha `A`.
GaussianBlur(S)	Blurs images using a gaussian kernel with size `S`.
AverageBlur(K)	Blurs images using a simple averaging kernel with size `K`.
MedianBlur(K)	Blurs images using a median over neihbourhoods of size `K`.
BilateralBlur(D, SC, SS)	Blurs images using a bilateral filter with distance `D` (like kernel size). `SC` is a sigma for the (influence) distance in color space, `SS` a sigma for the spatial distance.
Convolve(M)	Convolves images with matrix `M`, which can be a lambda function.
Sharpen(A, L)	Runs a sharpening kernel over each image with lightness `L` (low values result in dark images). Mixes the result with the original image using alpha `A`.
Emboss(A, S)	Runs an emboss kernel over each image with strength `S`. Mixes the result with the original image using alpha `A`.
EdgeDetect(A)	Runs an edge detection kernel over each image. Mixes the result with the original image using alpha `A`.
DirectedEdgeDetect(A, D)	Runs a directed edge detection kernel over each image, which detects each from direction `D` (default: random direction from 0 to 360 degrees, chosen per image). Mixes the result with the original image using alpha `A`.
Add(V, PCH)	Adds value `V` to each image. If `PCH` is true, then the the sampled values may be different per channel.
AddElementwise(V, PCH)	Adds value `V` to each pixel. If `PCH` is true, then the the sampled values may be different per channel (and pixel).
AddToHueAndSaturation(V, PCH, F, C)	Adds value `V` to each pixel in HSV space (i.e. modifying hue and saturation). Converts from colorspace F to HSV (default is F=RGB). Selects channels C before augmenting (default is C=[0,1]). If `PCH` is true, then the the sampled values may be different per channel.

Augmenter	Description
AdditiveGaussianNoise(L, S, PCH)	Adds white/gaussian noise pixelwise to an image. The noise comes from the normal distribution `N(L,S)`. If `PCH` is true, then the sampled values may be different per channel (and pixel).
Multiply(V, PCH)	Multiplies each image by value `V`, leading to darker/brighter images. If `PCH` is true, then the sampled values may be different per channel.
MultiplyElementwise(V, PCH)	Multiplies each pixel by value `V`, leading to darker/brighter pixels. If `PCH` is true, then the sampled values may be different per channel (and pixel).
Dropout(P, PCH)	Sets pixels to zero with probability `P`. If `PCH` is true, then channels may be treated differently, otherwise whole pixels are set to zero.
CoarseDropout(P, SPX, SPC, PCH)	Like `Dropout`, but samples the locations of pixels that are to be set to zero from a coarser/smaller image, which has pixel size `SPX` or relative size `SPC`. I.e. if `SPC` has a small value, the coarse map is small, resulting in large rectangles being dropped.
Invert(P, PCH)	Inverts with probability `P` all pixels in an image, i.e. sets them to (1-pixel_value). If `PCH` is true, each channel is treated individually (leading to only some channels being inverted).
ContrastNormalization(S, PCH)	Changes the contrast in images, by moving pixel values away or closer to 128. The direction and strength is defined by `S`. If `PCH` is set to true, the process happens channel-wise with possibly different `S`.
Affine(S, TPX, TPC, R, SH, O, M, CVAL)	Applies affine transformations to images. Scales them by `S` (>1=zoom in, <1=zoom out), translates them by `TPX` pixels or `TPC` percent, rotates them by `R` degrees and shears them by `SH` degrees. Interpolation happens with order `O` (0 or 1 are good and fast). Areas can appear in the resulting image, which have no corresponding area in the original image. `M` defines, how to handle these. If `M='constant'` then `CVAL` defines a constant value with which to fill the area.
PiecewiseAffine(S, R, C, O, M, CVAL)	Places a regular grid of points on the image. The grid has `R` rows and `C` columns. Then moves the points (and the image areas around them) by amounts that are samples from normal distribution N(0, `S`), leading to local distortions of varying strengths. `O`, `M` and `CVAL` are defined as in `Affine`.
PerspectiveTransform(S, KS)	Applies a random four-point perspective transform to the image (kinda like an advanced form of cropping). Each point has a random distance from the image corner, derived from a normal distribution with sigma `S`. If `KS` is set to True (default), each image will be resized back to its original size.
ElasticTransformation(S, SM)	Moves each pixel individually around based on distortion fields. `SM` defines the smoothness of the distortion field and `S` its strength.
Alpha(F, A, B, PCH)	Augments images using augmenters `A` and `B` independently, then overlays the result using alpha `F`. Both `A` and `B` default to doing nothing if not provided. E.g. use `Alpha(0.9, A)` to augment images via `A`, then blend the result, keeping 10% of the original image (before `A`). If `PCH` is set to true, the process happens channel-wise with possibly different `F` (`A` and `B` are computed once per image).
AlphaElementwise(F, A, B, PCH)	Same as `Alpha`, but performs the blending pixel-wise using a continuous mask (values 0.0 to 1.0) sampled from `F`. If `PCH` is set to true, the process happens both pixel- and channel-wise.
SimplexNoiseAlpha(A, B, PCH, SM, UP, I, AGG, SIG, SIGT)	Similar to `Alpha`, but uses a mask to blend the results from augmenters `A` and `B`. The mask is sampled from simplex noise, which tends to be blobby. The mask is gathered in `I` iterations (default 1-3), each iteration is combined using aggregation method `AGG` (default max, i.e. maximum value from all iterations per pixel). Each mask is sampled in low resolution space with max resolution `SM` (default 2 to 16px) and upscaled to image size using method `UP` (default: linear or cubic or nearest neighbour upsampling). If `SIG` is true, a sigmoid is applied to the mask with threshold `SIGT`, which makes the blobs have values closer to 0.0 or 1.0.

Augmenter	Description
FrequencyNoiseAlpha(E, A, B, PCH, SM, UP, I, AGG, SIG, SIGT)	Similar to `SimplexNoiseAlpha`, but generates noise masks from the frequency domain. Exponent `E` is used to increase/decrease frequency components. High values lead to more pronounced high frequency components. Use values in the range -4 to 4, with -2 roughly generated cloud-like patterns.