albumentations Documentation

Release 1.1.0

Alexander Buslaev, Alex Parinov, Vladimir Iglovikov, Evegene Khv

Jun 10, 2022

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/albumentations-team/albumentations
- GitHub repository with examples: https://github.com/albumentations-team/albumentations_examples
- 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/albumentations-team/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 repository with examples and example.ipynb and example_16_bit_tiff.ipynb

4.2 Contributing

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

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

4.3 To create a pull request:

1. Fork the repository.
2. Clone it.
3. Install pre-commit hook:

```
pip install pre-commit black flake8 isort mypy
```

4. Initialize it from the folder with the repo:

```
pre-commit install
```

4. Make desired changes to the code.
5. Install the library in development mode:

```
pip install -e .[tests]
```

6. Run tests:

```
pytest
```

7. Push code to your forked repo.

8. Create pull request.

4.4 Augmentations overview

You can find examples in this repository.

4.5 API

4.5.1 Core API (albumentations.core)

Composition

Transforms interface

Serialization

4.5.2 Augmentations (albumentations.augmentations)

Transforms

Functional transforms

Helper functions for working with bounding boxes

Helper functions for working with keypoints

4.5.3 imgaug helpers (albumentations.imgaug)

Transforms

4.5.4 PyTorch helpers (albumentations.pytorch)

Transforms

4.6 About probabilities.

4.6.1 Default probability values

All pre / post processing transforms: **Compose**, **PadIfNeeded**, **CenterCrop**, **RandomCrop**, **Crop**, **RandomCropNearBBox**, **RandomSizedCrop**, **RandomResizedCrop**, **RandomSizedBBoxSafeCrop**, **CropNonEmptyMaskIfExists**, **Lambda**, **Normalize**, **ToFloat**, **FromFloat**, **ToTensor**, **LongestMaxSize** have default probability values equal to **1**. All others are equal to **0.5**

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

(continues on next page)

(continued from previous page)

```
def aug(p1, p2, p3):
    return Compose([
        RandomRotate90(p=p2),
        OneOf([
            IAAAdditiveGaussianNoise(p=0.9),
            GaussNoise(p=0.6),
        ], p=p3)
    ], 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(p1=0.9, p2=0.7, p3=0.3)
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.

4.6.2 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 we call **GaussNoise** with flag **force_apply=True**.

4.6.3 Example calculations

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

1. **RandomRotate90**: *p1 * p2*
2. **IAAAdditiveGaussianNoise**: *p1 * p3 * (0.9 / (0.9 + 0.6))*
3. **GaussianNoise**: *p1 * p3 * (0.6 / (0.9 + 0.6))*

4.7 Writing tests

4.7.1 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_multiplier = 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
```

4.7.2 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.

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

Let's say that we want to test the `vflip` 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
```

4.7.4 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:

```
@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):

```
@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)
```

4.7.5 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
```

4.7.6 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
```

4.8 Hall of Fame

Albumentations are widely used in Computer Vision Competitions at Kaggle an other platforms.

Here are the links to the competitions, names of the winners and to their solutions.

We follow these rules, when adding a solution to the "Hall of Fame":

1. There should be a description of the solution: post at the forum / code / blog post / paper / pre-print.
2. Solution should have some value:

- For Kaggle: gold or silver medal solutions.
- For Topcoder and other platforms: in money.
- For competitions held as a part of the academic conferences: there is a paper or pre-print describing the solution.

4.8.1 Kaggle

Carvana Image Masking Challenge

1. **Vladimir Iglovikov**, **Alexander Buslaev**, **Artem Sanakoev** solution

Data Science Bowl 2018

1. **Alexander Buslaev**, **Selim Seferbekov**, **Victor Durnov** solution

Humpback Whale Identification

5. **Roman Solovyev**, **Weimin Wang** blog post, code

TGS Salt Identification Challenge

1. **b.e.s.**, **phalanx** solution, code, pre-print
27. **Insaf Ashrapov**, **Mikhail Karchevskiy**, **Leonid Kozinkin** blog post, code, pre-print

APTOS 2019 Blindness Detection

7. **Eugene Khvedchenya** solution, code
76. **Insaf Ashrapov**, **Mamat Shamshiev**, **Mishunyayev Nikita** solution, code

SIIM-ACR Pneumothorax Segmentation

1. **Anuar Aimoldin** solution, code, video, presentation
4. **Miras Amir** solution, code
33. **Renat Alimbekov**, **Ivan Vassilenko** solution
50. **AlexeyK**, **wayfarer**, **Kudaibergen R** code and solution

iMaterialist (Fashion) 2019 at FGVC6

1. **Miras Amir** solution, code

Google Landmark Recognition 2019

20. **Artyom Palvelev** solution, code

Inclusive Images Challenge

3. **Roman Solovyev**, **Weimin Wang** solution

Airbus Ship Detection Challenge

30. **Konstantin Maksimov** paper

Severstal: Steel Defect Detection Challenge

27. **Ilia Larchenko** solution

4.8.2 Topcoder

2019

Neptune - Facial Detection Marathon Match

2. **Miras Amir** solution

Neptune - Facial Re-Identification Marathon Match

2. **Miras Amir** solution

2018

SpaceNet Challenge Round 4: Off-Nadir Buildings

3. **Konstantin Maksimov** solution

4.8.3 CVPR

2018

DeepGlobe: Road Extraction

2. **Vladimir Iglovikov**, **Alexander Buslaev**, **Selim Seferbekov**, **Alexey Shvets** paper

Deepglobe: Building detection

2. **Vladimir Iglovikov**, **Alexander Buslaev**, **Selim Seferbekov**, **Alexey Shvets** paper

Deepglobe: Land Cover Classification

3. **Vladimir Iglovikov**, **Alexander Buslaev**, **Selim Seferbekov**, **Alexey Shvets** paper

4.8.4 MICCAI

2017

Robotic Instrument Segmentation

1. **Vladimir Iglovikov**, **Alexey Shvets** paper, pre-print from organizers

GIANA: Angiodysplasia localization

1. **Vladimir Iglovikov**, **Alexey Shvets** paper

4.9 Citations

Albumentations is widely used in research areas related to computer vision and deep learning. If you find this library useful for your research, please consider citing:

```
@article{2018arXiv180906839B,
  author = {A. Buslaev, A. Parinov, E. Khvedchenya, V.~I. Iglovikov and A.~A. Kalinin}
↪,
  title = "{Albumentations: fast and flexible image augmentations}",
  journal = {ArXiv e-prints},
  eprint = {1809.06839},
  year = 2018
}
```

4.9.1 List of papers that cite Albumentations

1. Camera Model Identification Using Convolutional Neural Networks.

October 2018 - A Kuzin, A Fattakhov, I Kibardin.

2. Automatic lesion boundary detection in dermoscopy.

December 2018 - G Kechyn.

3. Automatic salt deposits segmentation: A deep learning approach.

December 2018 - M Karchevskiy, I Ashrapov, L Kozinkin.

4. Adapting Convolutional Neural Networks for Geographical Domain Shift.

January 2019 - P Ostyakov, SI Nikolenko.

5. Cell Nuclear Morphology Analysis Using 3D Shape Modeling, Machine Learning and Visual Analytics.

February 2019 - A Kalinin.

6. Safe Augmentation: Learning Task-Specific Transformations from Data.

February 2019 - I Baran, O Kupyn, A Kravchenko.

7. Self-supervised Learning for Dense Depth Estimation in Monocular Endoscopy.

February 2019 - X Liu, A Sinha, M Ishii, GD Hager, A Reiter.

8. U-NetPlus: A Modified Encoder-Decoder U-Net Architecture for Semantic and Instance Segmentation of Surgical Instrument.

February 2019 - SM Hasan, CA Linte.

9. General Purpose (GenP) Bioimage Ensemble of Handcrafted and Learned Features with Data Augmentation.

March 2019 - L Nanni, S Brahnam, S Ghidoni, G Maguolo.

10. Breast Tumor Cellularity Assessment using Deep Neural Networks.

May 2019 - A Rakhlin, AA Shvets, AA Kalinin, A Tiulpin.

11. A Deep 3D Object Pose Estimation Framework for Robots with RGB-D Sensors.

June 2019 - AY Wagh.

12. Data Augmentation From RGB to Chlorophyll Fluorescence Imaging Application to Leaf Segmentation of Arabidopsis thaliana From Top View Images.

June 2019 - N Sapoukhina, S Samiei, P Rasti.

13. CLoDSA: a tool for augmentation in classification, localization, detection, semantic segmentation and instance segmentation tasks.

June 2019 - Á Casado-García, C Domínguez.

14. Urban Sound Tagging using Convolutional Neural Networks.

July 2019 - S Adapa.

15. A Framework for Knowing Who is Doing What in Aerial Surveillance Videos.

July 2019 - F Yang, S Sakti, Y Wu, S Nakamura.

16. Visual Anomaly Detection For Automatic Quality Control.

July 2019 - F Piccoli.

17. Physical Cue based Depth-Sensing by Color Coding with Deaberration Network.

August 2019 - N Mishima, T Kozakaya, A Moriya, R Okada.

18. Bayesian Feature Pyramid Networks for Automatic Multi-Label Segmentation of Chest X-rays and Assessment of Cardio-Thoratic Ratio.

August 2019 - R Solovyev, I Melekhov, T Lesonen.

19. DeblurGAN-v2: Deblurring (Orders-of-Magnitude) Faster and Better.

August 2019 - O Kupyn, T Martyniuk, J Wu, Z Wang.

20. The Impact of Padding on Image Classification by Using Pre-trained Convolutional Neural Networks.

September 2019 - H Tang, A Ortis, S Battiato.

21. A load frame for in situ tomography at PETRA III.

September 2019 - J Moosmann, DCF Wieland.

22. Deep convolutions for in-depth automated rock typing.

September 2019 - EE Baraboshkin, LS Ismailova, DM Orlov.

23. CGAN .

September 2019 - .

24. Cloud Recognition and Masking of Earth Observation Imagery-An Optimized approach for Automatic Labeling of Sentinel-2 Imagery for Object Detection.

September 2019 - LM Ellefsen.