Hands-On AI Part 14: Image Data Preprocessing and Augmentation

Published on October 13, 2017 Translate

A Tutorial Series for Software Developers, Data Scientists, and Data Center Managers

Preprocessing is the general term for all the transformation done to the data before feeding them into the model, including centering, normalization, shift, rotation, shear, and so on. Generally, there are two occasions when one might want to do preprocessing:

- **Cleaning up the data.** Let's assume that you have some artifacts in the images. To make the learning process easier for the model, we can remove the artifacts using preprocessing.
- ***Augmenting*** the data. Sometimes small datasets are not enough for the deep model to learn sufficiently well. The ***data augmentation*** approach is useful in solving this problem. It is the process of transforming each data sample in numerous possible ways and adding all of the augmented samples to the dataset. By doing this one can increase the effective size of the dataset. Transformations to apply are usually chosen randomly from the predefined set.

Let's take a look at some of the possible preprocessing transformations and see how they can be implemented via Keras*. All the materials including corresponding code, notebook, and Dockerfile* are located on Dropbox (https://www.dropbox.com/sh/wxg73c4wfmzuaer/AADx_2qeOUKCjk7r85R7jSwba?dl=0)* (http://www.dropbox.com/sh/wxg73c4wfmzuaer/AADx_2qeOUKCjk7r85R7jSwba?dl=0).

Data

In this and following articles we will use the image sentiment analysis dataset. It contains about 1,500 examples of images divided into two classes—positive and negative. Let's take a look at some examples.

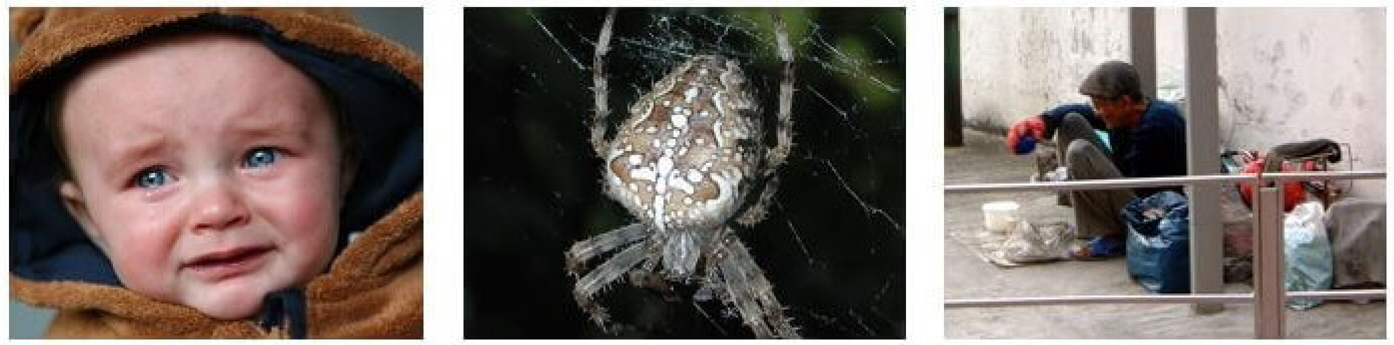


Figure 1: *Negative examples.*

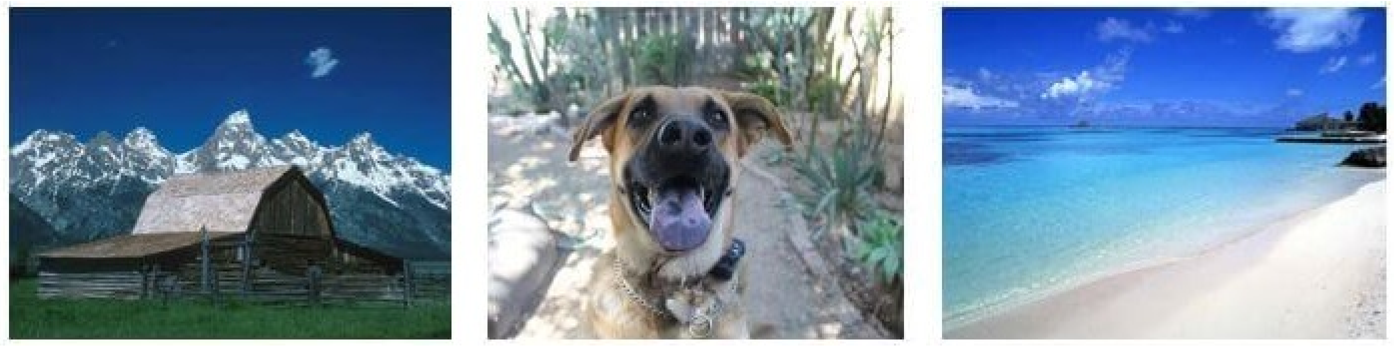


Figure 2: *Positive examples.*

Cleaning Transformations

Now, let’s take a look at the set of possible transformations that are usually applied for cleaning up the data, their implementation, and influence on images.

All the code snippets can be found in the *Preprocessing.ipynb* (https://www.dropbox.com/sh/wxg73c4wfmzuaer/AADI2sA7VBMuE53CJp6-N5ICa/notebooks/Preprocessing.ipynb?dl=0) notebook.

- ### Rescaling

  The images are usually stored in an RGB (Red Green Blue) format. In this format the image is represented as a three-dimensional (or three-channel) array.

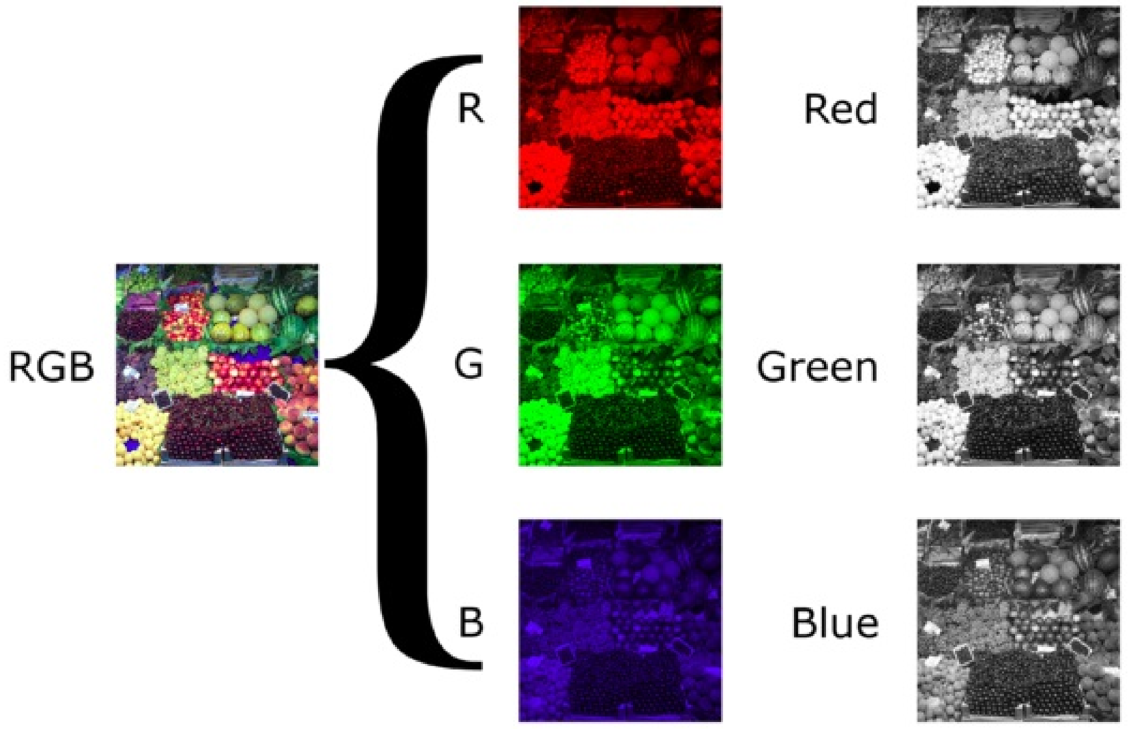


Figure 3: *RGB decomposition of the image. The diagram is taken from Wikiwand (http://www.wikiwand.com/en/Grayscale)*.*

One dimension is for channels (red, green, and blue colors) and two other dimensions are spatial dimension. Thus, every pixel is encoded through three numbers. Each number is usually stored as an 8-bit unsigned integer type (0 to 255).

Rescaling is an operation that moves your data from one numerical range to another by simple division using a predefined constant. In deep neural networks you might want to restrict your input to the range from 0 to 1, due to possible overflow, optimization, stability issues, and so on.

For example, let's cast our data from [0; 255] range to [0; 1] range. Here and below we will use the Keras *ImageDataGenerator* (https://keras.io/preprocessing/image/) class, which allows us to do all transformations on the fly.

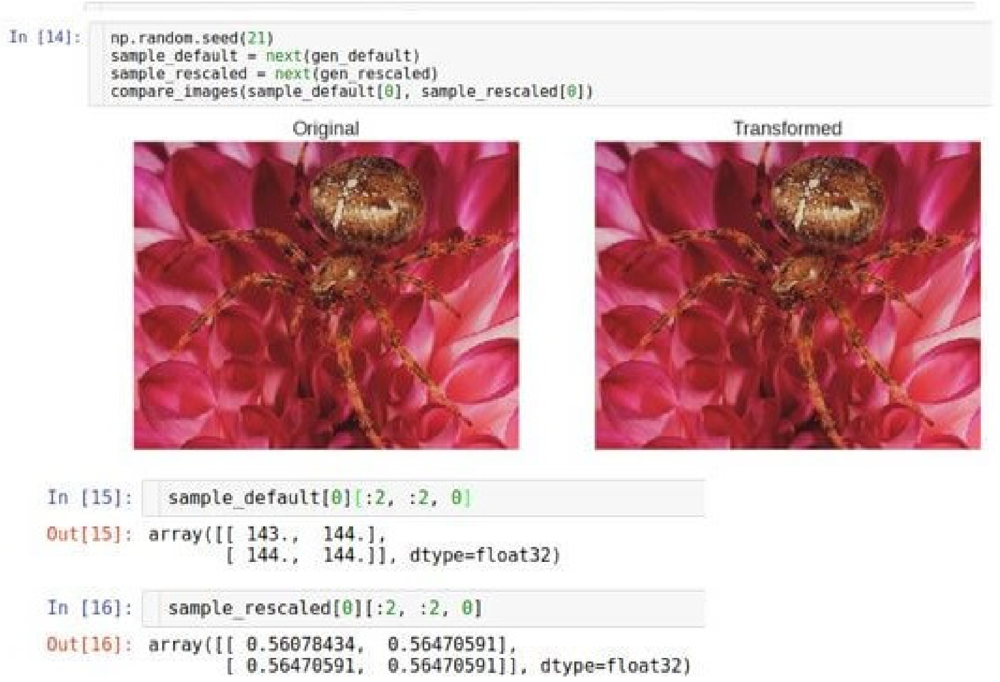
Let’s create two instances of this class: one for transformed data and one for the initial

```
In [12]:  datagen_rescaled = ImageDataGenerator(rescale=1. / 255.)
          datagen_default = ImageDataGenerator()
```

(or default). We just need to specify the scaling constant. Moreover, the ImageDataGenerator class allows us to stream the data directly from the hard drive directory using the flow_from_directory method.

All the parameters can be found in the [documentation (https://keras.io/preprocessing/image/)](https://keras.io/preprocessing/image/), but the main parameters are the path to stream from and the target size of the image (the generator would just crop or pad the image if it doesn’t fit the target size). Finally, let’s get a sample from the generator and see the results.

Visually both images are identical, but that’s just because Python* tools rescale images



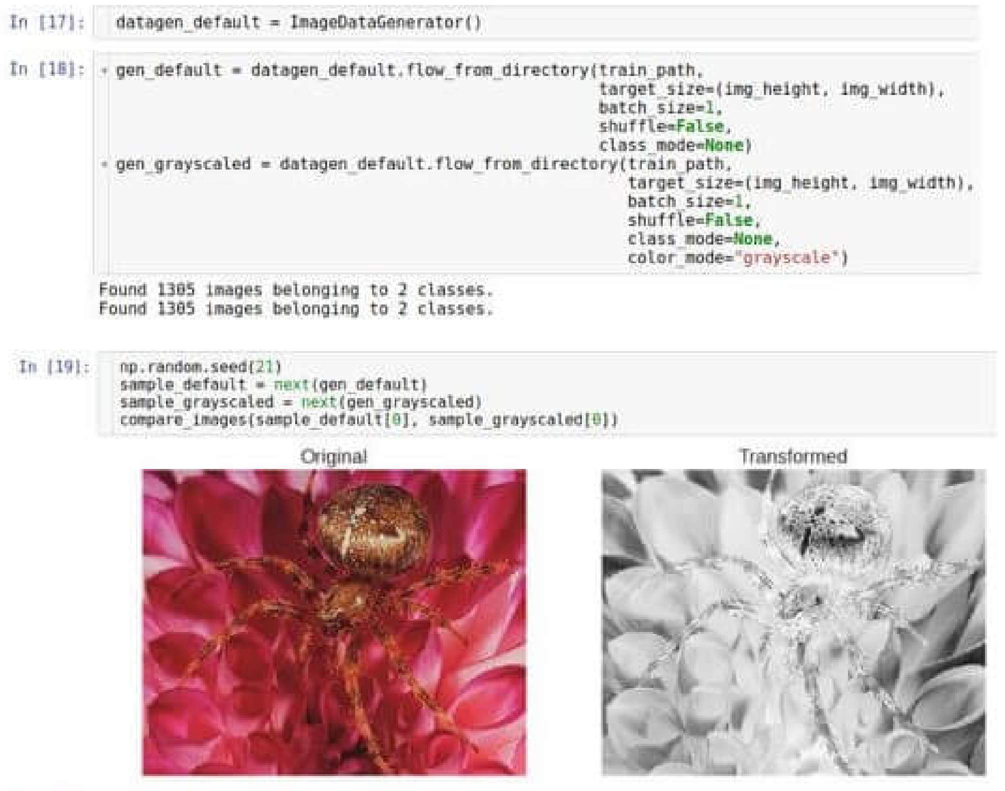
```
In [14]:  np.random.seed(21)
          sample_default = next(gen_default)
          sample_rescaled = next(gen_rescaled)
          compare_images(sample_default[0], sample_rescaled[0])
```

```
In [15]:  sample_default[0][:2, :2, 0]

Out[15]: array([[ 143.,  144.],
                [ 144.,  144.]], dtype=float32)

In [16]:  sample_rescaled[0][:2, :2, 0]

Out[16]: array([[ 0.56078434,  0.56470591],
                [ 0.56470591,  0.56470591]], dtype=float32)
```

automatically to the default range to be able to display them. Let’s take a look at the raw data, which are arrays. As one can see, raw arrays differ exactly by a factor of 255.

- ## Grayscaling

Another type of transformation that might be useful is *grayscaling*, which turns a color RGB image into images with only shades of gray representing colors. Conventional image processing might have used grayscaling in combination with consequent thresholding. This pair of transformations can throw away noisy pixels and detect shapes in the picture. Nowadays, all these operations are learned through convolutional neural networks (CNN), but grayscaling as a preprocessing step might still be useful. Let’s run that step in Keras with the same generator class.



```
In [17]: datagen_default = ImageDataGenerator()
```

```
In [18]: gen_default = datagen_default.flow_from_directory(train_path,
                                                           target_size=(img_height, img_width),
                                                           batch_size=1,
                                                           shuffle=False,
                                                           class_mode=None)
         gen_grayscaled = datagen_default.flow_from_directory(train_path,
                                                              target_size=(img_height, img_width),
                                                              batch_size=1,
                                                              shuffle=False,
                                                              class_mode=None,
                                                              color_mode="grayscale")
```

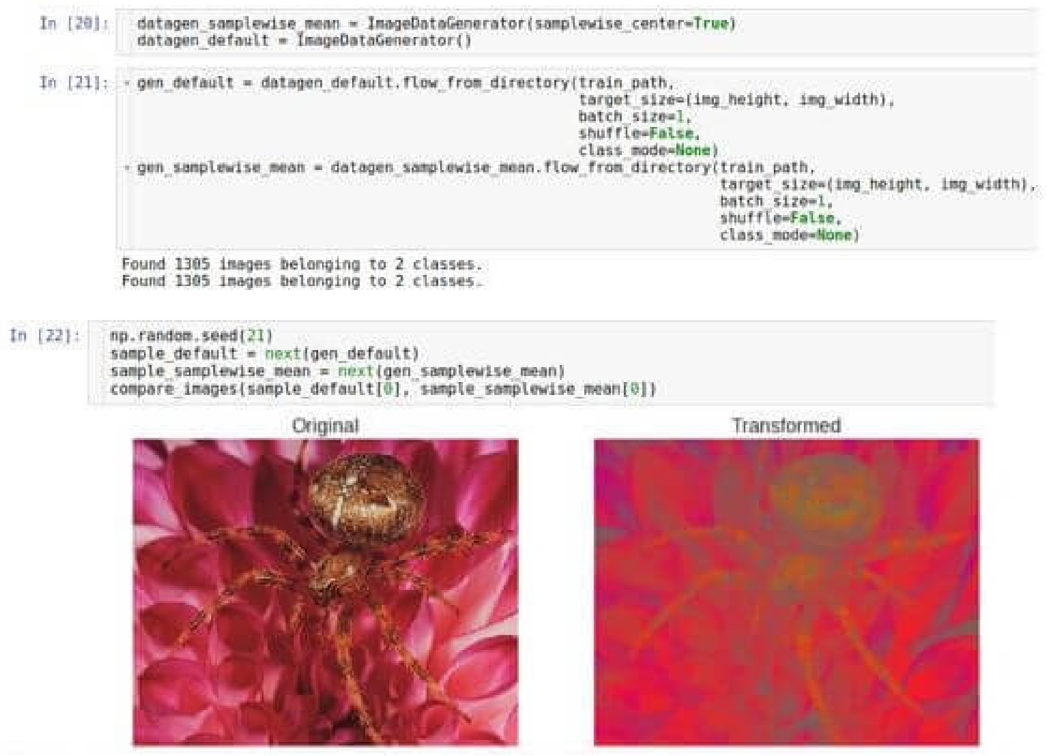
```
Found 1305 images belonging to 2 classes.
Found 1305 images belonging to 2 classes.
```

```
In [19]: np.random.seed(21)
         sample_default = next(gen_default)
         sample_grayscaled = next(gen_grayscaled)
         compare_images(sample_default[0], sample_grayscaled[0])
```

Here, we create only one instance of the class but two different generators are taken from it. The second one sets the color_mode option to “**grayscale**” (while the default value is “**RGB**”).

- ## Samplewise Centering

We’ve already seen that raw data values are from 0 to 255. So, one sample is a 3D array of numbers from 0 to 255. Following the optimization stability considerations (get rid of vanishing or saturating values problems) *we might want to normalize the dataset such that the mean value of each data sample would be equal to 0*.



```
In [20]:  datagen_samplewise_mean = ImageDataGenerator(samplewise_center=True)
          datagen_default = ImageDataGenerator()

In [21]:  gen_default = datagen_default.flow_from_directory(train_path,
                                                        target_size=(img_height, img_width),
                                                        batch_size=1,
                                                        shuffle=False,
                                                        class_mode=None)
          gen_samplewise_mean = datagen_samplewise_mean.flow_from_directory(train_path,
                                                        target_size=(img_height, img_width),
                                                        batch_size=1,
                                                        shuffle=False,
                                                        class_mode=None)

          Found 1305 images belonging to 2 classes.
          Found 1305 images belonging to 2 classes.

In [22]:  np.random.seed(21)
          sample_default = next(gen_default)
          sample_samplewise_mean = next(gen_samplewise_mean)
          compare_images(sample_default[0], sample_samplewise_mean[0])
```

For that purpose, we need to calculate the mean value across one whole sample and subtract it from each number in it.

In Keras it can be done through the “samplewise_center” option. The results are shown below.

- ## Samplewise std normalization

This preprocessing step follows the same idea as samplewise centering, but instead of setting the mean value to 0, it sets the standard deviation value to 1.

```
In [23]: datagen_samplewise_std = ImageDataGenerator(samplewise_std_normalization=True)
         datagen_default = ImageDataGenerator()
```

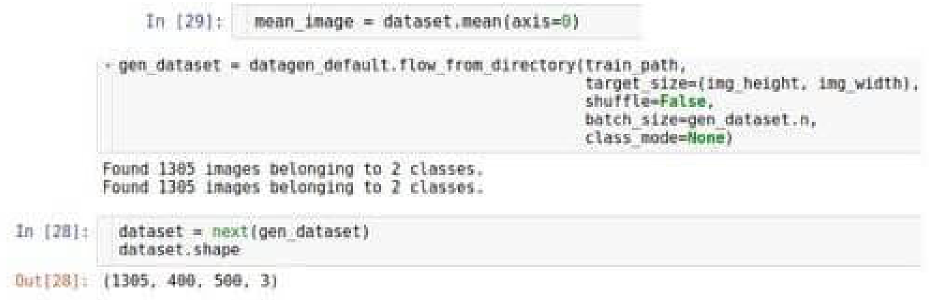
Std normalization is controlled by the option *samplewise_std_normalization*. It's worth mentioning that these two samplewise normalization options are often used simultaneously.

This transformation might be applied in deep learning models to improve the optimization stability by reducing the influence of the exploding gradients problem.

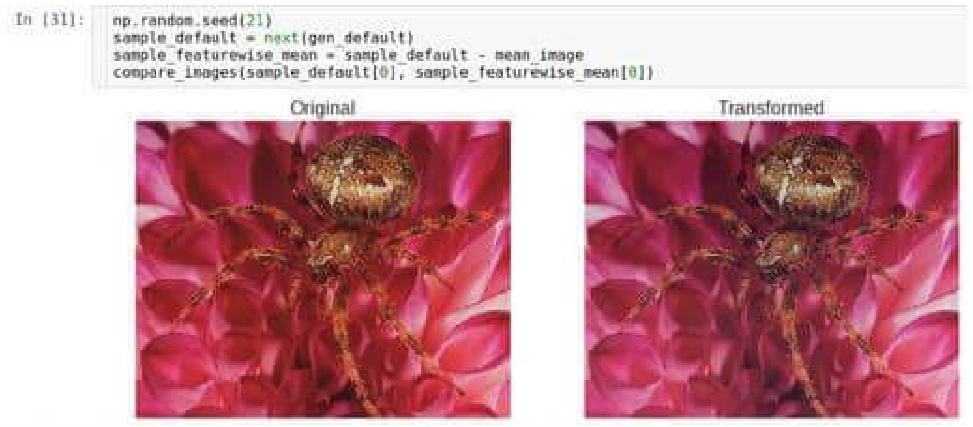
- ## Featurewise centering

In the previous two sections we used a normalization technique that looks at each particular sample at a time. There is an alternative approach to the normalization procedure. Let's treat each number in the image array as a feature. Then, *each image is represented by a vector of features*. There are plenty of such vectors in the dataset; therefore, we can treat them coming from some unknown *distribution*. This distribution will be multivariate, and the dimension of the space will be equal to the number of features, which is width * height * 3. Although we don't know the real distribution of the data we can try to normalize it by subtracting the mean value of the distribution. Note that here the mean value is the vector of the same dimension as space; that is, it is an image itself. In other words, we average across the dataset and not across one sample.

There is a special Keras option called *featurewise_centering*, but unfortunately as of August 2017 it had a bug in implementation; thus, let's implement it ourselves. First of all, read the whole dataset into memory (it's affordable because the dataset is small). We did it by setting the batch size to the size of the dataset. Now, let's calculate the mean image across the dataset and, finally, subtract it from the test image.



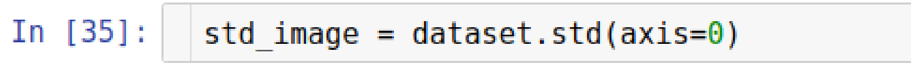
```
In [29]:  mean_image = dataset.mean(axis=0)

gen_dataset = datagen_default.flow_from_directory(train_path,
                                                  target_size=(img_height, img_width),
                                                  shuffle=False,
                                                  batch_size=gen_dataset.n,
                                                  class_mode=None)
Found 1305 images belonging to 2 classes.
Found 1305 images belonging to 2 classes.

In [28]:  dataset = next(gen_dataset)
          dataset.shape

Out[28]: (1305, 400, 500, 3)
```



```
In [31]:  np.random.seed(21)
          sample_default = next(gen_default)
          sample_featurewise_mean = sample_default - mean_image
          compare_images(sample_default[0], sample_featurewise_mean[0])
```

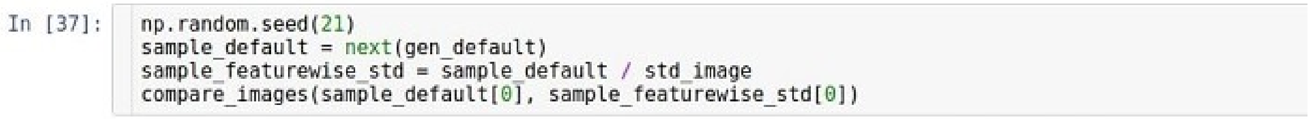
- ## Featurewise std normalization

The idea behind featurewise standard deviation normalization is exactly the same as behind centering. The only difference is that we divide by the sample standard deviation instead of subtracting the mean value. The result again does not differ a lot visually. The same thing



```
In [35]:  std_image = dataset.std(axis=0)
```

happened with rescaling, because featurewise std normalization is no more than rescaling but the normalization constant is adaptively calculated, while in rescaling one should specify it with hands. Note that the same idea of normalization across batches of data is the origin of the state-of-the-art deep learning technique called *BatchNormalization* (https://arxiv.org/pdf/1502.03167.pdf).



```
In [37]: np.random.seed(21)
         sample_default = next(gen_default)
         sample_featurewise_std = sample_default / std_image
         compare_images(sample_default[0], sample_featurewise_std[0])
```

Augmentation Transformations

In this section, we’re going to discuss more data-dependent transformations, which explicitly use the graphical nature of data. These kinds of transformations are often used for data augmentation procedures.

- ### Rotation

  This transformation rotates the image in a certain direction (clockwise or counterclockwise).

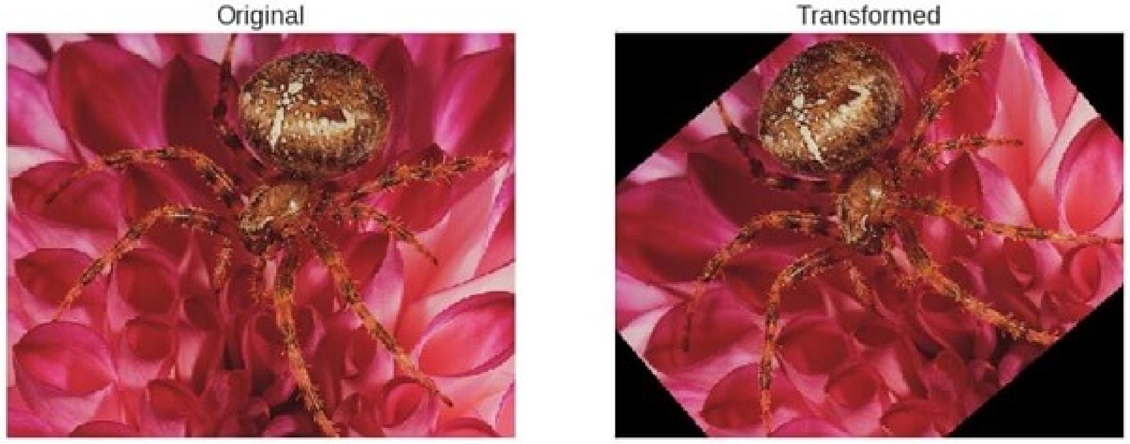
  The parameter that allows the rotations is called *rotation_range*. It specifies the range of rotations in degrees from which the random angle will be chosen uniformly to do a rotation. Note that during the rotation the size of the image remains the same. Thus, some of the image regions will be cropped out and some of the regions of the new image will need to be filled.

```
In [38]: datagen_rotated = ImageDataGenerator(rotation_range=45, fill_mode="constant")
         datagen_default = ImageDataGenerator()
```

```
In [39]: gen_default = datagen_default.flow_from_directory(train_path,
                                                           target_size=(img_height, img_width),
                                                           batch_size=1,
                                                           shuffle=False,
                                                           class_mode=None)
         gen_rotated = datagen_rotated.flow_from_directory(train_path,
                                                           target_size=(img_height, img_width),
                                                           batch_size=1,
                                                           shuffle=False,
                                                           class_mode=None)

         Found 1305 images belonging to 2 classes.
         Found 1305 images belonging to 2 classes.
```

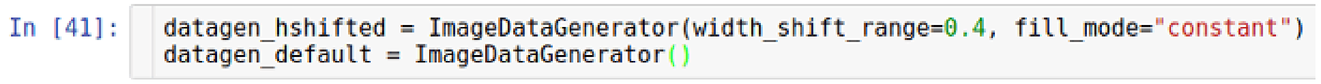
The filling mode can be set up by the *fill_mode* parameter. It supports a variety of different ways for filling, but here we use *constant* just for the sake of the example.

```
In [40]: np.random.seed(21)
         sample_default = next(gen_default)
         sample_rotated = next(gen_rotated)
         compare_images(sample_default[0], sample_rotated[0])
```



- ## Horizontal shift

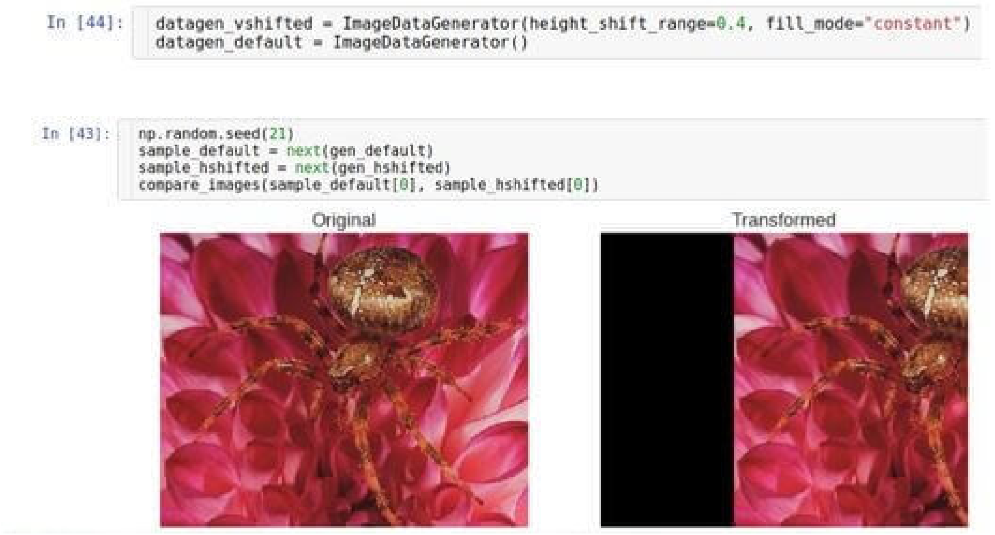
This transformation *shifts* the image to a certain direction along the horizontal axis (left or right).



```
In [41]: datagen_hshifted = ImageDataGenerator(width_shift_range=0.4, fill_mode="constant")
         datagen_default = ImageDataGenerator()
```

The size of the shift can be determined using the *width_shift_range* parameter and is measured as a fraction of the total width.

Vertical shift



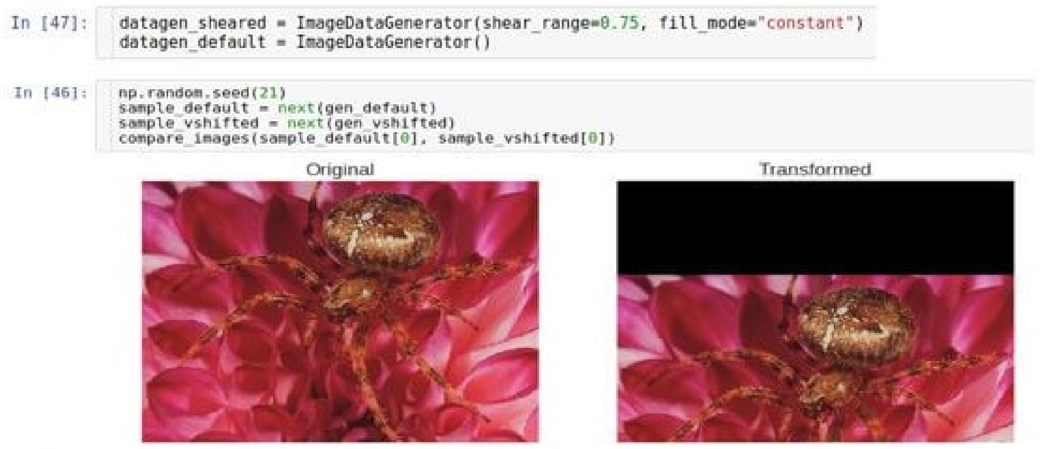
```
In [44]: datagen_vshifted = ImageDataGenerator(height_shift_range=0.4, fill_mode="constant")
         datagen_default = ImageDataGenerator()
```

```
In [43]: np.random.seed(21)
         sample_default = next(gen_default)
         sample_hshifted = next(gen_hshifted)
         compare_images(sample_default[0], sample_hshifted[0])
```

It *shifts* the image along the vertical axis (up or down). The parameter through which we can control the range of shift is called the *height_shift* generator, and is also measured as a fraction of total height.

- ## Shearing

[Shear mapping (https://en.wikipedia.org/wiki/Shear_mapping)](https://en.wikipedia.org/wiki/Shear_mapping) or *shearing* displaces each point in the vertical direction by an amount proportional to its distance from an edge of the image. Note that in general the direction does not have to be vertical and can be arbitrary.



```
In [47]: datagen_sheared = ImageDataGenerator(shear_range=0.75, fill_mode="constant")
         datagen_default = ImageDataGenerator()
```

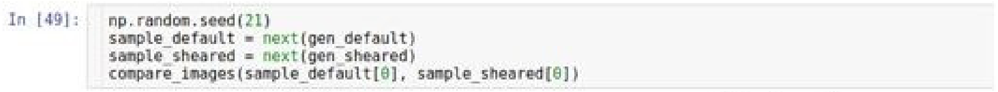
```
In [46]: np.random.seed(21)
         sample_default = next(gen_default)
         sample_vshifted = next(gen_vshifted)
         compare_images(sample_default[0], sample_vshifted[0])
```

The parameter that controls the displacement rate is called *shear_range* and corresponds to the deviation angle (in radians) between a horizontal line in the original picture and the image (in the mathematical sense) of this line in the transformed image.

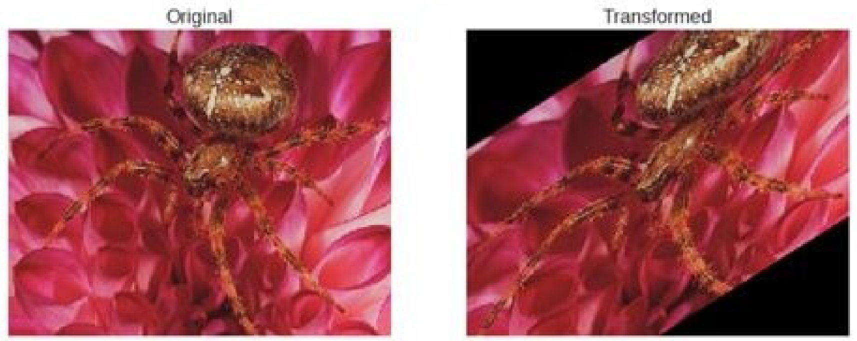
- ## Zoom

```
In [50]: datagen_zoomed = ImageDataGenerator(zoom_range=0.5, fill_mode="constant")
         datagen_default = ImageDataGenerator()
```

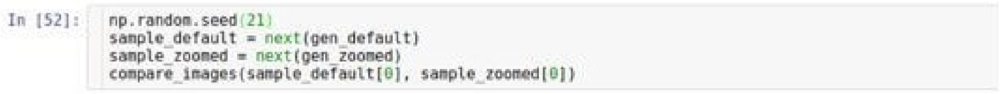
This transformation zooms the initial image in or out. The *zoom_range* parameter controls the zooming factor.



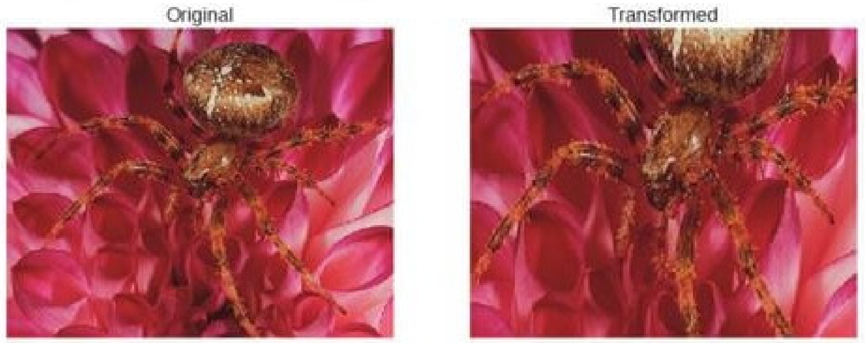
```
In [49]: np.random.seed(21)
         sample_default = next(gen_default)
         sample_sheared = next(gen_sheared)
         compare_images(sample_default[0], sample_sheared[0])
```



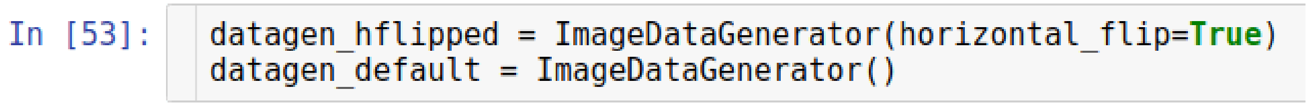
For example, *zoom_range* is equal to 0.5 means that the zooming factor will be chosen from the range [0.5, 1.5].



```
In [52]: np.random.seed(21)
         sample_default = next(gen_default)
         sample_zoomed = next(gen_zoomed)
         compare_images(sample_default[0], sample_zoomed[0])
```



- ## Horizontal flip



```
In [53]: datagen_hflipped = ImageDataGenerator(horizontal_flip=True)
         datagen_default = ImageDataGenerator()
```

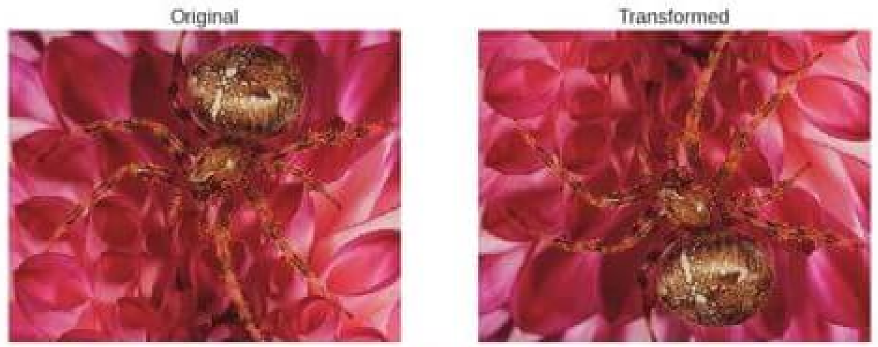
It flips the image with respect to the vertical axis. One can either turn it on or off using the *horizontal_flip* parameter.

- **Vertical flip**

```
In [55]: np.random.seed(21)
         sample_default = next(gen_default)
         sample_hflipped = next(gen_hflipped)
         compare_images(sample_default[0], sample_hflipped[0])
```

```
In [56]: datagen_vflipped = ImageDataGenerator(vertical_flip=True)
         datagen_default = ImageDataGenerator()
```

```
In [58]: np.random.seed(21)
         sample_default = next(gen_default)
         sample_vflipped = next(gen_vflipped)
         compare_images(sample_default[0], sample_vflipped[0])
```

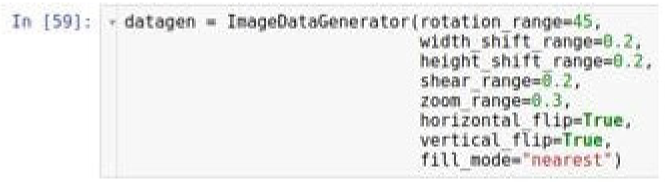


It flips the image with regard to the horizontal axis. The *vertical_flip* Boolean parameter controls the presence of this transformation.

Combination

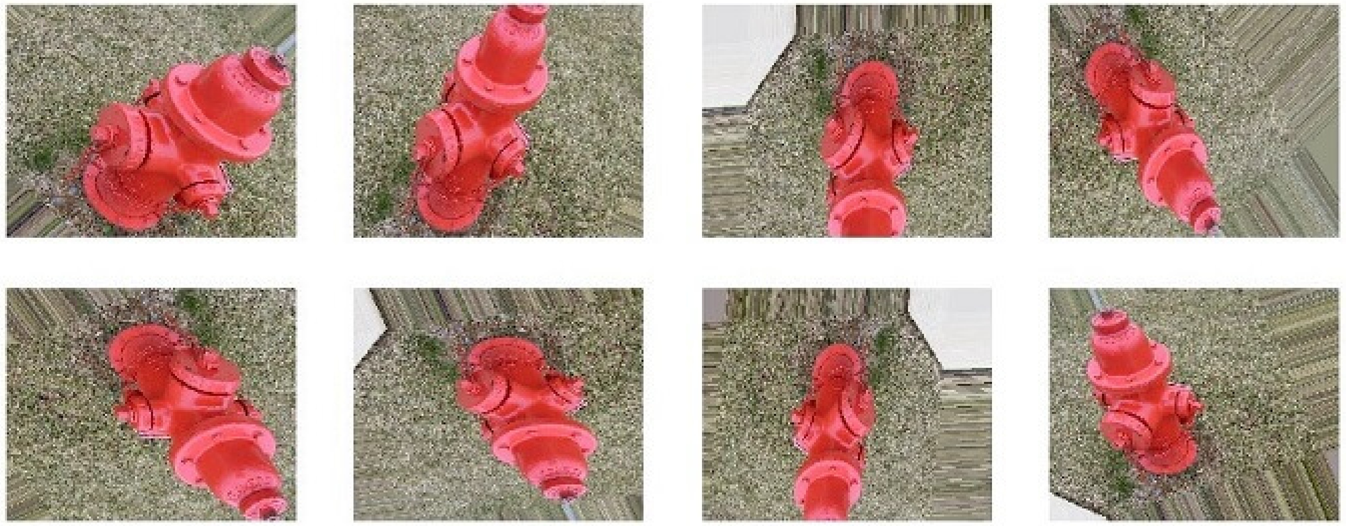
Let’s try to apply all the described augmentation transformations simultaneously and see what happens. Recall that the parameters of each of the transformations are chosen randomly from the specified range; thus, we should have a considerably diverse set of samples.

Let’s initialize our *ImageDataGenerator* with all the available options turned on and test it on an image of a red hydrant.



```
In [59]:  datagen = ImageDataGenerator(rotation_range=45,
                                      width_shift_range=0.2,
                                      height_shift_range=0.2,
                                      shear_range=0.2,
                                      zoom_range=0.3,
                                      horizontal_flip=True,
                                      vertical_flip=True,
                                      fill_mode="nearest")
```

Note that previously we used constant filling mode just for better visualization. Now, we’re going to use a more elaborate filling mode which is called *nearest*; this mode assigns the color of the closest existing pixel to the pixel that should be blank.



Conclusion

In this article, we described an overview of the common techniques of image preprocessing such as scaling, normalization, rotation, shifting, and shearing. We also demonstrated how these transformations can be implemented with Keras, and plugged in to the deep learning pipeline both technically (*ImageDataGenerator* class) and ideologically (data augmentation).

In the next article, we're going to apply these techniques to build a baseline CNN model for image sentiment analysis.

Browse Hands-On AI Tutorials: Part 1: Image Data and Modeling ›

Find more helpful resources at the Intel® Nervana™ AI Academy. (/en-us/ai)

For more complete information about compiler optimizations, see our Optimization Notice (/en-us/articles/optimization-notice#opt-en).

- Technology
  - Intel® Technology
  - Intel® Hardware
  - Intel® AI DevCloud
  - Intel® Movidius™ Neural Compute Stick
- Connect
  - Follow @IntelSoftware
  - Developer Mesh Projects
  - Forum
- Sign Up
  - Developer Journey Series
  - AI Student Ambassador
  - University Club Sponsorship
  - Newsletter

Get the Newsletter

Follow us:

Terms of Use *Trademarks Privacy Cookies Email preferences