Image Augmentation on the fly using Keras ImageDataGenerator!

ADVANCED COMPUTER VISION DEEP LEARNING IMAGE PYTHON TECHNIQUE UNSTRUCTURED DATA

Overview

- Understand image augmentation
- Learn Image Augmentation using Keras ImageDataGenerator

Introduction

When working with deep learning models, I have often found myself in a peculiar situation when there is not much data to train my model. It was in times like these when I came across the concept of image augmentation.

The image augmentation technique is a great way to expand the size of your dataset. You can come up with new transformed images from your original dataset. But many people use the conservative way of augmenting the images i.e. augmenting images and storing them in a numpy array or in a folder. I have got to admit, I used to do this until I stumbled upon the *ImageDataGenerator* class.

Keras *ImageDataGenerator* is a gem! It lets you augment your images in real-time while your model is still training! You can apply any random transformations on each training image as it is passed to the model. This will not only make your model robust but will also save up on the overhead memory! Now let's dive deeper and check out the different ways in which this class is so great for image augmentation.

I am assuming that you are already familiar with neural networks. If not, I suggest going through the following resources first:

- *Introduction to Neural Networks*
- *Understanding and coding Neural Networks From Scratch in Python and R*

Table of Contents

- Keras Fit_generator Method
- Model building with Keras ImageDataGenerator

Image augmentation – A refresher

Image augmentation is a technique of applying different transformations to original images which results in multiple transformed copies of the same image. Each copy, however, is different from the other in certain aspects depending on the augmentation techniques you apply like shifting, rotating, flipping, etc.

Applying these small amounts of variations on the original image does not change its target class but only provides a new perspective of capturing the object in real life. And so, we use it is quite often for building deep learning models.

These image augmentation techniques not only expand the size of your dataset but also incorporate a level of variation in the dataset which allows your model to generalize better on unseen data. Also, the model becomes more robust when it is trained on new, slightly altered images.

So, with just a few lines of code, you can instantly create a large corpus of similar images without having to worry about collecting new images, which is not feasible in a real-world scenario. Now, let's see how it's done with the good old Keras library!

Image augmentation in Keras

Keras **ImageDataGenerator** class provides a quick and easy way to augment your images. It provides a host of different augmentation techniques like standardization, rotation, shifts, flips, brightness change, and many more. You can find more on its official documentation page.

However, the main benefit of using the Keras ImageDataGenerator class is that it is designed to provide real-time data augmentation. Meaning it is generating augmented images on the fly while your model is still in the training stage. How cool is that!

ImageDataGenerator class ensures that the model receives new variations of the images at each epoch. But it only returns the transformed images and does not add it to the original corpus of images. If it was, in fact, the case, then the model would be seeing the original images multiple times which would definitely overfit our model.

Another advantage of ImageDataGenerator is that it requires lower memory usage. This is so because without using this class, we load all the images at once. But on using it, we are loading the images in batches which saves a lot of memory.

Now let's have a look at a few augmentation techniques with Keras ImageDataGenerator class.

Augmentation techniques with Keras ImageDataGenerator class

1. Random Rotations

Image rotation is one of the widely used augmentation techniques and allows the model to become invariant to the orientation of the object.

ImageDataGenerator class allows you to randomly rotate images through any degree between 0 and 360 by providing an integer value in the **rotation_range** argument.

When the image is rotated, some pixels will move outside the image and leave an empty area that needs to be filled in. You can fill this in different ways like a constant value or nearest pixel values, etc. This is specified in the **fill_mode** argument and the default value is **"nearest"** which simply replaces the empty area with the nearest pixel values.

```
# ImageDataGenerator rotation
datagen = ImageDataGenerator(rotation_range=30, fill_mode='nearest')

# iterator
aug_iter = datagen.flow(img, batch_size=1)

# generate samples and plot
fig, ax = plt.subplots(nrows=1, ncols=3, figsize=(15,15))

# generate batch of images
for i in range(3):

        # convert to unsigned integers
        image = next(aug_iter)[0].astype('uint8')

        # plot image
        ax[i].imshow(image)
        ax[i].axis('off')
```

view raw

rotation.py hosted with ❤ by GitHub

2. Random Shifts

It may happen that the object may not always be in the center of the image. To overcome this problem we can shift the pixels of the image either horizontally or vertically; this is done by adding a certain constant value to all the pixels.

ImageDataGenerator class has the argument **height_shift_range** for a vertical shift of image and **width_shift_range** for a horizontal shift of image. If the value is a float number, that would indicate the percentage of width or height of the image to shift. Otherwise, if it is an integer value then simply the width or height are shifted by those many pixel values.

```
# ImageDataGenerator shifting
```

```
datagen = ImageDataGenerator(width_shift_range=0.2, height_shift_range=0.2)

# iterator
aug_iter = datagen.flow(img, batch_size=1)

# generate samples and plot
fig, ax = plt.subplots(nrows=1, ncols=3, figsize=(15,15))

# generate batch of images
for i in range(3):

        # convert to unsigned integers
        image = next(aug_iter)[0].astype('uint8')

        # plot image
        ax[i].imshow(image)
        ax[i].axis('off')
```

view raw

shift.py hosted with ❤ by **GitHub**



3. Random Flips

Flipping images is also a great augmentation technique and it makes sense to use it with a lot of different objects.

ImageDataGenerator class has parameters **horizontal_flip** and **vertical_flip** for flipping along the vertical or the horizontal axis. However, this technique should be according to the object in the image. For example, vertical flipping of a car would not be a sensible thing compared to doing it for a symmetrical object like football or something else. Having said that, I am going to flip my image in both ways just to demonstrate the effect of the augmentation.

```
# ImageDataGenerator flipping
datagen = ImageDataGenerator(horizontal_flip=True, vertical_flip=True)

# iterator
aug_iter = datagen.flow(img, batch_size=1)

# generate samples and plot
fig, ax = plt.subplots(nrows=1, ncols=3, figsize=(15,15))

# generate batch of images
for i in range(3):

        # convert to unsigned integers
        image = next(aug_iter)[0].astype('uint8')

        # plot image
        ax[i].imshow(image)
        ax[i].axis('off')
```

view raw

flip.py hosted with ❤ by **GitHub**



4. Random Brightness

It randomly changes the brightness of the image. It is also a very useful augmentation technique because most of the time our object will not be under perfect lighting condition. So, it becomes imperative to train our model on images under different lighting conditions.

Brightness can be controlled in the ImageDataGenrator class through the **brightness_range** argument. It accepts a list of two float values and picks a brightness shift value from that range. Values less than 1.0 darkens the image, whereas values above 1.0 brighten the image.

```
# ImageDataGenerator brightness
datagen = ImageDataGenerator(brightness_range=[0.4,1.5])

# iterator
aug_iter = datagen.flow(img, batch_size=1)

# generate samples and plot
fig, ax = plt.subplots(nrows=1, ncols=3, figsize=(15,15))

# generate batch of images
for i in range(3):

        # convert to unsigned integers
        image = next(aug_iter)[0].astype('uint8')

        # plot image
        ax[i].imshow(image)
        ax[i].axis('off')
```

view raw

brightness.py hosted with ❤ by **GitHub**



5. Random Zoom

The zoom augmentation either randomly zooms in on the image or zooms out of the image.

ImageDataGenerator class takes in a float value for zooming in the **zoom_range** argument. You could provide a list with two values specifying the lower and the upper limit. Else, if you specify a float value, then zoom will be done in the range [1-zoom_range,1+zoom_range].

Any value smaller than 1 will zoom in on the image. Whereas any value greater than 1 will zoom out on the image.

```
# ImageDataGenerator zoom
datagen = ImageDataGenerator(zoom_range=0.3

# iterator
aug_iter = datagen.flow(img, batch_size=1)

# generate samples and plot
fig, ax = plt.subplots(nrows=1, ncols=3, figsize=(15,15))

# generate batch of images
for i in range(3):

        # convert to unsigned integers
        image = next(aug_iter)[0].astype('uint8')

        # plot image
        ax[i].imshow(image)
        ax[i].axis('off')
```

view raw

zoom.py hosted with ❤ by GitHub

There are many more augmentation techniques that I have not covered in this article but I encourage you to check them out in the official documentation.

Introducing the Dataset

Before we explore the methods of the ImageDataGenerator class, let's first get familiar with the dataset we will be working with.

We have a dataset of emergency (like fire trucks, ambulances, police vehicles, etc.) and non-emergency vehicles. There are a total of 1646 unique images in the dataset. Since these are not a lot of images to create a robust neural network, it will act as a great dataset to test the potential of the ImageDataGenerator class!

You can download the dataset from here.

ImageDataGenerator methods

So far we have seen how to augment images using **ImageDataGenerator().flow()** method. However, there are a few methods in the same class which are actually quite helpful and which implement augmentation on the fly.

Let's first come up with the augmentations we would want to apply to the images.

```
# ImageDataGenerator
datagen = ImageDataGenerator(
        rotation_range=10, # rotation
```

```
        width_shift_range=0.2, # horizontal shift
        height_shift_range=0.2, # vertical shift
        zoom_range=0.2, # zoom
        horizontal_flip=True, # horizontal flip
        brightness_range=[0.2,1.2]) # brightness
```

view raw

ImageDataGenerator.py hosted with ❤ by GitHub

1. Flow_from_directory

The **flow_from_directory()** method allows you to read the images directly from the directory and augment them while the neural network model is learning on the training data.

The method expects that images belonging to different classes are present in different folders but are inside the same parent folder. So let's create that first using the following code:

```
import pandas as pd
import os
import shutil
from sklearn.model_selection import train_test_split

# Home directory
home_path = r'C:/Users/Dell/Desktop/Analytics Vidhya/ImageDataGenerator/emergency_vs_non-emergency_dataset/emergency_vs_non-emer

# Create train and validation directories
train_path = os.path.join(home_path,'train')
os.mkdir(train_path)
val_path = os.path.join(home_path,'valid')
os.mkdir(val_path)

# Create sub-directories
emergency_train_path = os.path.join(home_path + r'/train','emergency')
os.mkdir(emergency_train_path)

non_emergency_train_path = os.path.join(home_path + r'/train','non_emergency')
os.mkdir(non_emergency_train_path)

emergency_val_path = os.path.join(home_path + r'/valid','emergency')
os.mkdir(emergency_val_path)

non_emergency_val_path = os.path.join(home_path + r'/valid','non_emergency')
os.mkdir(non_emergency_val_path)

# Original df
df = pd.read_csv(home_path + r'/emergency_train.csv')

# Images and Labels
X = df.loc[:,'image_names']
y = df.loc[:,'emergency_or_not']

# Train-Test splitfor train and validation images
train_x, val_x, train_y, val_y = train_test_split(X, y, test_size = 0.1, random_state = 27, stratify=y)

# Train df
df_train = pd.DataFrame(columns=['image_names','emergency_or_not'])
df_train['image_names'] = train_x
df_train['emergency_or_not'] = train_y

# Validation df
df_valid= pd.DataFrame(columns=['image_names','emergency_or_not'])
df_valid['image_names'] = val_x
df_valid['emergency_or_not'] = val_y

# Reset index
df_train.reset_index(drop=True, inplace=True)
df_valid.reset_index(drop=True, inplace=True)

# Save train images
for i in range(len(df_train)):

    image = df_train.loc[i,'image_names']
```

```
    if df_train.loc[i,'emergency_or_not'] == 0:
        shutil.copy(home_path + r'/images/' + image, non_emergency_train)
    else:
        shutil.copy(home_path + r'/images/' + image, emergency_train)

# Save validation images
for i in range(len(df_valid)):

    image = df_valid.loc[i,'image_names']

    if df_valid.loc[i,'emergency_or_not'] == 0:
        shutil.copy(home_path + r'/images/' + image, non_emergency_val)
    else:
        shutil.copy(home_path + r'/images/' + image, emergency_val)
```

view raw

folder_structure.py hosted with ❤ by GitHub

Now, let's try to augment the images using the class method.

```
# ImageDataGenerator flow_from_directory

train_generator = datagen.flow_from_directory(
                  directory=home_path + r'/train/',
                  target_size=(400, 400), # resize to this size
                  color_mode="rgb", # for coloured images
                  batch_size=1, # number of images to extract from folder for every batch
                  class_mode="binary", # classes to predict
                  seed=2020 # to make the result reproducible
                  )

fig, ax = plt.subplots(nrows=1, ncols=4, figsize=(15,15))

for i in range(4):

  # convert to unsigned integers for plotting
  image = next(train_generator)[0].astype('uint8')

  # changing size from (1, 200, 200, 3) to (200, 200, 3) for plotting the image
  image = np.squeeze(image)

  # plot raw pixel data
  ax[i].imshow(image)
  ax[i].axis('off')
```

view raw

flow_from_directory.py hosted with ❤ by GitHub

```
Found 1481 images belonging to 2 classes.
```

The following are few important parameters of this method:

1. **directory**: this is the path to the parent folder which contains the subfolder for the different class images.
2. **target_size**: Size of the input image.
3. **color_mode**: Set to **rgb** for colored images otherwise **grayscale** if the images are black and white.
4. **batch_size**: Size of the batches of data.
5. **class_mode**: Set to **binary** is for 1-D binary labels whereas **categorical** is for 2-D one-hot encoded labels.

6. **seed**: Set to reproduce the result.

2. Flow_from_dataframe

The **flow_from_dataframe()** is another great method in the ImageDataGenerator class that allows you to directly augment images by reading its name and target value from a dataframe.

This comes very handily when you have all the images stored within the same folder.

```
# ImageDataGenerator flow_from_dataframe

df_train = pd.read_csv(home_path + r'/emergency_train.csv')
df_train['emergency_or_not'] = df_train['emergency_or_not'].astype('str') # requires target in string format

train_generator_df = datagen.flow_from_dataframe(dataframe=df_train,
                                                 directory=home_path+'/images/',
                                                 x_col="image_names",
                                                 y_col="emergency_or_not",
                                                 class_mode="binary",
                                                 target_size=(200, 200),
                                                 batch_size=1,
                                                 rescale=1.0/255,
                                                 seed=2020)
# plotting images
fig, ax = plt.subplots(nrows=1, ncols=4, figsize=(15,15))

for i in range(4):

  # convert to unsigned integers for plotting
  image = next(train_generator_df)[0].astype('uint8')

  # changing size from (1, 200, 200, 3) to (200, 200, 3) for plotting the image
  image = np.squeeze(image)

  # plot raw pixel data
  ax[i].imshow(image)
  ax[i].axis('off')
```

view raw

flow_from_dataframe.py hosted with ❤ by **GitHub**

```
Found 1646 validated image filenames belonging to 2 classes.
```

This method also has a few parameters that need to be explained in brief:

1. **dataframe**: The Pandas DataFrame that contains the image names and target values.
2. **directory**: The path to the folder that contains all the images.
3. **x_col**: The column name in the DataFrame that has the image names.
4. **y_col**: The column name in the DataFrame that has the target values.
5. **class_mode**: Set to **binary** is for 1-D binary labels whereas **categorical** is for 2-D one-hot encoded labels.

6. **target_size**: Size of input images.
7. **batch_size**: Size of the batches of data.
8. **seed**: Set to reproduce the result.

Keras Fit_generator Method

Right, you have created the iterators for augmenting the images. But how do you feed it to the neural network so that it can augment on the fly?

For that, all you need to do is feed the iterator as an input to the Keras **fit_generator()** method applied on the neural network model along with epochs, batch_size, and other important arguments. We will be using a Convolutional Neural Network(CNN) model. The fit_generator() method fits the model on data that is yielded batch-wise by a Python generator.

You can use either of the iterator methods mentioned above as input to the model.

```
# Augmenting on the fly with fit_generator()

# Directly use .flow()
model.fit_generator(datagen.flow(x_train, y_train, batch_size=batch_size),
                    epochs=epochs, # one forward/backward pass of training data
                    steps_per_epoch=x_train.shape[0]//batch_size, # number of images comprising of one epoch
                    validation_data=(x_test, y_test), # data for validation
                    validation_steps=x_test.shape[0]//batch_size)

# or use iterator from .flow_from_directory()
model.fit_generator(train_generator,
                    epochs=epochs,  # one forward/backward pass of training data
                    steps_per_epoch=x_train.shape[0]//batch_size,  # number of images comprising of one epoch
                    validation_data=(x_test, y_test), # Or validation_data=valid_generator
                    validation_steps=x_test.shape[0]//batch_size)

# or use iterator from .flow_from_datafram()
model.fit_generator(train_generator_df,
                    epochs=epochs,  # one forward/backward pass of training data
                    steps_per_epoch=x_train.shape[0]//batch_size,  # number of images comprising of one epoch
                    validation_data=(x_test, y_test),  # Or validation_data=valid_generator
                    validation_steps=x_test.shape[0]//batch_size)
```

view raw

model_fit_generator_methods.py hosted with ❤ by **GitHub**

Let's take a moment to understand the arguments of the **fit_generator()** method first before we start building our model.

- The first argument is the iterator for the train images that we get from the flow() or flow_from_dataframe() or flow_from_directory() method.
- **Epochs** are the number of forward/backward passes of the training data.
- **Steps_per_epoch** is an important argument. It specifies the number of batches of images that are in a single epoch. It is usually taken as the length of the original dataset divided by the batch size.
- **Validation_data** takes the validation dataset or the validation generator output from the generator method.
- **Validation_steps** is similar to steps_per_epoch, but for validation data. *This can be used when you are augmenting the validation set images as well.*

Note that this method might be removed in a future version of Keras. The Keras **fit()** method now supports generators and so we will be using the same to train our model.

Model Building with Keras ImageDataGenerator

Now that we have discussed the various methods of ImageDataGenerator class, it is time to build our own CNN model and see how well the class performs. We will compare the performance of the model both, with and without augmentation to get an idea of how helpful augmentation is.

Let's first import the relevant libraries.

```
from keras.models import Sequential
from keras.layers import Dropout, Flatten, Dense, Conv2D, MaxPooling2D, BatchNormalization
from keras.optimizers import SGD
import keras
from sklearn.model_selection import train_test_split
import cv2
```

view raw

load_pretrained_model.py hosted with ❤ by **GitHub**

Now let's prepare the dataset for the model. Here I split the original batch of images into train and validation parts. 90% will be used for training and 10% will be used for validation.

```
# Split data into train-test data sets

X = df.loc[:,'image_names']
y = df.loc[:,'emergency_or_not']

# Split
train_x, val_x, train_y, val_y = train_test_split(X, y,
                                                  test_size = 0.1,
                                                  random_state = 27,
                                                  stratify=y)

# Train df
df_train = pd.DataFrame(columns=['image_name','category'])
df_train['image_name'] = train_x
df_train['category'] = train_y

# Test df
df_test= pd.DataFrame(columns=['image_name','category'])
df_test['image_name'] = val_x
df_test['category'] = val_y

df_train.reset_index(drop=True, inplace=True)
df_test.reset_index(drop=True, inplace=True)
```

view raw

train_test_split.py hosted with ❤ by **GitHub**

Then we can append them into a list and prepare them to be input to the model.

```
# Images
train_images = df_train.loc[:,'image_name']
train_labels = df_train.loc[:,'category']

test_images = df_test.loc[:,'image_name']
test_labels = df_test.loc[:,'category']

# Train images
x_train = []
for i in train_images:
    image = home_path+'/images/'+i
    img = cv2.imread(image)
    x_train.append(img)

# Train labels
y_train=keras.utils.to_categorical(train_labels)

# Test images
x_test = []
for i in test_images:
    image = home_path+'/images/'+i
    img = cv2.imread(image)
    x_test.append(img)
```

```python
# Test labels
y_test=keras.utils.to_categorical(test_labels)

# Normalize images
x_train = np.array(x_train, dtype="float") / 255.0
x_test = np.array(x_test, dtype="float") / 255.0
```

view raw

train_test_images.py hosted with ❤ by GitHub

Let's create the architecture for our CNN model. The architecture is simple. It has three Convolutional layers and two fully connected layers.

```python
# Model architechture
model = Sequential()

model.add(Conv2D(32, (3, 3), input_shape=(224, 224, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(32, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Flatten())
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.24))
model.add(Dense(2,activation='softmax'))

model.summary()
```

view raw

model_building.py hosted with ❤ by GitHub

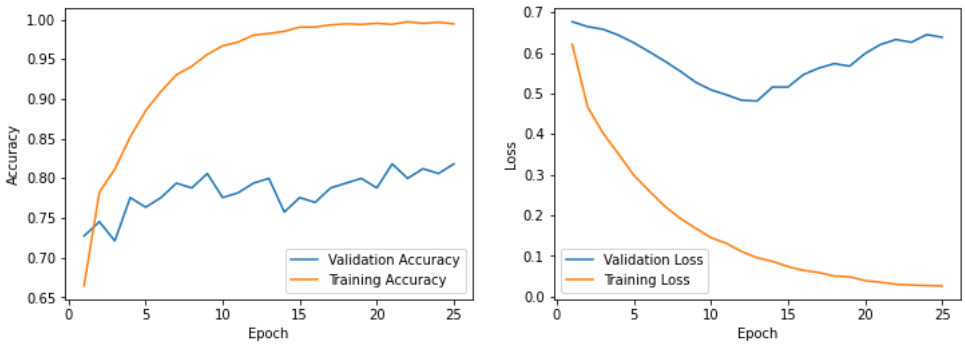
Now that we have created the architecture for our model, we can compile it and start training it.

```python
# Compile
optim = RMSprop(learning_rate=0.00001)
model.compile(loss='categorical_crossentropy',
              optimizer=optim,
              metrics=['accuracy'])
# Fit
history = model.fit(x_train,y_train,
                    epochs=25,
                    validation_data=(x_test,y_test),
                    batch_size=32,
                    verbose=1)
```

view raw

non_augment_model.py hosted with ❤ by GitHub

Here is the result I got after training the model for 25 epochs without augmenting the images.



You can probably notice overfitting happening here. Let's see if we can alleviate this using augmentation.

I am going to use the **flow()** method to augment the images on the fly. You can use other methods discussed in the previous section. We will be applying the following augmentation techniques to the training images.

```
# Augmentation
train_datagen = ImageDataGenerator(rotation_range=5,  # rotation
                                   width_shift_range=0.2,  # horizontal shift
                                   zoom_range=0.2,  # zoom
                                   horizontal_flip=True,  # horizontal flip
                                   brightness_range=[0.2,0.8])  # brightness
```

view raw

model_augment_flow.py hosted with ❤ by GitHub

Train the Model

Finally, let's train our model and see if the augmentations had any positive impact on the result!

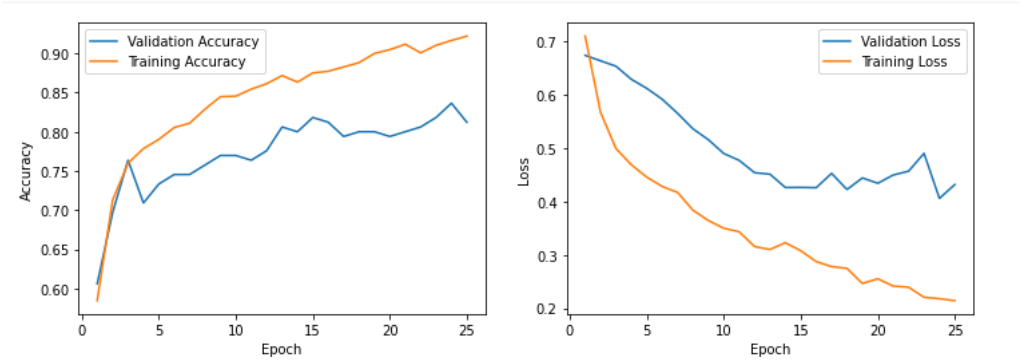
```
# Epochs
epochs = 25
# Batch size
batch_size = 32

history = model.fit(train_datagen.flow(x_train,y_train,
                                       batch_size=batch_size,
                                       seed=27,
                                       shuffle=False),
                    epochs=epochs,
                    steps_per_epoch=x_train.shape[0] // batch_size,
                    validation_data=(x_test,y_test),
                    verbose=1)
```

view raw

model_fit_generator.py hosted with ❤ by GitHub

After 25 epochs we get the following loss and accuracy for the model on the augmented data.



As you can notice here, the training and validation loss are both decreasing here with little divergence as compared to the outcome from the previous model. Also, notice how the training and validation accuracy is increasing together. They are comparatively closer than before the augmentation.

Such is the power of augmentation that our model is able to generalize on the images now!

I urge you to experiment around with this dataset yourself and see if you can improve the accuracy of the model!

You can learn how to build CNN models in detail in this awesome article.

End Notes

And just like that, you have learned to augment images in the easiest and quickest way possible. To summarize, in this article we learned how to avoid the conventional image augmentation technique by using the Keras ImageDataGenerator. Now forget the old school way of augmenting your images and saving them in a separate folder. You now know how to augment images on the fly!

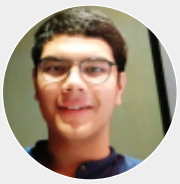
If you are looking to learn Image augmentation using PyTorch, I recommend going through this in-depth article.

Going further, if you are interested in learning more about deep learning and computer vision, I recommend you check out the following awesome courses curated by our team at Analytics Vidhya:

- Fundamentals of Deep Learning
- Computer Vision using Deep Learning 2.0

You can apply many more augmentation techniques than the ones discussed here that suit your image dataset and feel free to share your insights in the comments below.

Article Url - https://www.analyticsvidhya.com/blog/2020/08/image-augmentation-on-the-fly-using-keras-imagedatagenerator/



Aniruddha Bhandari

I am on a journey to becoming a data scientist. I love to unravel trends in data, visualize it and predict the future with ML algorithms! But the most satisfying part of this journey is sharing my learnings, from the challenges that I face, with the community to make the world a better place!