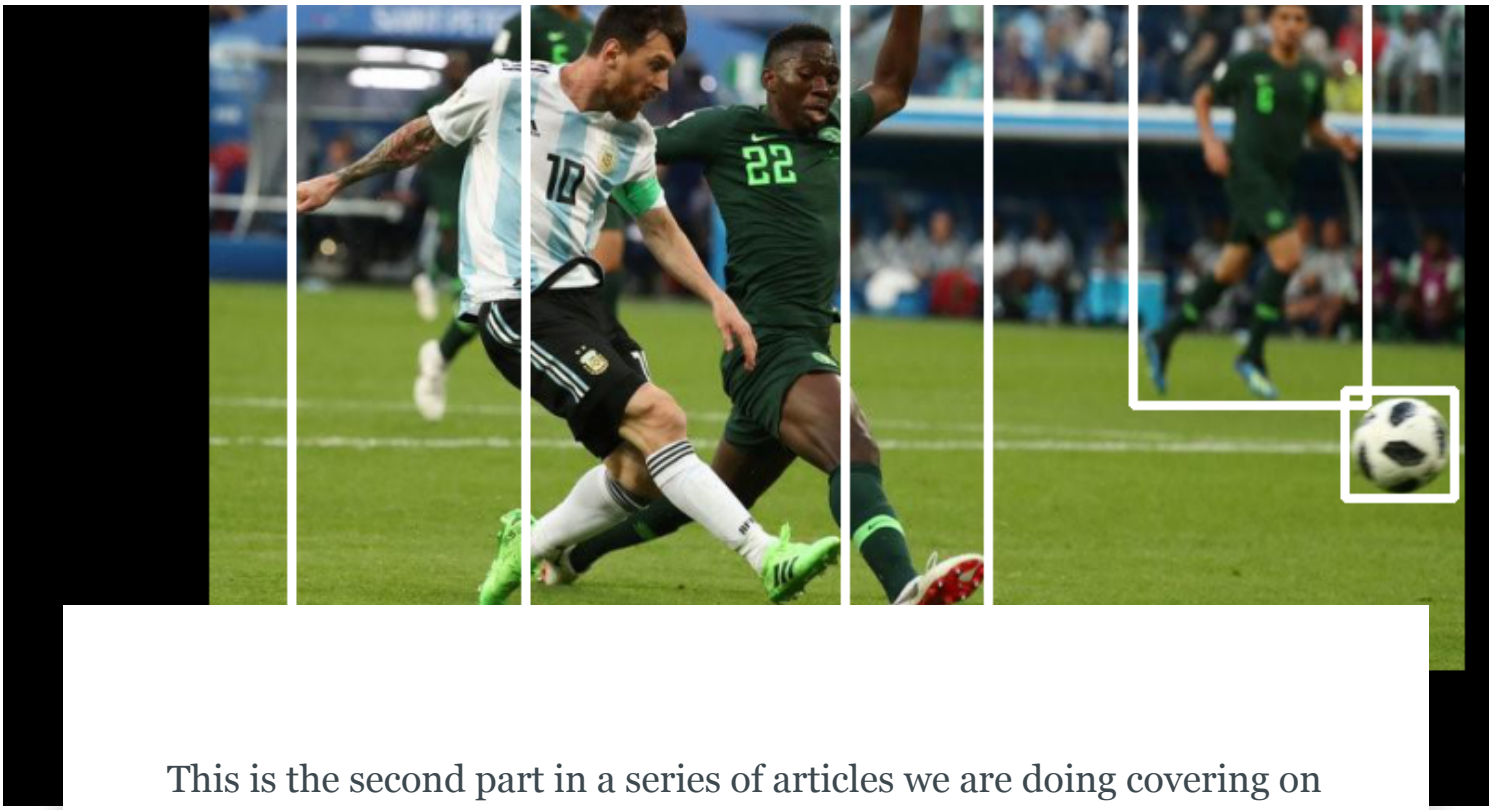
12 SEPTEMBER 2018 / SERIES: DATA AUGMENTATION

Data Augmentation for Bounding Boxes: Scaling and Translation



This is the second part in a series of articles we are doing covering on implementing adapting the image augmentation techniques for object detection tasks. In this part, we will cover how to implement the Scale and Translate augmentation techniques, and what to do if a portion of your bounding box is outside the image after the augmentation.

In the last part, we covered a uniform way to implement a augmentation as well as the HorizontalFlip augmentation.

GitHub Repo

Everything from this article and the entire augmentation library can be found in the following Github Repo.

https://github.com/Paperspace/DataAugmentationForObjectDetection

Documentation

The documentation for this project can be found by opening the `docs/build/html/index.html` in your browser or at this link.

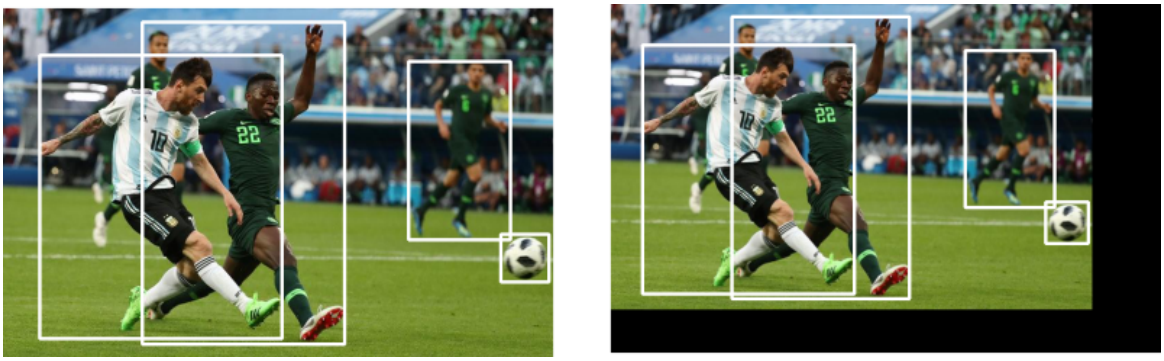
Part 1 of the series is a pre-requisite to this article and it's highly recommended you go through it.

This series has 4 parts.
1. Part 1: Basic Design and Horizontal Flipping
2. Part 2: Scaling and Translation
3. Part 3: Rotation and Shearing
4. Part 4: Baking augmentation into input pipelines

In this post, we will implement a couple of augmentations called **Scale** and **Translate**, which obviously do what they mean.

Scale

The result of a Scale translation would look like.



Left: Original Image Right: Scale Augmentation applied.

Design Decisions

- The very first thing we need to think about is the parameters of the Scale augmentation. The obvious choice is to ask by what **factor of the original image's dimension**, we want it to scale the our image. Therefore, **it must be a value greater than -1**. You can cannot scaling less than it's own dimensions.
- One can chose to maintain the aspect ratio by constraining the scaling factor to be the same for both height and width. However, we can allow the scale factors to differ which not only produces a scaling augmentation but also changes the aspect ratio of the images. We introduce a boolean variable `diff` that can turn on/off this functionality.
- While implementing a Stochastic version of this augmentation, we need a sample a scale factor randomly from a interval. The way we deal with it is, that if the user can provide the range from which the scaling factor `scale` will be sampled. If user provides only one float `scale`, it must be positive, and the scaling factor is sampled from (-`scale`, `scale`).

Let us now define the `__init__` method.

```
class RandomScale(object):
    """Randomly scales an image

    Bounding boxes which have an area of less than 25% in the rema
    transformed image is dropped. The resolution is maintained, an
    area if any is filled by black color.

    Parameters
    ----------
    scale: float or tuple(float)
        if **float**, the image is scaled by a factor drawn
        randomly from a range (1 - `scale` , 1 + `scale`). If **tu
        the `scale` is drawn randomly from values specified by the
        tuple

    Returns
    -------

    numpy.ndaaray
        Scaled image in the numpy format of shape `HxWxC`

    numpy.ndarray
        Tranformed bounding box co-ordinates of the format `n x 4`
        number of bounding boxes and 4 represents `x1,y1,x2,y2` of

    """

    def __init__(self, scale = 0.2, diff = False):
        self.scale = scale


        if type(self.scale) == tuple:
            assert len(self.scale) == 2, "Invalid range"
            assert self.scale[0] > -1, "Scale factor can't be less
            assert self.scale[1] > -1, "Scale factor can't be less
        else:
            assert self.scale > 0, "Please input a positive float"
            self.scale = (max(-1, -self.scale), self.scale)

        self.diff = diff
```

One of the takeaway lessons here is that shouldn't sample the final parameter in `__init__` function. If you do sample the parameter in the `__init__` function, the same value of the parameter would be used every time you call function. This takes away the stochastic element of augmentation.

Moving it to the `__call__` function will result in the parameter having a different value (from the range) every time the augmentation is applied.

Augmentation Logic

The logic of the Scale transformation is fairly simple. We use the OpenCV function `cv2.resize` to scale our image, and scale our bounding boxes by the scale factor(s).

```
img_shape = img.shape

if self.diff:
        scale_x = random.uniform(*self.scale)
        scale_y = random.uniform(*self.scale)
else:
        scale_x = random.uniform(*self.scale)
        scale_y = scale_x

resize_scale_x = 1 + scale_x
resize_scale_y = 1 + scale_y

img=  cv2.resize(img, None, fx = resize_scale_x, fy = resize_scale_

bboxes[:,:4] *= [resize_scale_x, resize_scale_y, resize_scale_x, r
```

However, we will keep the size constant. If we are going to scale down, there would be remaining area. We will color it black. The output would look like the augmented image shown above.

First we start by creating a black image of the size of our original image.

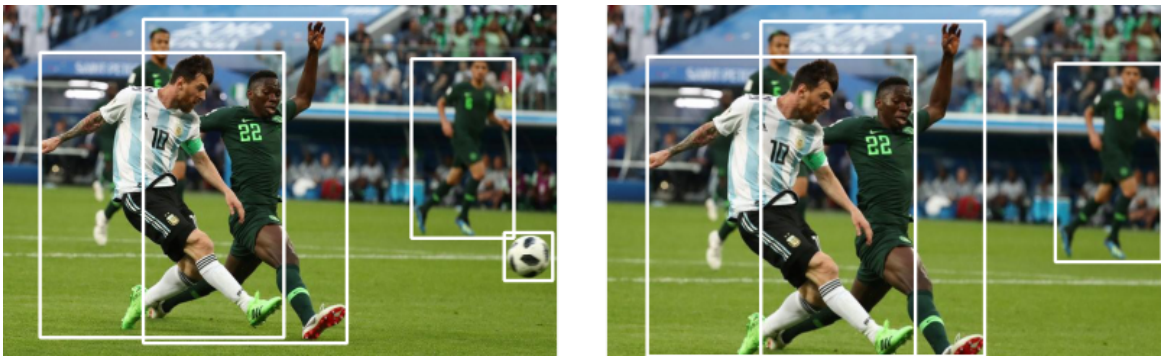
```
canvas = np.zeros(img_shape, dtype = np.uint8)
```

Then, we determine the the size of our scaled image. If it exceeds the dimensions of the original image (we are scaling up) , then it needs to be cut off at the original dimensions. We then "paste" the resized image on the canvas.

```
y_lim = int(min(resize_scale_y,1)*img_shape[0])
x_lim = int(min(resize_scale_x,1)*img_shape[1])

canvas[:y_lim,:x_lim,:] =  img[:y_lim,:x_lim,:]

img = canvas
```

Bounding Box Clipping

The only last thing which remains is the case an object is expelled from the image as a result of the scaling. For example, consider scaling up by a factor of 0.1 which means the final image dimensions are 1.1x the original image. This expels the football from our image.



The football is expelled as result of scaling up.

Think about it, and one realises such a scenario arises not only while scaling up, but other augmentations like translation too.

Therefore, we define a function `clip_box` in our helper file `bbox_utils.py` which clips bounding boxes based on the percentage of their total area lying inside the boundaries of the image. This percentage is a controllable parameter.

In the file `bbox_utils.py` define,

```
def clip_box(bbox, clip_box, alpha):
    """Clip the bounding boxes to the borders of an image

    Parameters
    ----------

    bbox: numpy.ndarray
        Numpy array containing bounding boxes of shape `N X 4` whe
        number of bounding boxes and the bounding boxes are repres
        format `x1 y1 x2 y2`

    clip_box: numpy.ndarray
        An array of shape (4,) specifying the diagonal co-ordinate
        The coordinates are represented in the format `x1 y1 x2 y2
```

```
    alpha: float
        If the fraction of a bounding box left in the image after
        less than `alpha` the bounding box is dropped.

    Returns
    -------

    numpy.ndarray
        Numpy array containing **clipped** bounding boxes of shape
        number of bounding boxes left are being clipped and the bo
        format `x1 y1 x2 y2`

    """
    ar_ = (bbox_area(bbox))
    x_min = np.maximum(bbox[:,0], clip_box[0]).reshape(-1,1)
    y_min = np.maximum(bbox[:,1], clip_box[1]).reshape(-1,1)
    x_max = np.minimum(bbox[:,2], clip_box[2]).reshape(-1,1)
    y_max = np.minimum(bbox[:,3], clip_box[3]).reshape(-1,1)

    bbox = np.hstack((x_min, y_min, x_max, y_max, bbox[:,4:]))

    delta_area = ((ar_ - bbox_area(bbox))/ar_)

    mask = (delta_area < (1 - alpha)).astype(int)

    bbox = bbox[mask == 1,:]


    return bbox
```

The function above modifies `bboxes` array, and removes the bounding boxes which lose too much area as a result of the augmentation.

We then simply use this function in the `__call__` method of our `RandomScale` to make sure all the boxes are clipped. Here, we drop all those bounding boxes which have less than 25% of their areas in confines of the image.

```
bboxes = clip_box(bboxes, [0,0,1 + img_shape[1], img_shape[0]], 0.
```

In order to compute the area of a bounding box, we also define a function `bbox_area`.

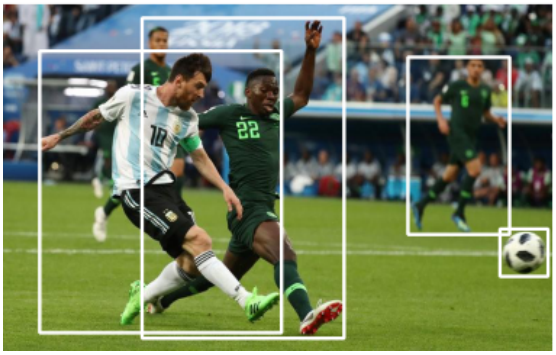
Finally, our completed `__call__` method looks like:

```
def __call__(self, img, bboxes):


        #Chose a random digit to scale by

        img_shape = img.shape

        if self.diff:
                scale_x = random.uniform(*self.scale)
                scale_y = random.uniform(*self.scale)
        else:
                scale_x = random.uniform(*self.scale)
                scale_y = scale_x

    resize_scale_x = 1 + scale_x
    resize_scale_y = 1 + scale_y

    img=  cv2.resize(img, None, fx = resize_scale_x, fy = resize_s

    bboxes[:,:4] *= [resize_scale_x, resize_scale_y, resize_scale_



    canvas = np.zeros(img_shape, dtype = np.uint8)

    y_lim = int(min(resize_scale_y,1)*img_shape[0])
    x_lim = int(min(resize_scale_x,1)*img_shape[1])

    print(y_lim, x_lim)

    canvas[:y_lim,:x_lim,:] =  img[:y_lim,:x_lim,:]

    img = canvas
```

```
    bboxes = clip_box(bboxes, [0,0,1 + img_shape[1], img_shape[0]]


    return img, bboxes
```

Translate

The next augmentation we are going to cover is the Translate, which produces an effect like this.



Again, like the Scale augmentation, the parameter of the augmentation is the factor of the dimensions of the image, by which the image should be translated. The same design decisions apply.

In addition to making sure that the translate factor is not less than -1, you should also make sure it isn't greater than 1, or else you're just gonna get a black image since whole of the image will be shifted.

```
class RandomTranslate(object):
    """Randomly Translates the image
```

```
    Bounding boxes which have an area of less than 25% in the rema
    transformed image is dropped. The resolution is maintained, an
    area if any is filled by black color.

    Parameters
    ----------
    translate: float or tuple(float)
        if **float**, the image is translated by a factor drawn
        randomly from a range (1 - `translate` , 1 + `translate`).
        `translate` is drawn randomly from values specified by the
        tuple

    Returns
    -------

    numpy.ndaaray
        Translated image in the numpy format of shape `HxWxC`

    numpy.ndarray
        Tranformed bounding box co-ordinates of the format `n x 4`
        number of bounding boxes and 4 represents `x1,y1,x2,y2` of

    """

    def __init__(self, translate = 0.2, diff = False):
        self.translate = translate

        if type(self.translate) == tuple:
            assert len(self.translate) == 2, "Invalid range"
            assert self.translate[0] > 0 & self.translate[0] < 1
            assert self.translate[1] > 0 & self.translate[1] < 1


        else:
            assert self.translate > 0 & self.translate < 1
            self.translate = (-self.translate, self.translate)


        self.diff = diff
```

Augmentation of Logic

The logic of this augmentation is a whole lot trickier than Scale. So, it

deserves some explaination.

We first start by setting up our variables.

```
def __call__(self, img, bboxes):
    #Chose a random digit to scale by
    img_shape = img.shape

    #translate the image

    #percentage of the dimension of the image to translate
    translate_factor_x = random.uniform(*self.translate)
    translate_factor_y = random.uniform(*self.translate)

    if not self.diff:
    translate_factor_y = translate_factor_x
```

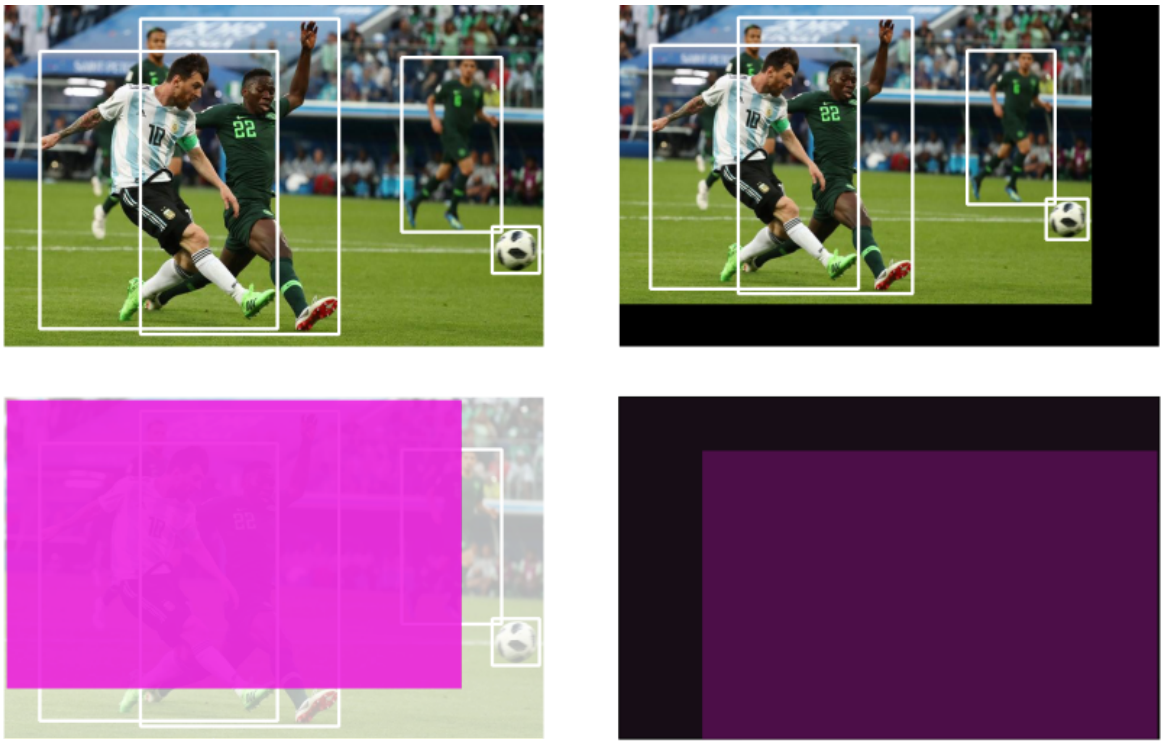
Now, when we translate the Image, it leaves some trailing space. We will color it black. If you look at the image above demonstrating translation, you can easily make out the black space.

We proceed as before. We first initialise a black image about the size of our original image.

```
canvas = np.zeros(img_shape)
```

Then, we have two tasks. First, as shown in image one to determine which part of the black canvas the image will be pasted on.

Second, which part of the image will be pasted.



Left: The part of image which will be pasted. Right: The area of Canvas on which the image will be pasted,

So, we first figure out the area of the image on which we will paste on the canvas image. (Purple patch on the image, left in the above image).

```
#get the top-left corner co-ordinates of the shifted image
corner_x = int(translate_factor_x*img.shape[1])
corner_y = int(translate_factor_y*img.shape[0])

mask = img[max(-corner_y, 0):min(img.shape[0], -corner_y + img_sha
```

Now let us get the portion of the canvas on which we "paste" `mask`

```
orig_box_cords =  [max(0,corner_y), max(corner_x,0), min(img_shape

canvas[orig_box_cords[0]:orig_box_cords[2], orig_box_cords[1]:orig
img = canvas
```

Shifting boxes is relatively simple. You simply need to offset the corners of the bounding boxes. We also clip the bounding boxes that have less than 25% of area inside the image as a result of the augmentation.

```
bboxes[:,:4] += [corner_x, corner_y, corner_x, corner_y]

bboxes = clip_box(bboxes, [0,0,img_shape[1], img_shape[0]], 0.25)
```

To sum up, our call function looks like this.

```
def __call__(self, img, bboxes):
    #Chose a random digit to scale by
    img_shape = img.shape

    #translate the image

    #percentage of the dimension of the image to translate
    translate_factor_x = random.uniform(*self.translate)
    translate_factor_y = random.uniform(*self.translate)

    if not self.diff:
        translate_factor_y = translate_factor_x

    canvas = np.zeros(img_shape).astype(np.uint8)
```

```
    corner_x = int(translate_factor_x*img.shape[1])
    corner_y = int(translate_factor_y*img.shape[0])



    #change the origin to the top-left corner of the translated bo
    orig_box_cords =  [max(0,corner_y), max(corner_x,0), min(img_s

    mask = img[max(-corner_y, 0):min(img.shape[0], -corner_y + img
    canvas[orig_box_cords[0]:orig_box_cords[2], orig_box_cords[1]:
    img = canvas

    bboxes[:,:4] += [corner_x, corner_y, corner_x, corner_y]


    bboxes = clip_box(bboxes, [0,0,img_shape[1], img_shape[0]], 0.



    return img, bboxes
```

Testing

As detailed in the last example, you can test these augmentations on the sample image, or your own image provided you have stored the augmentation in the correct format.

```
from data_aug.bbox_utils import *
import matplotlib.pyplot as plt

scale = RandomScale(0.2, diff = True)
translate = RandomTranslate(0.2, diff = True)

img, bboxes = translate(img, bboxes)
img,bboxes = scale(img, bboxes)
```

```
plt.imshow(draw_rect(img, bboxes))
```

The end result might look like this.

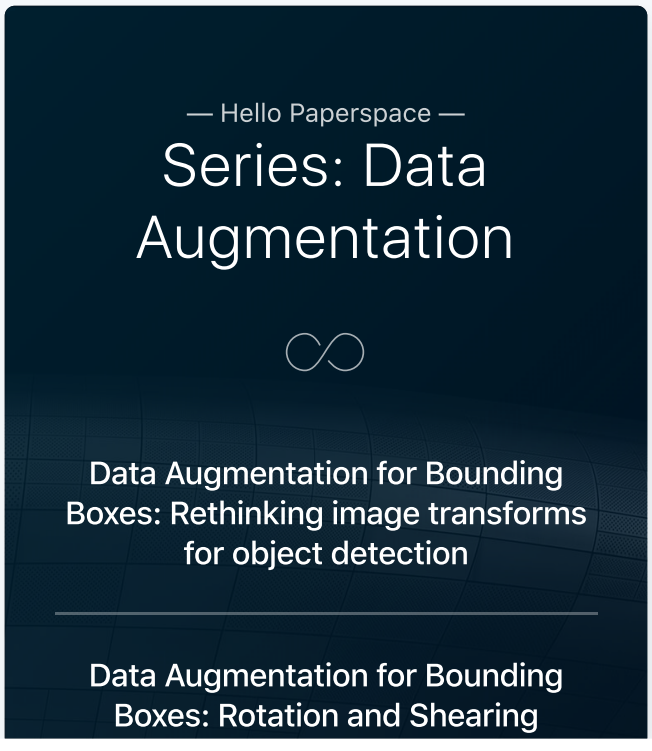
You can try doing scaling first, and then translating. You might find that the results for the same values of parameters are different. Can you explain it? This is left as an exercise of the reader.

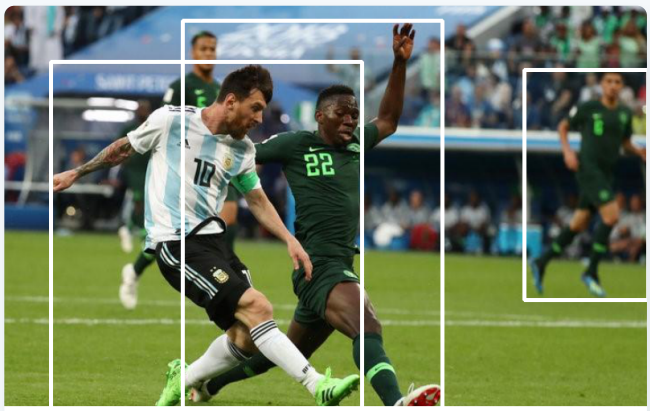
Another exercise can be trying to implement the deterministic versions of the augmentations we've implemented.

This is it for this part. In the next part, the augmentation will get a bit more advance and a bit more messier with Rotation and Shearing where we will be using the affine transformation features of OpenCV. Stay tuned!

Ayoosh Kathuria

Currently a research intern at DRDO, the premier defence research facility in India, I'm extremely passionate about Computer Vision, multimodal learning and generative modelling.





SERIES: DATA AUGMENTATION

Data Augmentation for Bounding Boxes: Rethinking image transforms for object detection

How to adapt major image

Data Augmentation For Bounding Boxes: Building Input Pipelines for your detector

See all 3 posts →

augmentation techniques for object detection purposes. We also cover the implementation of horizontal flip augmentation.

9 MIN READ

COMPUTER VISION

Data Augmentation for Bounding Boxes: Rotation and Shearing

This is part 3 of the series where we are looking at ways to adapt image augmentation techniques to object detection tasks. In this part, we will cover how to implement how to rotate and shear images as well as bounding boxes using OpenCV's affine transformation features.

12 MIN READ