albumentations Documentation

Release 0.3.0

Buslaev Alexander, Alexander Parinov, Vladimir Iglovikov

Jun 26, 2019

Contents

albumentations is a fast image augmentation library and easy to use wrapper around other libraries.

CHAPTER 1

Features

- Great fast augmentations based on highly-optimized OpenCV library.
- Super simple yet powerful interface for different tasks like (segmentation, detection, etc).
- Easy to customize.
- Easy to add other frameworks.

CHAPTER 2

Project info

- GitHub repository: https://github.com/albu/albumentations
- License: MIT

CHAPTER 3

Installation

You can use pip to install albumentations:

```
pip install albumentations
```

If you want to get the latest version of the code before it is released on PyPI you can install the library from GitHub:

```
pip install -U git+https://github.com/albu/albumentations
```

CHAPTER 4

Demo

You can use this Google Colaboratory notebook to adjust image augmentation parameters and see the resulting images.

4.1 Examples

```
from albumentations import (
    HorizontalFlip, IAAPerspective, ShiftScaleRotate, CLAHE, RandomRotate90,
    Transpose, ShiftScaleRotate, Blur, OpticalDistortion, GridDistortion,␣
↪HueSaturationValue,
    IAAAdditiveGaussianNoise, GaussNoise, MotionBlur, MedianBlur, IAAPiecewiseAffine,
    IAASharpen, IAAEmboss, RandomBrightnessContrast, Flip, OneOf, Compose
)
import numpy as np

def strong_aug(p=0.5):
    return Compose([
        RandomRotate90(),
        Flip(),
        Transpose(),
        OneOf([
            IAAAdditiveGaussianNoise(),
            GaussNoise(),
        ], p=0.2),
        OneOf([
            MotionBlur(p=0.2),
            MedianBlur(blur_limit=3, p=0.1),
            Blur(blur_limit=3, p=0.1),
        ], p=0.2),
        ShiftScaleRotate(shift_limit=0.0625, scale_limit=0.2, rotate_limit=45, p=0.2),
        OneOf([
            OpticalDistortion(p=0.3),
            GridDistortion(p=0.1),
            IAAPiecewiseAffine(p=0.3),
```

(continues on next page)

(continued from previous page)

```
        ], p=0.2),
        OneOf([
            CLAHE(clip_limit=2),
            IAASharpen(),
            IAAEmboss(),
            RandomBrightnessContrast(),
        ], p=0.3),
        HueSaturationValue(p=0.3),
    ], p=p)

image = np.ones((300, 300, 3), dtype=np.uint8)
mask = np.ones((300, 300), dtype=np.uint8)
whatever_data = "my name"
augmentation = strong_aug(p=0.9)
data = {"image": image, "mask": mask, "whatever_data": whatever_data, "additional":
↪"hello"}
augmented = augmentation(**data)
image, mask, whatever_data, additional = augmented["image"], augmented["mask"],␣
↪augmented["whatever_data"], augmented["additional"]
```

For more examples see example.ipynb and example_16_bit_tiff.ipynb

4.2 Contributing

All development is done on GitHub: https://github.com/albu/albumentations

If you find a bug or have a feature request file an issue at https://github.com/albu/albumentations/issues

4.3 API

4.3.1 Core API (albumentations.core)

Composition

class `albumentations.core.composition.`**`Compose`**(*transforms*, *bbox_params=None*, *keypoint_params=None*, *additional_targets=None*, *p=1.0*)

Compose transforms and handle all transformations regrading bounding boxes

Parameters

- **transforms** (*list*) – list of transformations to compose.
- **bbox_params** (*dict*) – Parameters for bounding boxes transforms
- **keypoint_params** (*dict*) – Parameters for keypoints transforms
- **additional_targets** (*dict*) – Dict with keys - new target name, values - old target name. ex: {'image2': 'image'}
- **p** (*float*) – probability of applying all list of transforms. Default: 1.0.

bbox_params dictionary contains the following keys:

- **format** (*str*): format of bounding boxes. Should be 'coco', 'pascal_voc' or 'albumentations'. If None - don't use bboxes. The *coco* format of a bounding box looks like *[x_min, y_min, width, height]*, e.g. [97, 12, 150, 200]. The *pascal_voc* format of a bounding box looks like *[x_min, y_min, x_max, y_max]*, e.g. [97, 12, 247, 212]. The *albumentations* format of a bounding box looks like *pascal_voc*, but between [0, 1], in other words: [x_min, y_min, x_max, y_max]', e.g. [0.2, 0.3, 0.4, 0.5].
- **label_fields** (*list*): list of fields that are joined with boxes, e.g labels.
  Should be same type as boxes.
- **min_area** (*float*): minimum area of a bounding box. All bounding boxes whose
  visible area in pixels is less than this value will be removed. Default: 0.0.
- **min_visibility** (*float*): minimum fraction of area for a bounding box
  to remain this box in list. Default: 0.0.

class `albumentations.core.composition.`**`OneOf`**(*transforms*, *p=0.5*)
Select on of transforms to apply

Parameters

- **`transforms`** (*`list`*) – list of transformations to compose.
- **`p`** (*`float`*) – probability of applying selected transform. Default: 0.5.

Transforms interface

`albumentations.core.transforms_interface.`**`to_tuple`**(*param*, *low=None*, *bias=None*)
Convert input argument to min-max tuple :param param: Input value.

If value is scalar, return value would be (offset - value, offset + value). If value is tuple, return value would be value + offset (broadcasted).

Parameters

- **`low`** – Second element of tuple can be passed as optional argument
- **`bias`** – An offset factor added to each element

class `albumentations.core.transforms_interface.`**`DualTransform`**(*always_apply=False*, *p=0.5*)
Transform for segmentation task.

class `albumentations.core.transforms_interface.`**`ImageOnlyTransform`**(*always_apply=False*, *p=0.5*)
Transform applied to image only.

class `albumentations.core.transforms_interface.`**`NoOp`**(*always_apply=False*, *p=0.5*)
Does nothing

Serialization

`albumentations.core.serialization.`**`to_dict`**(*transform*, *on_not_implemented_error='raise'*)
Take a transform pipeline and convert it to a serializable representation that uses only standard python data types: dictionaries, lists, strings, integers, and floats.

Parameters **`transform`** (*`object`*) – A transform that should be serialized. If the transform doesn't implement the *to_dict* method and *on_not_implemented_error* equals to 'raise' then

NotImplementedError is raised. If *on_not_implemented_error* equals to 'warn' then *NotImplementedError* will be ignored but no transform parameters will be serialized.

`albumentations.core.serialization.`**`from_dict`**(*transform_dict*, *lambda_transforms=None*)

Parameters

- **`transform`** (*`dict`*) – A dictionary with serialized transform pipeline.
- **`lambda_transforms`** (*`dict`*) – A dictionary that contains lambda transforms, that is instances of the Lambda class. This dictionary is required when you are restoring a pipeline that contains lambda transforms. Keys in that dictionary should be named same as *name* arguments in respective lambda transforms from a serialized pipeline.

`albumentations.core.serialization.`**`save`**(*transform*, *filepath*, *data_format='json'*, *on_not_implemented_error='raise'*)

Take a transform pipeline, serialize it and save a serialized version to a file using either json or yaml format.

Parameters

- **`transform`** (*`obj`*) – Transform to serialize.
- **`filepath`** (*`str`*) – Filepath to write to.
- **`data_format`** (*`str`*) – Serialization format. Should be either *json* or 'yaml'.
- **`on_not_implemented_error`** (*`str`*) – Parameter that describes what to do if a transform doesn't implement the *to_dict* method. If 'raise' then *NotImplementedError* is raised, if *warn* then the exception will be ignored and no transform arguments will be saved.

`albumentations.core.serialization.`**`load`**(*filepath*, *data_format='json'*, *lambda_transforms=None*)

Load a serialized pipeline from a json or yaml file and construct a transform pipeline.

Parameters

- **`transform`** (*`obj`*) – Transform to serialize.
- **`filepath`** (*`str`*) – Filepath to read from.
- **`data_format`** (*`str`*) – Serialization format. Should be either *json* or 'yaml'.
- **`lambda_transforms`** (*`dict`*) – A dictionary that contains lambda transforms, that is instances of the Lambda class. This dictionary is required when you are restoring a pipeline that contains lambda transforms. Keys in that dictionary should be named same as *name* arguments in respective lambda transforms from a serialized pipeline.

4.3.2 Augmentations (albumentations.augmentations)

Transforms

`class` `albumentations.augmentations.transforms.`**`Blur`**(*blur_limit=7*, *always_apply=False*, *p=0.5*)

Blur the input image using a random-sized kernel.

Parameters

- **`blur_limit`** (*`int`*) – maximum kernel size for blurring the input image. Default: 7.
- **`p`** (*`float`*) – probability of applying the transform. Default: 0.5.

Targets: image

Image types: uint8, float32

class albumentations.augmentations.transforms.**VerticalFlip**(*always_apply=False*, *p=0.5*)

Flip the input vertically around the x-axis.

Parameters **p** (*float*) – probability of applying the transform. Default: 0.5.

Targets: image, mask, bboxes, keypoints

Image types: uint8, float32

class albumentations.augmentations.transforms.**HorizontalFlip**(*always_apply=False*, *p=0.5*)

Flip the input horizontally around the y-axis.

Parameters **p** (*float*) – probability of applying the transform. Default: 0.5.

Targets: image, mask, bboxes, keypoints

Image types: uint8, float32

class albumentations.augmentations.transforms.**Flip**(*always_apply=False*, *p=0.5*)

Flip the input either horizontally, vertically or both horizontally and vertically.

Parameters **p** (*float*) – probability of applying the transform. Default: 0.5.

Targets: image, mask, bboxes, keypoints

Image types: uint8, float32

apply(*img*, *d=0*, ***params*)

Args: d (int): code that specifies how to flip the input. 0 for vertical flipping, 1 for horizontal flipping,

-1 for both vertical and horizontal flipping (which is also could be seen as rotating the input by 180 degrees).

class albumentations.augmentations.transforms.**Normalize**(*mean=(0.485, 0.456, 0.406)*, *std=(0.229, 0.224, 0.225)*, *max_pixel_value=255.0*, *always_apply=False*, *p=1.0*)

Divide pixel values by 255 = 2**8 - 1, subtract mean per channel and divide by std per channel.

Parameters

- **mean** (*float, float, float*) – mean values
- **std** (*float, float, float*) – std values
- **max_pixel_value** (*float*) – maximum possible pixel value

Targets: image

Image types: uint8, float32

class albumentations.augmentations.transforms.**Transpose**(*always_apply=False*, *p=0.5*)

Transpose the input by swapping rows and columns.

Parameters **p** (*float*) – probability of applying the transform. Default: 0.5.

Targets: image, mask, bboxes

Image types: uint8, float32

class albumentations.augmentations.transforms.**RandomCrop**(*height*, *width*, *always_apply=False*, *p=1.0*)

Crop a random part of the input.

Parameters

- **height** (*int*) – height of the crop.
- **width** (*int*) – width of the crop.
- **p** (*float*) – probability of applying the transform. Default: 1.

Targets: image, mask, bboxes, keypoints

Image types: uint8, float32

class albumentations.augmentations.transforms.**RandomGamma**(*gamma_limit=(80, 120)*, *always_apply=False*, *p=0.5*)

Targets: image

Image types: uint8, float32

class albumentations.augmentations.transforms.**RandomRotate90**(*always_apply=False*, *p=0.5*)

Randomly rotate the input by 90 degrees zero or more times.

Parameters **p** (*float*) – probability of applying the transform. Default: 0.5.

Targets: image, mask, bboxes, keypoints

Image types: uint8, float32

apply(*img*, *factor=0*, ***params*)

Parameters **factor** (*int*) – number of times the input will be rotated by 90 degrees.

class albumentations.augmentations.transforms.**Rotate**(*limit=90*, *interpolation=1*, *border_mode=4*, *value=None*, *always_apply=False*, *p=0.5*)

Rotate the input by an angle selected randomly from the uniform distribution.

Parameters

- **limit** (*(int, int) or int*) – range from which a random angle is picked. If limit is a single int an angle is picked from (-limit, limit). Default: 90
- **interpolation** (*OpenCV flag*) – flag that is used to specify the interpolation algorithm. Should be one of: cv2.INTER_NEAREST, cv2.INTER_LINEAR, cv2.INTER_CUBIC, cv2.INTER_AREA, cv2.INTER_LANCZOS4. Default: cv2.INTER_LINEAR.
- **border_mode** (*OpenCV flag*) – flag that is used to specify the pixel extrapolation method. Should be one of: cv2.BORDER_CONSTANT, cv2.BORDER_REPLICATE, cv2.BORDER_REFLECT, cv2.BORDER_WRAP, cv2.BORDER_REFLECT_101. Default: cv2.BORDER_REFLECT_101

- **value** (*list of ints [r, g, b]*) – padding value if border_mode is cv2.BORDER_CONSTANT.
- **p** (*float*) – probability of applying the transform. Default: 0.5.

Targets: image, mask, bboxes, keypoints

Image types: uint8, float32

class albumentations.augmentations.transforms.**ShiftScaleRotate**(*shift_limit=0.0625, scale_limit=0.1, rotate_limit=45, interpolation=1, border_mode=4, value=None, always_apply=False, p=0.5*)

Randomly apply affine transforms: translate, scale and rotate the input.

Parameters

- **shift_limit** (*(float, float) or float*) – shift factor range for both height and width. If shift_limit is a single float value, the range will be (-shift_limit, shift_limit). Absolute values for lower and upper bounds should lie in range [0, 1]. Default: 0.0625.
- **scale_limit** (*(float, float) or float*) – scaling factor range. If scale_limit is a single float value, the range will be (-scale_limit, scale_limit). Default: 0.1.
- **rotate_limit** (*(int, int) or int*) – rotation range. If rotate_limit is a single int value, the range will be (-rotate_limit, rotate_limit). Default: 45.
- **interpolation** (*OpenCV flag*) – flag that is used to specify the interpolation algorithm. Should be one of: cv2.INTER_NEAREST, cv2.INTER_LINEAR, cv2.INTER_CUBIC, cv2.INTER_AREA, cv2.INTER_LANCZOS4. Default: cv2.INTER_LINEAR.
- **border_mode** (*OpenCV flag*) – flag that is used to specify the pixel extrapolation method. Should be one of: cv2.BORDER_CONSTANT, cv2.BORDER_REPLICATE, cv2.BORDER_REFLECT, cv2.BORDER_WRAP, cv2.BORDER_REFLECT_101. Default: cv2.BORDER_REFLECT_101
- **value** (*list of ints [r, g, b]*) – padding value if border_mode is cv2.BORDER_CONSTANT.
- **p** (*float*) – probability of applying the transform. Default: 0.5.

Targets: image, mask, keypoints

Image types: uint8, float32

class albumentations.augmentations.transforms.**CenterCrop**(*height, width, always_apply=False, p=1.0*)

Crop the central part of the input.

Parameters

- **height** (*int*) – height of the crop.
- **width** (*int*) – width of the crop.

- **p** (*float*) – probability of applying the transform. Default: 1.

Targets: image, mask, bboxes, keypoints

Image types: uint8, float32

Note: It is recommended to use uint8 images as input. Otherwise the operation will require internal conversion float32 -> uint8 -> float32 that causes worse performance.

class albumentations.augmentations.transforms.**OpticalDistortion**(*distort_limit=0.05, shift_limit=0.05, interpolation=1, border_mode=4, value=None, always_apply=False, p=0.5*)

Targets: image, mask

Image types: uint8, float32

class albumentations.augmentations.transforms.**GridDistortion**(*num_steps=5, distort_limit=0.3, interpolation=1, border_mode=4, value=None, always_apply=False, p=0.5*)

Targets: image, mask

Image types: uint8, float32

class albumentations.augmentations.transforms.**ElasticTransform**(*alpha=1, sigma=50, alpha_affine=50, interpolation=1, border_mode=4, value=None, always_apply=False, approximate=False, p=0.5*)

Elastic deformation of images as described in [Simard2003] (with modifications). Based on https://gist.github.com/erniejunior/601cdf56d2b424757de5

Parameters **approximate** (*boolean*) – Whether to smooth displacement map with fixed kernel size. Enabling this option gives ~2X speedup on large images.

Targets: image, mask

Image types: uint8, float32

class albumentations.augmentations.transforms.**HueSaturationValue**(*hue_shift_limit=20*, *sat_shift_limit=30*, *val_shift_limit=20*, *always_apply=False*, *p=0.5*)

Randomly change hue, saturation and value of the input image.

Parameters

- **hue_shift_limit** (*(int, int) or int*) – range for changing hue. If hue_shift_limit is a single int, the range will be (-hue_shift_limit, hue_shift_limit). Default: 20.
- **sat_shift_limit** (*(int, int) or int*) – range for changing saturation. If sat_shift_limit is a single int, the range will be (-sat_shift_limit, sat_shift_limit). Default: 30.
- **val_shift_limit** (*(int, int) or int*) – range for changing value. If val_shift_limit is a single int, the range will be (-val_shift_limit, val_shift_limit). Default: 20.
- **p** (*float*) – probability of applying the transform. Default: 0.5.

Targets: image

Image types: uint8, float32

class albumentations.augmentations.transforms.**PadIfNeeded**(*min_height=1024*, *min_width=1024*, *border_mode=4*, *value=None*, *always_apply=False*, *p=1.0*)

Pad side of the image / max if side is less than desired number.

Parameters

- **p** (*float*) – probability of applying the transform. Default: 1.0.
- **value** (*list of ints [r, g, b]*) – padding value if border_mode is cv2.BORDER_CONSTANT.

Targets: image, mask, bbox, keypoints

Image types: uint8, float32

class albumentations.augmentations.transforms.**RGBShift**(*r_shift_limit=20*, *g_shift_limit=20*, *b_shift_limit=20*, *always_apply=False*, *p=0.5*)

Randomly shift values for each channel of the input RGB image.

Parameters

- **r_shift_limit** (*(int, int) or int*) – range for changing values for the red channel. If r_shift_limit is a single int, the range will be (-r_shift_limit, r_shift_limit). Default: 20.

- **g_shift_limit** (*(int, int) or int*) – range for changing values for the green channel. If g_shift_limit is a single int, the range will be (-g_shift_limit, g_shift_limit). Default: 20.
- **b_shift_limit** (*(int, int) or int*) – range for changing values for the blue channel. If b_shift_limit is a single int, the range will be (-b_shift_limit, b_shift_limit). Default: 20.
- **p** (*float*) – probability of applying the transform. Default: 0.5.

Targets: image

Image types: uint8, float32

class albumentations.augmentations.transforms.**RandomBrightness**(*limit=0.2*, *always_apply=False*, *p=0.5*)

class albumentations.augmentations.transforms.**RandomContrast**(*limit=0.2*, *always_apply=False*, *p=0.5*)

class albumentations.augmentations.transforms.**MotionBlur**(*blur_limit=7*, *always_apply=False*, *p=0.5*)

Apply motion blur to the input image using a random-sized kernel.

Parameters

- **blur_limit** (*int*) – maximum kernel size for blurring the input image. Default: 7.
- **p** (*float*) – probability of applying the transform. Default: 0.5.

Targets: image

Image types: uint8, float32

class albumentations.augmentations.transforms.**MedianBlur**(*blur_limit=7*, *always_apply=False*, *p=0.5*)

Blur the input image using using a median filter with a random aperture linear size.

Parameters

- **blur_limit** (*int*) – maximum aperture linear size for blurring the input image. Default: 7.
- **p** (*float*) – probability of applying the transform. Default: 0.5.

Targets: image

Image types: uint8, float32

class albumentations.augmentations.transforms.**GaussianBlur**(*blur_limit=7*, *always_apply=False*, *p=0.5*)

Blur the input image using using a Gaussian filter with a random kernel size.

Parameters

- **blur_limit** (*int*) – maximum Gaussian kernel size for blurring the input image. Default: 7.

- **p** (*float*) – probability of applying the transform. Default: 0.5.

Targets: image

Image types: uint8, float32

class albumentations.augmentations.transforms.**GaussNoise**(*var_limit=(10.0, 50.0)*, *always_apply=False*, *p=0.5*)

Apply gaussian noise to the input image.

Parameters

- **var_limit** (*(float, float) or float*) – variance range for noise. If var_limit is a single float, the range will be (-var_limit, var_limit). Default: (10., 50.).
- **p** (*float*) – probability of applying the transform. Default: 0.5.

Targets: image

Image types: uint8

class albumentations.augmentations.transforms.**CLAHE**(*clip_limit=4.0*, *tile_grid_size=(8, 8)*, *always_apply=False*, *p=0.5*)

Apply Contrast Limited Adaptive Histogram Equalization to the input image.

Parameters

- **clip_limit** (*float*) – upper threshold value for contrast limiting. Default: 4.0. tile_grid_size ((int, int)): size of grid for histogram equalization. Default: (8, 8).
- **p** (*float*) – probability of applying the transform. Default: 0.5.

Targets: image

Image types: uint8

class albumentations.augmentations.transforms.**ChannelShuffle**(*always_apply=False*, *p=0.5*)

Randomly rearrange channels of the input RGB image.

Parameters **p** (*float*) – probability of applying the transform. Default: 0.5.

Targets: image

Image types: uint8, float32

class albumentations.augmentations.transforms.**InvertImg**(*always_apply=False*, *p=0.5*)

Invert the input image by subtracting pixel values from 255.

Parameters **p** (*float*) – probability of applying the transform. Default: 0.5.

Targets: image

Image types: uint8

class albumentations.augmentations.transforms.**ToGray**(*always_apply=False*, *p=0.5*)

Convert the input RGB image to grayscale. If the mean pixel value for the resulting image is greater than 127, invert the resulting grayscale image.

Parameters **p** (*float*) – probability of applying the transform. Default: 0.5.

Targets: image

Image types: uint8, float32

class albumentations.augmentations.transforms.**JpegCompression**(*quality_lower=99, quality_upper=100, always_apply=False, p=0.5*)

Decrease Jpeg compression of an image.

Parameters

- **quality_lower** (*float*) – lower bound on the jpeg quality. Should be in [0, 100] range
- **quality_upper** (*float*) – upper bound on the jpeg quality. Should be in [0, 100] range

Targets: image

Image types: uint8, float32

class albumentations.augmentations.transforms.**Cutout**(*num_holes=8, max_h_size=8, max_w_size=8, fill_value=0, always_apply=False, p=0.5*)

CoarseDropout of the square regions in the image. :param num_holes: number of regions to zero out :type num_holes: int :param max_h_size: maximum height of the hole :type max_h_size: int :param max_w_size: maximum width of the hole :type max_w_size: int

Targets: image

Image types: uint8, float32

Reference: | https://arxiv.org/abs/1708.04552 | https://github.com/uoguelph-mlrg/Cutout/blob/master/util/cutout.py | https://github.com/aleju/imgaug/blob/master/imgaug/augmenters/arithmetic.py

class albumentations.augmentations.transforms.**CoarseDropout**(*max_holes=8, max_height=8, max_width=8, min_holes=None, min_height=None, min_width=None, fill_value=0, always_apply=False, p=0.5*)

CoarseDropout of the rectangular regions in the image.

Parameters

- **max_holes** (*int*) – Maximum number of regions to zero out.
- **max_height** (*int*) – Maximum height of the hole.
- **min_width** (*int*) – Maximum width of the hole.
- **min_holes** (*int*) – Minimum number of regions to zero out. If *None*, *min_holes* is be set to *max_holes*. Default: *None*.
- **min_height** (*int*) – Minimum height of the hole. Default: None. If *None*, *min_height* is set to *max_height*. Default: *None*.

- **min_width** – Minimum width of the hole. If *None*, *min_height* is set to *max_width*. Default: *None*.

Targets: image

Image types: uint8, float32

Reference: | https://arxiv.org/abs/1708.04552 | https://github.com/uoguelph-mlrg/Cutout/blob/master/util/cutout.py | https://github.com/aleju/imgaug/blob/master/imgaug/augmenters/arithmetic.py

class `albumentations.augmentations.transforms.`**`ToFloat`**(*max_value=None*, *always_apply=False*, *p=1.0*)

Divide pixel values by *max_value* to get a float32 output array where all values lie in the range [0, 1.0]. If *max_value* is None the transform will try to infer the maximum value by inspecting the data type of the input image.

See also:

`FromFloat`

Parameters

- **max_value** (*float*) – maximum possible input value. Default: None.
- **p** (*float*) – probability of applying the transform. Default: 1.0.

Targets: image

Image types: any type

class `albumentations.augmentations.transforms.`**`FromFloat`**(*dtype='uint16'*, *max_value=None*, *always_apply=False*, *p=1.0*)

Take an input array where all values should lie in the range [0, 1.0], multiply them by *max_value* and then cast the resulted value to a type specified by *dtype*. If *max_value* is None the transform will try to infer the maximum value for the data type from the *dtype* argument.

This is the inverse transform for *`ToFloat`*.

Parameters

- **max_value** (*float*) – maximum possible input value. Default: None.
- **dtype** (*string or numpy data type*) – data type of the output. See the 'Data types' page from the NumPy docs. Default: 'uint16'.
- **p** (*float*) – probability of applying the transform. Default: 1.0.

Targets: image

Image types: float32

class `albumentations.augmentations.transforms.`**`Crop`**(*x_min=0*, *y_min=0*, *x_max=1024*, *y_max=1024*, *always_apply=False*, *p=1.0*)

Crop region from image.

Parameters

- **x_min** (*int*) – minimum upper left x coordinate

- **y_min** (*int*) – minimum upper left y coordinate
- **x_max** (*int*) – maximum lower right x coordinate
- **y_max** (*int*) – maximum lower right y coordinate

Targets: image, mask, bboxes

Image types: uint8, float32

class albumentations.augmentations.transforms.**RandomScale**(*scale_limit=0.1*, *interpolation=1*, *always_apply=False*, *p=0.5*)

Randomly resize the input. Output image size is different from the input image size.

Parameters

- **scale_limit** (*(float, float) or float*) – scaling factor range. If scale_limit is a single float value, the range will be (1 - scale_limit, 1 + scale_limit). Default: 0.1.
- **interpolation** (*OpenCV flag*) – flag that is used to specify the interpolation algorithm. Should be one of: cv2.INTER_NEAREST, cv2.INTER_LINEAR, cv2.INTER_CUBIC, cv2.INTER_AREA, cv2.INTER_LANCZOS4. Default: cv2.INTER_LINEAR.
- **p** (*float*) – probability of applying the transform. Default: 0.5.

Targets: image, mask, bboxes, keypoints

Image types: uint8, float32

class albumentations.augmentations.transforms.**LongestMaxSize**(*max_size=1024*, *interpolation=1*, *always_apply=False*, *p=1*)

Rescale an image so that maximum side is equal to max_size, keeping the aspect ratio of the initial image.

Parameters

- **p** (*float*) – probability of applying the transform. Default: 1.
- **max_size** (*int*) – maximum size of the image after the transformation

Targets: image, mask, bboxes

Image types: uint8, float32

class albumentations.augmentations.transforms.**SmallestMaxSize**(*max_size=1024*, *interpolation=1*, *always_apply=False*, *p=1*)

Rescale an image so that minimum side is equal to max_size, keeping the aspect ratio of the initial image.

Parameters

- **p** (*float*) – probability of applying the transform. Default: 1.
- **max_size** (*int*) – maximum size of smallest side of the image after the transformation

Targets: image, mask, bboxes

Image types: uint8, float32

class albumentations.augmentations.transforms.**Resize**(*height*, *width*, *interpolation=1*, *always_apply=False*, *p=1*)

Resize the input to the given height and width.

Parameters

- **p** (*float*) – probability of applying the transform. Default: 1.
- **height** (*int*) – desired height of the output.
- **width** (*int*) – desired width of the output.
- **interpolation** (*OpenCV flag*) – flag that is used to specify the interpolation algorithm. Should be one of: cv2.INTER_NEAREST, cv2.INTER_LINEAR, cv2.INTER_CUBIC, cv2.INTER_AREA, cv2.INTER_LANCZOS4. Default: cv2.INTER_LINEAR.

Targets: image, mask, bboxes

Image types: uint8, float32

class albumentations.augmentations.transforms.**RandomSizedCrop**(*min_max_height*, *height*, *width*, *w2h_ratio=1.0*, *interpolation=1*, *always_apply=False*, *p=1.0*)

Crop a random part of the input and rescale it to some size.

Parameters

- **min_max_height** (*(int, int)*) – crop size limits.
- **height** (*int*) – height after crop and resize.
- **width** (*int*) – width after crop and resize.
- **w2h_ratio** (*float*) – aspect ratio of crop.
- **interpolation** (*OpenCV flag*) – flag that is used to specify the interpolation algorithm. Should be one of: cv2.INTER_NEAREST, cv2.INTER_LINEAR, cv2.INTER_CUBIC, cv2.INTER_AREA, cv2.INTER_LANCZOS4. Default: cv2.INTER_LINEAR.
- **p** (*float*) – probability of applying the transform. Default: 1.

Targets: image, mask, bboxes, keypoints

Image types: uint8, float32

class albumentations.augmentations.transforms.**RandomBrightnessContrast**(*brightness_limit=0.2*, *contrast_limit=0.2*, *always_apply=False*, *p=0.5*)

Randomly change brightness and contrast of the input image.

Parameters

- **brightness_limit** (*(float, float) or float*) – factor range for changing brightness. If limit is a single float, the range will be (-limit, limit). Default: 0.2.
- **contrast_limit** (*(float, float) or float*) – factor range for changing contrast. If limit is a single float, the range will be (-limit, limit). Default: 0.2.
- **p** (*float*) – probability of applying the transform. Default: 0.5.

Targets: image

Image types: uint8, float32

class albumentations.augmentations.transforms.**RandomCropNearBBox**(*max_part_shift=0.3*, *always_apply=False*, *p=1.0*)

Crop bbox from image with random shift by x,y coordinates

Parameters

- **max_part_shift** (*float*) – float value in (0.0, 1.0) range. Default 0.3
- **p** (*float*) – probability of applying the transform. Default: 1.

Targets: image

Image types: uint8, float32

class albumentations.augmentations.transforms.**RandomSizedBBoxSafeCrop**(*height*, *width*, *erosion_rate=0.0*, *interpolation=1*, *always_apply=False*, *p=1.0*)

Crop a random part of the input and rescale it to some size without loss of bboxes.

Parameters

- **height** (*int*) – height after crop and resize.
- **width** (*int*) – width after crop and resize.
- **erosion_rate** (*float*) – erosion rate applied on input image height before crop.
- **interpolation** (*OpenCV flag*) – flag that is used to specify the interpolation algorithm. Should be one of: cv2.INTER_NEAREST, cv2.INTER_LINEAR, cv2.INTER_CUBIC, cv2.INTER_AREA, cv2.INTER_LANCZOS4. Default: cv2.INTER_LINEAR.
- **p** (*float*) – probability of applying the transform. Default: 1.

Targets: image, mask, bboxes

Image types: uint8, float32

class albumentations.augmentations.transforms.**RandomSnow**(*snow_point_lower=0.1*, *snow_point_upper=0.3*, *brightness_coeff=2.5*, *always_apply=False*, *p=0.5*)

Bleach out some pixel values simulating snow.

From https://github.com/UjjwalSaxena/Automold–Road-Augmentation-Library

Parameters

- **snow_point_lower** (*float*) – lower_bond of the amount of snow. Should be in [0, 1] range
- **snow_point_upper** (*float*) – upper_bond of the amount of snow. Should be in [0, 1] range
- **brightness_coeff** (*float*) – larger number will lead to a more snow on the image. Should be >= 0

Targets: image

Image types: uint8, float32

class albumentations.augmentations.transforms.**RandomRain**(*slant_lower=-10*, *slant_upper=10*, *drop_length=20*, *drop_width=1*, *drop_color=(200, 200, 200)*, *blur_value=7*, *brightness_coefficient=0.7*, *rain_type=None*, *always_apply=False*, *p=0.5*)

Adds rain effects.

From https://github.com/UjjwalSaxena/Automold–Road-Augmentation-Library

Parameters

- **slant_lower** –
- **slant_upper** –
- **drop_length** –
- **drop_width** –
- **drop_color** –
- **blur_value** (*int*) – rainy view are blurry
- **brightness_coefficient** (*float*) – rainy days are usually shady
- **rain_type** – [None, "drizzle", "heavy", "torrestial"]

Targets: image

Image types: uint8, float32

class albumentations.augmentations.transforms.**RandomFog**(*fog_coef_lower=0.3*, *fog_coef_upper=1*, *alpha_coef=0.08*, *always_apply=False*, *p=0.5*)

Simulates fog for the image

From https://github.com/UjjwalSaxena/Automold–Road-Augmentation-Library

Parameters

- **fog_coef_lower** (*float*) – lower limit for fog intensity coefficient. Should be in [0, 1] range.
- **fog_coef_upper** (*float*) – upper limit for fog intensity coefficient. Should be in [0, 1] range.
- **alpha_coef** (*float*) – transparence of the fog circles. Should be in [0, 1] range.

Targets: image

Image types: uint8, float32

class albumentations.augmentations.transforms.**RandomSunFlare**(*flare_roi=(0, 0, 1, 0.5)*, *angle_lower=0*, *angle_upper=1*, *num_flare_circles_lower=6*, *num_flare_circles_upper=10*, *src_radius=400*, *src_color=(255, 255, 255)*, *always_apply=False*, *p=0.5*)

Simulates Sun Flare for the image

From https://github.com/UjjwalSaxena/Automold–Road-Augmentation-Library

Parameters

- **flare_roi** (*float, float, float, float*) – region of the image where flare will appear (x_min, y_min, x_max, y_max)
- **angle_lower** (*float*) –
- **angle_upper** (*float*) –
- **num_flare_circles_lower** (*int*) – lower limit for the number of flare circles.
- **num_flare_circles_upper** (*int*) – upper limit for the number of flare circles.
- **src_radius** (*int*) –
- **src_color** (*int, int, int*) – color of the flare

Targets: image

Image types: uint8, float32

class albumentations.augmentations.transforms.**RandomShadow**(*shadow_roi=(0, 0.5, 1, 1), num_shadows_lower=1, num_shadows_upper=2, shadow_dimension=5, always_apply=False, p=0.5*)

Simulates shadows for the image

From https://github.com/UjjwalSaxena/Automold–Road-Augmentation-Library

Parameters

- **shadow_roi** (*float, float, float, float*) – region of the image where shadows will appear (x_min, y_min, x_max, y_max)
- **num_shadows_lower** (*int*) – Lower limit for the possible number of shadows.
- **num_shadows_upper** (*int*) – Lower limit for the possible number of shadows.
- **shadow_dimension** (*int*) – number of edges in the shadow polygons

Targets: image

Image types: uint8, float32

class albumentations.augmentations.transforms.**Lambda**(*image=None, mask=None, keypoint=None, bbox=None, name=None, always_apply=False, p=1.0*)

A flexible transformation class for using user-defined transformation functions per targets. Function signature must include **kwargs to accept optinal arguments like interpolation method, image size, etc:

Parameters

- **image** (*callable*) – Image transformation function.
- **mask** (*callable*) – Mask transformation function.
- **keypoint** (*callable*) – Keypoint transformation function.
- **bbox** (*callable*) – BBox transformation function.
- **always_apply** (*bool*) – Indicates whether this transformation should be always applied.
- **p** (*float*) – probability of applying the transform. Default: 1.0.

Targets: image, mask, bboxes, keypoints

Image types: Any

class albumentations.augmentations.transforms.**ChannelDropout**(*channel_drop_range=(1, 1), fill_value=0, always_apply=False, p=0.5*)

Randomly Drop Channels in the input Image.

Parameters

- **channel_drop_range** (*int, int*) – range from which we choose the number of channels to drop.
- **fill_value** – pixel value for the dropped channel.

- **p** (*float*) – probability of applying the transform. Default: 0.5.

Targets: image

Image types: uint8, uint16, unit32, float32

Functional transforms

albumentations.augmentations.functional.**add_fog**(*img*, *fog_coef*, *alpha_coef*, *haze_list*)

Add fog to the image.

From https://github.com/UjjwalSaxena/Automold–Road-Augmentation-Library

Parameters

- **img** (*np.array*) –
- **fog_coef** (*float*) –
- **alpha_coef** (*float*) –
- **haze_list** (*list*) –

Returns:

albumentations.augmentations.functional.**add_rain**(*img*, *slant*, *drop_length*, *drop_width*, *drop_color*, *blur_value*, *brightness_coefficient*, *rain_drops*)

From https://github.com/UjjwalSaxena/Automold–Road-Augmentation-Library

Parameters

- **img** (*np.uint8*) –
- **slant** (*int*) –
- **drop_length** –
- **drop_width** –
- **drop_color** –
- **blur_value** (*int*) – rainy view are blurry
- **brightness_coefficient** (*float*) – rainy days are usually shady
- **rain_drops** –

Returns:

albumentations.augmentations.functional.**add_shadow**(*img*, *vertices_list*)

Add shadows to the image.

From https://github.com/UjjwalSaxena/Automold–Road-Augmentation-Library

Parameters

- **img** (*np.array*) –
- **vertices_list** (*list*) –

Returns:

albumentations.augmentations.functional.**add_snow**(*img*, *snow_point*, *brightness_coeff*)

Bleaches out pixels, mitation snow.

From https://github.com/UjjwalSaxena/Automold–Road-Augmentation-Library

Parameters

- **img** –
- **snow_point** –
- **brightness_coeff** –

Returns:

albumentations.augmentations.functional.**add_sun_flare**(*img*, *flare_center_x*, *flare_center_y*, *src_radius*, *src_color*, *circles*)

Add sun flare.

From https://github.com/UjjwalSaxena/Automold–Road-Augmentation-Library

Parameters

- **img** (*np.array*) –
- **flare_center_x** (*float*) –
- **flare_center_y** (*float*) –
- **src_radius** –
- **src_color** (*int, int, int*) –
- **circles** (*list*) –

Returns:

albumentations.augmentations.functional.**bbox_flip**(*bbox*, *d*, *rows*, *cols*)

Flip a bounding box either vertically, horizontally or both depending on the value of *d*.

Raises ValueError – if value of *d* is not -1, 0 or 1.

albumentations.augmentations.functional.**bbox_hflip**(*bbox*, *rows*, *cols*)

Flip a bounding box horizontally around the y-axis.

albumentations.augmentations.functional.**bbox_rot90**(*bbox*, *factor*, *rows*, *cols*)

Rotates a bounding box by 90 degrees CCW (see np.rot90)

Parameters

- **bbox** (*tuple*) – A tuple (x_min, y_min, x_max, y_max).
- **factor** (*int*) – Number of CCW rotations. Must be in range [0;3] See np.rot90.
- **rows** (*int*) – Image rows.
- **cols** (*int*) – Image cols.

albumentations.augmentations.functional.**bbox_rotate**(*bbox*, *angle*, *rows*, *cols*, *interpolation*)

Rotates a bounding box by angle degrees

Parameters

- **bbox** (*tuple*) – A tuple (x_min, y_min, x_max, y_max).
- **angle** (*int*) – Angle of rotation in degrees
- **rows** (*int*) – Image rows.
- **cols** (*int*) – Image cols.
- **interpolation** (*int*) – interpolation method.

- **a tuple** (*return*) –

`albumentations.augmentations.functional.`**`bbox_transpose`**(*bbox*, *axis*, *rows*, *cols*)

Transposes a bounding box along given axis.

Parameters

- **bbox** (*tuple*) – A tuple (x_min, y_min, x_max, y_max).
- **axis** (*int*) – 0 - main axis, 1 - secondary axis.
- **rows** (*int*) – Image rows.
- **cols** (*int*) – Image cols.

`albumentations.augmentations.functional.`**`bbox_vflip`**(*bbox*, *rows*, *cols*)

Flip a bounding box vertically around the x-axis.

`albumentations.augmentations.functional.`**`crop_bbox_by_coords`**(*bbox*, *crop_coords*, *crop_height*, *crop_width*, *rows*, *cols*)

Crop a bounding box using the provided coordinates of bottom-left and top-right corners in pixels and the required height and width of the crop.

`albumentations.augmentations.functional.`**`crop_keypoint_by_coords`**(*keypoint*, *crop_coords*, *crop_height*, *crop_width*, *rows*, *cols*)

Crop a keypoint using the provided coordinates of bottom-left and top-right corners in pixels and the required height and width of the crop.

`albumentations.augmentations.functional.`**`elastic_transform`**(*image*, *alpha*, *sigma*, *alpha_affine*, *interpolation=1*, *border_mode=4*, *value=None*, *random_state=None*, *approximate=False*)

Elastic deformation of images as described in [Simard2003] (with modifications). Based on https://gist.github.com/erniejunior/601cdf56d2b424757de5

`albumentations.augmentations.functional.`**`elastic_transform_approx`**(*image*, *alpha*, *sigma*, *alpha_affine*, *interpolation=1*, *border_mode=4*, *random_state=None*)

Elastic deformation of images as described in [Simard2003] (with modifications for speed). Based on https://gist.github.com/erniejunior/601cdf56d2b424757de5

`albumentations.augmentations.functional.`**`grid_distortion`**(*img*, *num_steps=10*, *xsteps=[]*, *ysteps=[]*, *interpolation=1*, *border_mode=4*, *value=None*)

Reference: http://pythology.blogspot.sg/2014/03/interpolation-on-regular-distorted-grid.html

albumentations.augmentations.functional.**keypoint_flip**(*bbox*, *d*, *rows*, *cols*)
Flip a keypoint either vertically, horizontally or both depending on the value of *d*.

Raises `ValueError` – if value of *d* is not -1, 0 or 1.

albumentations.augmentations.functional.**keypoint_hflip**(*kp*, *rows*, *cols*)
Flip a keypoint horizontally around the y-axis.

albumentations.augmentations.functional.**keypoint_rot90**(*keypoint*, *factor*, *rows*, *cols*, ***params*)
Rotates a keypoint by 90 degrees CCW (see np.rot90)

Parameters

- **keypoint** (*tuple*) – A tuple (x, y, angle, scale).
- **factor** (*int*) – Number of CCW rotations. Must be in range [0;3] See np.rot90.
- **rows** (*int*) – Image rows.
- **cols** (*int*) – Image cols.

albumentations.augmentations.functional.**keypoint_scale**(*keypoint*, *scale_x*, *scale_y*, ***params*)
Scales a keypoint by scale_x and scale_y.

albumentations.augmentations.functional.**keypoint_vflip**(*kp*, *rows*, *cols*)
Flip a keypoint vertically around the x-axis.

albumentations.augmentations.functional.**optical_distortion**(*img*, *k=0*, *dx=0*, *dy=0*, *interpolation=1*, *border_mode=4*, *value=None*)
Barrel / pincushion distortion. Unconventional augment.

Reference:

https://stackoverflow.com/questions/6199636/formulas-for-barrel-pincushion-distortion
https://stackoverflow.com/questions/10364201/image-transformation-in-opencv
https://stackoverflow.com/questions/2477774/correcting-fisheye-distortion-programmatically
http://www.coldvision.io/2017/03/02/advanced-lane-finding-using-opencv/

albumentations.augmentations.functional.**preserve_channel_dim**(*func*)
Preserve dummy channel dim.

albumentations.augmentations.functional.**preserve_shape**(*func*)
Preserve shape of the image.

albumentations.augmentations.functional.**py3round**(*number*)
Unified rounding in all python versions.

Helper functions for working with bounding boxes

albumentations.augmentations.bbox_utils.**normalize_bbox**(*bbox*, *rows*, *cols*)
Normalize coordinates of a bounding box. Divide x-coordinates by image width and y-coordinates by image height.

albumentations.augmentations.bbox_utils.**denormalize_bbox**(*bbox*, *rows*, *cols*)
Denormalize coordinates of a bounding box. Multiply x-coordinates by image width and y-coordinates by image height. This is an inverse operation for `normalize_bbox()`.

albumentations.augmentations.bbox_utils.**normalize_bboxes**(*bboxes*, *rows*, *cols*)
Normalize a list of bounding boxes.

albumentations.augmentations.bbox_utils.**denormalize_bboxes**(*bboxes*, *rows*, *cols*)
Denormalize a list of bounding boxes.

albumentations.augmentations.bbox_utils.**calculate_bbox_area**(*bbox*, *rows*, *cols*)
Calculate the area of a bounding box in pixels.

albumentations.augmentations.bbox_utils.**filter_bboxes_by_visibility**(*original_shape*, *bboxes*, *transformed_shape*, *transformed_bboxes*, *threshold=0.0*, *min_area=0.0*)
Filter bounding boxes and return only those boxes whose visibility after transformation is above the threshold and minimal area of bounding box in pixels is more then min_area.

Parameters

- **original_shape** (*tuple*) – original image shape
- **bboxes** (*list*) – original bounding boxes
- **transformed_shape** (*tuple*) – transformed image
- **transformed_bboxes** (*list*) – transformed bounding boxes
- **threshold** (*float*) – visibility threshold. Should be a value in the range [0.0, 1.0].
- **min_area** (*float*) – Minimal area threshold.

albumentations.augmentations.bbox_utils.**convert_bbox_to_albumentations**(*bbox*, *source_format*, *rows*, *cols*, *check_validity=False*)
Convert a bounding box from a format specified in *source_format* to the format used by albumentations: normalized coordinates of bottom-left and top-right corners of the bounding box in a form of *[x_min, y_min, x_max, y_max]* e.g. *[0.15, 0.27, 0.67, 0.5]*.

Parameters

- **bbox** (*list*) – bounding box
- **source_format** (*str*) – format of the bounding box. Should be 'coco' or 'pascal_voc'.
- **check_validity** (*bool*) – check if all boxes are valid boxes
- **rows** (*int*) – image height
- **cols** (*int*) – image width

Note: The *coco* format of a bounding box looks like *[x_min, y_min, width, height]*, e.g. [97, 12, 150, 200]. The *pascal_voc* format of a bounding box looks like *[x_min, y_min, x_max, y_max]*, e.g. [97, 12, 247, 212].

Raises ValueError – if *target_format* is not equal to *coco* or *pascal_voc*.

albumentations.augmentations.bbox_utils.**convert_bbox_from_albumentations**(*bbox*, *target_format*, *rows*, *cols*, *check_validity=False*)

Convert a bounding box from the format used by albumentations to a format, specified in *target_format*.

Parameters

- **bbox** (*list*) – bounding box with coordinates in the format used by albumentations
- **target_format** (*str*) – required format of the output bounding box. Should be 'coco' or 'pascal_voc'.
- **rows** (*int*) – image height
- **cols** (*int*) – image width
- **check_validity** (*bool*) – check if all boxes are valid boxes

Note: The *coco* format of a bounding box looks like *[x_min, y_min, width, height]*, e.g. [97, 12, 150, 200]. The *pascal_voc* format of a bounding box looks like *[x_min, y_min, x_max, y_max]*, e.g. [97, 12, 247, 212].

Raises `ValueError` – if *target_format* is not equal to *coco* or *pascal_voc*.

albumentations.augmentations.bbox_utils.**convert_bboxes_to_albumentations**(*bboxes*, *source_format*, *rows*, *cols*, *check_validity=False*)

Convert a list bounding boxes from a format specified in *source_format* to the format used by albumentations

albumentations.augmentations.bbox_utils.**convert_bboxes_from_albumentations**(*bboxes*, *target_format*, *rows*, *cols*, *check_validity=False*)

Convert a list of bounding boxes from the format used by albumentations to a format, specified in *target_format*.

Parameters

- **bboxes** (*list*) – List of bounding box with coordinates in the format used by albumentations
- **target_format** (*str*) – required format of the output bounding box. Should be 'coco' or 'pascal_voc'.
- **rows** (*int*) – image height
- **cols** (*int*) – image width
- **check_validity** (*bool*) – check if all boxes are valid boxes

4.3.3 imgaug helpers (albumentations.imgaug)

Transforms

class albumentations.imgaug.transforms.**DualIAATransform**(*always_apply=False*, *p=0.5*)

class albumentations.imgaug.transforms.**ImageOnlyIAATransform**(*always_apply=False*, *p=0.5*)

class albumentations.imgaug.transforms.**IAAEmboss**(*alpha=(0.2, 0.5)*, *strength=(0.2, 0.7)*, *always_apply=False*, *p=0.5*)

Emboss the input image and overlays the result with the original image.

Parameters

- **alpha** (*(float, float)*) – range to choose the visibility of the embossed image. At 0, only the original image is visible,at 1.0 only its embossed version is visible. Default: (0.2, 0.5).
- **strength** (*(float, float)*) – strength range of the embossing. Default: (0.2, 0.7).
- **p** (*float*) – probability of applying the transform. Default: 0.5.

Targets: image

class albumentations.imgaug.transforms.**IAASuperpixels**(*p_replace=0.1*, *n_segments=100*, *always_apply=False*, *p=0.5*)

Completely or partially transform the input image to its superpixel representation. Uses skimage's version of the SLIC algorithm. May be slow.

Parameters

- **p_replace** (*float*) – defines the probability of any superpixel area being replaced by the superpixel, i.e. by the average pixel color within its area. Default: 0.1.
- **n_segments** (*int*) – target number of superpixels to generate. Default: 100.
- **p** (*float*) – probability of applying the transform. Default: 0.5.

Targets: image

class albumentations.imgaug.transforms.**IAASharpen**(*alpha=(0.2, 0.5)*, *lightness=(0.5, 1.0)*, *always_apply=False*, *p=0.5*)

Sharpen the input image and overlays the result with the original image.

Parameters

- **alpha** (*(float, float)*) – range to choose the visibility of the sharpened image. At 0, only the original image is visible, at 1.0 only its sharpened version is visible. Default: (0.2, 0.5).
- **lightness** (*(float, float)*) – range to choose the lightness of the sharpened image. Default: (0.5, 1.0).
- **p** (*float*) – probability of applying the transform. Default: 0.5.

Targets: image

class albumentations.imgaug.transforms.**IAAAdditiveGaussianNoise**(*loc=0, scale=(2.5500000000000003, 12.75), per_channel=False, always_apply=False, p=0.5*)

Add gaussian noise to the input image.

Parameters

- **loc** (*int*) – mean of the normal distribution that generates the noise. Default: 0.
- **scale** (*(float, float)*) – standard deviation of the normal distribution that generates the noise. Default: (0.01 * 255, 0.05 * 255).
- **p** (*float*) – probability of applying the transform. Default: 0.5.

Targets: image

class albumentations.imgaug.transforms.**IAACropAndPad**(*px=None, percent=None, pad_mode='constant', pad_cval=0, keep_size=True, always_apply=False, p=1*)

class albumentations.imgaug.transforms.**IAAFliplr**(*always_apply=False, p=0.5*)

class albumentations.imgaug.transforms.**IAAFlipud**(*always_apply=False, p=0.5*)

class albumentations.imgaug.transforms.**IAAAffine**(*scale=1.0, translate_percent=None, translate_px=None, rotate=0.0, shear=0.0, order=1, cval=0, mode='reflect', always_apply=False, p=0.5*)

Place a regular grid of points on the input and randomly move the neighbourhood of these point around via affine transformations.

Note: This class introduce interpolation artifacts to mask if it has values other than {0;1}

Parameters **p** (*float*) – probability of applying the transform. Default: 0.5.

Targets: image, mask

class albumentations.imgaug.transforms.**IAAPiecewiseAffine**(*scale=(0.03, 0.05), nb_rows=4, nb_cols=4, order=1, cval=0, mode='constant', always_apply=False, p=0.5*)

Place a regular grid of points on the input and randomly move the neighbourhood of these point around via affine transformations.

Note: This class introduce interpolation artifacts to mask if it has values other than {0;1}

Parameters

- **scale** (*(float, float*) – factor range that determines how far each point is moved. Default: (0.03, 0.05).
- **nb_rows** (*int*) – number of rows of points that the regular grid should have. Default: 4.

- **nb_cols** (*int*) – number of columns of points that the regular grid should have. Default: 4.
- **p** (*float*) – probability of applying the transform. Default: 0.5.

Targets: image, mask

class albumentations.imgaug.transforms.**IAAPerspective**(*scale=(0.05, 0.1)*, *keep_size=True*, *always_apply=False*, *p=0.5*)

Perform a random four point perspective transform of the input.

Note: This class introduce interpolation artifacts to mask if it has values other than {0;1}

Parameters

- **scale** (*(float, float)*) – standard deviation of the normal distributions. These are used to sample the random distances of the subimage's corners from the full image's corners. Default: (0.05, 0.1).
- **p** (*float*) – probability of applying the transform. Default: 0.5.

Targets: image, mask

4.3.4 PyTorch helpers (albumentations.pytorch)

Transforms

class albumentations.pytorch.transforms.**ToTensor**(*num_classes=1*, *sigmoid=True*, *normalize=None*)

Convert image and mask to *torch.Tensor* and divide by 255 if image or mask are *uint8* type. WARNING! Please use this with care and look into sources before usage.

Parameters

- **num_classes** (*int*) – only for segmentation
- **sigmoid** (*bool, optional*) – only for segmentation, transform mask to LongTensor or not.
- **normalize** (*dict, optional*) – dict with keys [mean, std] to pass it into torchvision.normalize

4.4 About probabilities.

4.4.1 Default probability values

Compose, **PadIfNeeded**, **CenterCrop**, **RandomCrop**, **Crop**, **Normalize**, **ToFloat**, **FromFloat**, **ToTensor**, **LongestMaxSize** have default probability values equal to **1**. All other are equal to **0.5**

```
from albumentations import (
    RandomRotate90, IAAAdditiveGaussianNoise, GaussNoise
)
import numpy as np

def aug(p1):
```

(continues on next page)

(continued from previous page)

```
    return Compose([
        RandomRotate90(p=p2),
        OneOf([
            IAAAdditiveGaussianNoise(p=0.9),
            GaussNoise(p=0.6),
        ], p3=0.2)
    ], p=p1)

image = np.ones((300, 300, 3), dtype=np.uint8)
mask = np.ones((300, 300), dtype=np.uint8)
whatever_data = "my name"
augmentation = aug(p=0.9)
data = {"image": image, "mask": mask, "whatever_data": whatever_data, "additional":
↪"hello"}
augmented = augmentation(**data)
image, mask, whatever_data, additional = augmented["image"], augmented["mask"],␣
↪augmented["whatever_data"], augmented["additional"]
```

In the above augmentation pipeline, we have three types of probabilities. Combination of them is the primary factor that decides how often each of them will be applied.

1. **p1**: decides if this augmentation will be applied. The most common case is **p1=1** means that we always apply the transformations from above. **p1=0** will mean that the transformation block will be ignored.
2. **p2**: every augmentation has an option to be applied with some probability.
3. **p3**: decide if **OneOf** will be applied.

4.4.2 OneOf Block

To decide which augmentation within **OneOf** block is used the following rule is applied.

1. We normalize all probabilities within a block to one. After this we pick augmentation based on the normalized probabilities. In the example above **IAAAdditiveGaussianNoise** has probability **0.9** and **GaussNoise** probability **0.6**. After normalization, they become **0.6** and **0.4**. Which means that we decide if we should use **IAAAdditiveGaussianNoise** with probability **0.6** and **GaussNoise** otherwise.
2. If we picked to consider **GaussNoise** the next step will be to decide if we should use it or not and **p=0.6** will be used in this case.

4.4.3 Example calculations

Thus, each augmentation in the example above will be applied with the probability:

1. **RandomRotate90**: *p1 * p2*
2. **IAAAdditiveGaussianNoise**: *p1 * (0.9) / (0.9 + 0.6) * 0.9*
3. **GaussianNoise**: *p1 * (0.6) / (0.9 + 0.6) * 0.6*

4.5 Writing tests

4.5.1 A first test.

We use `pytest` to run tests for albumentations. Python files with tests should be placed inside the `albumentations/tests` directory, filenames should start with `test_`, for example `test_bbox.py`. Names of test functions should also start with `test_`, for example, `def test_random_brightness():`.

Let's say that we want to test the `brightness_contrast_adjust` function. The purpose of this function is to take a NumPy array as input and multiply all the values of this array by a value specified in the argument `alpha`.

We will write a first test for this function that will check that if you pass a NumPy array with all values equal to 128 and a parameter `alpha` that equals to 1.5 as inputs the function should produce a NumPy array with all values equal to 192 as output (that's because 128 * 1.5 = 192).

In the directory `albumentations/tests` we will create a new file and name it `test_example.py`

Let's add all the necessary imports:

```
import numpy as np

import albumentations.augmentations.functional as F
```

Then let's add the test itself:

```
def test_random_contrast():
    img = np.ones((100, 100, 3), dtype=np.uint8) * 128
    img = F.brightness_contrast_adjust(img, alpha=1.5)
    expected_brightness = 192
    expected = np.ones((100, 100, 3), dtype=np.uint8) * expected_multiplier
    assert np.array_equal(img, expected)
```

We can run tests from `test_example.py` (right now it contains only one test) by executing the following command: `pytest tests/test_example.py -v`. The `-v` flag tells pytest to produce a more verbose output.

`pytest` will show that the test has been completed successfully:

```
tests/test_example.py::test_random_brightness PASSED
```

4.5.2 Test parametrization and the @pytest.mark.parametrize decorator.

Let's say that we also want to test that the function `brightness_contrast_adjust` correctly handles a situation in which after multiplying an input array by `alpha` some output values exceed 255. Because when we a pass a NumPy array with the data type `np.uint8` as input we expect that we will also get an array with the `np.uint8` data type as output and that means that output values should not exceed 255 (which is the maximum value for this data type). We also want to check that values don't overflow, so if inside the function we get a value 256 we should clip it to 255 and not overflow to 0.

Let's write a test:

```
def test_random_contrast_2():
    img = np.ones((100, 100, 3), dtype=np.uint8) * 128
    img = F.brightness_contrast_adjust(img, alpha=3)
    expected_multiplier = 255
    expected = np.ones((100, 100, 3), dtype=np.uint8) * expected_multiplier
    assert np.array_equal(img, expected)
```

Next, we will run the tests from `test_example.py`: `pytest tests/test_example.py -v`

Output:

```
tests/test_example.py::test_random_brightness PASSED
tests/test_example.py::test_random_brightness_2 PASSED
```

As we see functions `test_random_brightness` and `test_random_brightness_2` looks almost the same, the only difference is the values of `alpha` and `expected_multiplier`. To get rid of code duplication we can use the `@pytest.mark.parametrize` decorator. With this decorator we can describe which values should be passed as arguments to the test and the pytest will run the test multiple times, each time passing the next value from the decorator.

We can rewrite two previous tests as a one test using parametrization:

```
import pytest

@pytest.mark.parametrize(['alpha', 'expected_multiplier'], [(1.5, 192), (3, 255)])
def test_random_brightness(alpha, expected_multiplier):
    img = np.ones((100, 100, 3), dtype=np.uint8) * 128
    img = F.brightness_contrast_adjust(img, alpha=alpha)
    expected = np.ones((100, 100, 3), dtype=np.uint8) * expected_multiplier
    assert np.array_equal(img, expected)
```

This test will run two times, in the first run the `alpha` argument will be equal to 1.5 and the `expected_multiplier` argument will be equal to 192. In the second run the `alpha` argument will be equal to 3 and the `expected_multiplier` argument will be equal to 255.

Let's run this test:

```
tests/test_example.py::test_random_brightness[1.5-192] PASSED
tests/test_example.py::test_random_brightness[3-255] PASSED
```

As we see pytest prints arguments values at each run.

4.5.3 Simplifying tests for functions that work with both images and masks by using helper functions.

Let's say that we want to test the `hflip` function. This function vertically flips an image or mask that passed as input to it.

We will start with a test that checks that this function works correctly with masks, that is with two-dimensional NumPy arrays that have shape `(height, width)`.

```
def test_vflip_mask():
    mask = np.array(
        [[1, 1, 1],
         [0, 1, 1],
         [0, 0, 1]], dtype=np.uint8)
    expected_mask = np.array(
        [[0, 0, 1],
         [0, 1, 1],
         [1, 1, 1]], dtype=np.uint8)
    flipped_mask = F.vflip(mask)
    assert np.array_equal(flipped_mask, expected_mask)
```

Test running result:

```
tests/test_example.py::test_vflip_mask PASSED
```

Next, we will make a test that checks how the same function works with RGB-images, that is with three-dimensional NumPy arrays that have shape `(height, width, 3)`.

```
def test_vflip_img():
    img = np.array(
        [[[1, 1, 1],
          [1, 1, 1],
          [1, 1, 1]],
         [[0, 0, 0],
          [1, 1, 1],
          [1, 1, 1]],
         [[0, 0, 0],
          [0, 0, 0],
          [1, 1, 1]]], dtype=np.uint8)
    expected_img = np.array(
        [[[0, 0, 0],
          [0, 0, 0],
          [1, 1, 1]],
         [[0, 0, 0],
          [1, 1, 1],
          [1, 1, 1]],
         [[1, 1, 1],
          [1, 1, 1],
          [1, 1, 1]]], dtype=np.uint8)
    flipped_img = F.vflip(img)
    assert np.array_equal(flipped_img, expected_img)
```

In this test, the value of `img` is the same NumPy array that was assigned to the `mask` variable in `test_vflip_mask`, but this time it is repeated three times (one time for each of the three channels). And `expected_img` is also a repeated three times NumPy array that was assigned to the `expected_mask` variable in `test_vflip_mask`.

Let's run the test:

```
tests/test_example.py::test_vflip_img PASSED
```

In `test_vflip_img` we manually defined values of `img` and `expected_img` that equal to repeated three times values of `mask` and `expected_mask` respectively. To avoid unnecessary and duplicate code we can make a helper function that takes a NumPy array with shape `(height, width)` as input and repeats this value 3 times along a new axis to produce a NumPy array with shape `(height, width, 3)`:

```
def convert_2d_to_3d(array, num_channels=3):
    return np.repeat(array[:, :, np.newaxis], repeats=num_channels, axis=2)
```

Next, we can use this function to rewrite `test_vflip_img` as follows:

```
def test_vflip_img_2():
    mask = np.array(
        [[1, 1, 1],
         [0, 1, 1],
         [0, 0, 1]], dtype=np.uint8)
    expected_mask = np.array(
        [[0, 0, 1],
         [0, 1, 1],
         [1, 1, 1]], dtype=np.uint8)
    img = convert_2d_to_3d(mask)
```

(continues on next page)

(continued from previous page)

```
    expected_img = convert_2d_to_3d(expected_mask)
    flipped_img = F.vflip(img)
    assert np.array_equal(flipped_img, expected_img)
```

Let's run the test:

```
tests/test_example.py::test_vflip_img_2 PASSED
```

4.5.4 Simplifying tests for functions that work with both images and masks by using parametrization.

In the previous section we wrote two separate tests for `vflip`, the first one checked how `vflip` works with masks, the second one checked how `vflip` works with images.

Those tests share a large amount of the same code between them, so we can move common parts to a single function and use parametrization to pass information about input type as an argument to the test:

```
@pytest.mark.parametrize('target', ['mask', 'image'])
def test_vflip_img_and_mask(target):
    img = np.array(
        [[1, 1, 1],
         [0, 1, 1],
         [0, 0, 1]], dtype=np.uint8)
    expected = np.array(
        [[0, 0, 1],
         [0, 1, 1],
         [1, 1, 1]], dtype=np.uint8)
    if target == 'image':
        img = convert_2d_to_3d(img)
        expected = convert_2d_to_3d(expected)
    flipped_img = F.vflip(img)
    assert np.array_equal(flipped_img, expected)
```

This test will run two times, in the first run the `target` argument will be equal to `'mask'`, the condition `if target == 'image':` will not be executed and the test will check how `vflip` works with masks. In the second run the `target` argument will be equal to `'image'`, the condition `if target == 'image':` will be executed and the test will check how `vflip` works with images:

```
tests/test_example.py::test_vflip_img_and_mask[mask] PASSED
tests/test_example.py::test_vflip_img_and_mask[image] PASSED
```

We can reduce the amount of code even further by moving logic under `if target == 'image'` to a separate function:

```
def convert_2d_to_target_format(*arrays, target=None):
    if target == 'mask':
        return arrays[0] if len(arrays) == 1 else arrays
    elif target == 'image':
        return tuple(convert_2d_to_3d(array, num_channels=3) for array in arrays)
    else:
        raise ValueError('Unknown target {}'.format(target))
```

This function will take NumPy arrays with shape `(height, width)` as inputs and depending on the value of `target` will either return them as is or convert them to NumPy arrays with shape `(height, width, 3)`.

Using this helper function we can rewrite the test as follows:

```python
@pytest.mark.parametrize('target', ['mask', 'image'])
def test_vflip_img_and_mask(target):
    img = np.array(
        [[1, 1, 1],
         [0, 1, 1],
         [0, 0, 1]], dtype=np.uint8)
    expected = np.array(
        [[0, 0, 1],
         [0, 1, 1],
         [1, 1, 1]], dtype=np.uint8)
    img, expected = convert_2d_to_target_format(img, expected, target=target)
    flipped_img = F.vflip(img)
    assert np.array_equal(flipped_img, expected)
```

pytest output:

```
tests/test_example.py::test_vflip_img_and_mask[mask] PASSED
tests/test_example.py::test_vflip_img_and_mask[image] PASSED
```

Implementation notes:

Implementations of `convert_2d_to_target_format` and `convert_2d_to_3d` in albumentations slightly differ from implementations described above. We need to support both Python 2.7 and Python 3, so we can't use a function declaration like `def convert_2d_to_target_format(*arrays, target=None)` because it produces `SyntaxError` in Python 2 and only valid in Python 3 (see PEP3102 for more details). Because of this we use the following function declaration: `def convert_2d_to_target_format(arrays, target)` where the `arrays` argument should contain a list of NumPy arrays.

The test can be rewritten as follows to be compatible with the current albumentations' test suite (note an updated call to `convert_2d_to_target_format`, we pass `img` and `expected` arguments inside a single list):

```python
@pytest.mark.parametrize('target', ['mask', 'image'])
def test_vflip_img_and_mask(target):
    img = np.array(
        [[1, 1, 1],
         [0, 1, 1],
         [0, 0, 1]], dtype=np.uint8)
    expected = np.array(
        [[0, 0, 1],
         [0, 1, 1],
         [1, 1, 1]], dtype=np.uint8)
    img, expected = convert_2d_to_target_format([img, expected], target=target)
    flipped_img = F.vflip(img)
    assert np.array_equal(flipped_img, expected)
```

4.5.5 Using fixtures.

Let's say that we want to test a situation in which we pass an image and mask with the `np.uint8` data type to the `VerticalFlip` augmentation and we expect that it won't change data types of inputs and will produce an image and mask with the `np.uint8` data type as output.

Such a test can be written as follows:

```
from albumentations import VerticalFlip

def test_vertical_flip_dtype():
    aug = VerticalFlip(p=1)
    image = np.random.randint(low=0, high=256, size=(100, 100, 3), dtype=np.uint8)
    mask = np.random.randint(low=0, high=2, size=(100, 100), dtype=np.uint8)
    data = aug(image=image, mask=mask)
    assert data['image'].dtype == np.uint8
    assert data['mask'].dtype == np.uint8
```

We generate a random image and a random mask, then we pass them as inputs to the augmentation and then we check a data type of output values.

If we want to perform this check for other augmentations as well, we will have to write code to generate a random image and mask at the beginning of each test:

```
image = np.random.randint(low=0, high=256, size=(100, 100, 3), dtype=np.uint8)
mask = np.random.randint(low=0, high=2, size=(100, 100), dtype=np.uint8)
```

To avoid this duplication we can move code that generates random values to a fixture. Fixtures work as follows:

1. In the `tests/conftest.py` file we create functions that are wrapped with the `@pytest.fixture` decorator:

```
@pytest.fixture
def image():
    return np.random.randint(low=0, high=256, size=(100, 100, 3), dtype=np.uint8)

@pytest.fixture
def mask():
    return np.random.randint(low=0, high=2, size=(100, 100), dtype=np.uint8)
```

2. In our test we use fixture names as accepted arguments:

```
def test_vertical_flip_dtype(image, mask):
    ...
```

3. pytest will use arguments' names to find fixtures with the same names, then it will execute those fixture functions and will pass the outputs of this functions as arguments to the test function.

We can rewrite `test_vertical_flip_dtype` using fixtures as follows:

```
def test_vertical_flip_dtype(image, mask):
    aug = VerticalFlip(p=1)
    data = aug(image=image, mask=mask)
    assert data['image'].dtype == np.uint8
    assert data['mask'].dtype == np.uint8
```

4.5.6 Simultaneous use of fixtures and parametrization.

Let's say that besides `VerticalFlip` we also want to test that `HorizontalFlip` also returns values with the `np.uint8` data type if we passed a `np.uint8` input to it.

We can write test like this:

```
from albumentations import HorizontalFlip

def test_horizontal_flip_dtype(image, mask):
    aug = HorizontalFlip(p=1)
    data = aug(image=image, mask=mask)
    assert data['image'].dtype == np.uint8
    assert data['mask'].dtype == np.uint8
```

But this test is almost completely identical to `test_vertical_flip_dtype`. And to check each new augmentation we will have to copy practically almost the whole code from `test_vertical_flip_dtype` and change the value of the `aug` variable, so the test will use a new augmentation. However it would be great to get rid of unnecessary copying of code in tests. For this, we could use parametrization and pass a class as a parameter.

A test that checks both VerticalFlip and HorizontalFlip can be written as follows:

```
from albumentations import VerticalFlip, HorizontalFlip

@pytest.mark.parametrize('augmentation_cls', [
    VerticalFlip,
    HorizontalFlip,
])
def test_multiple_augmentations(augmentation_cls, image, mask):
    aug = augmentation_cls(p=1)
    data = aug(image=image, mask=mask)
    assert data['image'].dtype == np.uint8
    assert data['mask'].dtype == np.uint8
```

This test will run two times, in the first run the `augmentation_cls` argument will be equal to `VerticalFlip`. In the second run the `augmentation_cls` argument will be equal to `HorizontalFlip`.

pytest output:

```
tests/test_example.py::test_multiple_augmentations[VerticalFlip] PASSED
tests/test_example.py::test_multiple_augmentations[HorizontalFlip] PASSED
```

Bibliography

[Simard2003] Simard, Steinkraus and Platt, "Best Practices for Convolutional Neural Networks applied to Visual Document Analysis", in Proc. of the International Conference on Document Analysis and Recognition, 2003.

[Simard2003] Simard, Steinkraus and Platt, "Best Practices for Convolutional Neural Networks applied to Visual Document Analysis", in Proc. of the International Conference on Document Analysis and Recognition, 2003.

[Simard2003] Simard, Steinkraus and Platt, "Best Practices for Convolutional Neural Networks applied to Visual Document Analysis", in Proc. of the International Conference on Document Analysis and Recognition, 2003.

Python Module Index

a

Index

A

B

C

N

O

P

R

S

T

V