3mvwrpeqv

May 11, 2023

FASTER RCNN

Faster R-CNN is an object detection algorithm that helps computers recognize and locate objects within an image.

Here's how it works:

The algorithm takes an input image and feeds it through a convolutional neural network (CNN) to extract feature maps. It then applies a Region Proposal Network (RPN) to suggest potential object locations in the image. The RPN uses anchor boxes of different sizes and aspect ratios to generate these proposals. These proposals are then refined and pruned using a region of interest (ROI) pooling layer, which extracts a fixed-size feature map for each proposal. These feature maps are then fed into a series of fully connected layers to classify the objects within each proposal and predict their bounding boxes.

###**DATASET DOWNLOAD**

[ ]:
```
#I am working on object detection using Faster R-CNN and I need to use the COCO␣
  ↪dataset.
#To download the dataset, I am using the following command:

!wget 'http://images.cocodataset.org/zips/val2017.zip'
```

```
--2023-04-28 10:36:03--  http://images.cocodataset.org/zips/val2017.zip
Resolving images.cocodataset.org (images.cocodataset.org)… 52.216.52.209,
52.217.161.17, 52.216.95.75, …
Connecting to images.cocodataset.org
(images.cocodataset.org)|52.216.52.209|:80… connected.
HTTP request sent, awaiting response… 200 OK
Length: 815585330 (778M) [application/zip]
Saving to: ‘val2017.zip’

val2017.zip         100%[===================>] 777.80M  45.9MB/s    in 18s

2023-04-28 10:36:21 (42.4 MB/s) - ‘val2017.zip’ saved [815585330/815585330]
```

What do the following commands mean?

This command downloads a compressed file containing images from the COCO dataset from the specified URL. The file is being downloaded from the specified URL "http://images.cocodataset.org/zips/val2017.zip".

```
[ ]: # Once the file is downloaded, it can be extracted and used for object␣
      ↪detection using the Faster R-CNN algorithm.

     !unzip 'val2017.zip'
```

0.1 IMPORT LIBRARIES

```
[ ]: import torch
     import torchvision
     from torchvision import transforms as T
```

What do the following commands mean?

import torch: This line imports the PyTorch library. PyTorch is a popular open-source machine learning framework used for building neural networks.

import torchvision: This line imports the torchvision package, which provides datasets, transforms, and models specific to computer vision tasks.

from torchvision import transforms as T: This line imports the transforms module from torchvision, which provides various image transformation functions that can be used to preprocess images before training a neural network. The as T statement renames the module to T for ease of use in the code.

```
[ ]: from PIL import Image
     import cv2
     from google.colab.patches import cv2_imshow
```

What do the following commands mean?

from PIL import Image: The PIL (Python Imaging Library) module provides a set of Python classes that allow you to open, manipulate, and save different image file formats. In this case, it is being used to read and load images into the program.

import cv2: The OpenCV (Open Source Computer Vision) library is a computer vision and machine learning software library used for image and video processing. In this case, it is being used to manipulate and process images.

from google.colab.patches import cv2_imshow: This line is specific to the Google Colaboratory (Colab) environment, which is an online platform for running Python code. The cv2_imshow function provided by this module is used to display images in a Colab notebook.

0.2 FASTER RCNN

Here, we will be using pretrained Fatser RCNN which is already pre-trained on a large-scale object detection dataset such as COCO or ImageNet. For that we need to create instance of faster rcnn.

```
[ ]: # Creating instance of Faster RCNN
```

```
model = torchvision.models.detection.fasterrcnn_resnet50_fpn(pretrained = True)
```

```
/usr/local/lib/python3.10/dist-packages/torchvision/models/_utils.py:208:
UserWarning: The parameter 'pretrained' is deprecated since 0.13 and may be
removed in the future, please use 'weights' instead.
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/torchvision/models/_utils.py:223:
UserWarning: Arguments other than a weight enum or `None` for 'weights' are
deprecated since 0.13 and may be removed in the future. The current behavior is
equivalent to passing `weights=FasterRCNN_ResNet50_FPN_Weights.COCO_V1`. You can
also use `weights=FasterRCNN_ResNet50_FPN_Weights.DEFAULT` to get the most up-
to-date weights.
  warnings.warn(msg)
Downloading:
"https://download.pytorch.org/models/fasterrcnn_resnet50_fpn_coco-258fb6c6.pth"
to /root/.cache/torch/hub/checkpoints/fasterrcnn_resnet50_fpn_coco-258fb6c6.pth
100%|      | 160M/160M [00:01<00:00, 108MB/s]
```

What do the following commands mean?

torchvision.models.detection: This module contains various pre-built object detection models that can be used with PyTorch.

fasterrcnn__resnet50__fpn: This is the specific Faster R-CNN model being used. It is based on the ResNet-50 architecture and uses a feature pyramid network (FPN) to extract features at different scales.

pretrained=True: This argument specifies that the model should be initialized with pre-trained weights, which were trained on a large-scale object detection dataset such as COCO or ImageNet. This is often used as a starting point for transfer learning, where the model is fine-tuned on a smaller dataset such as the COCO dataset.

```
[ ]: #Set the model in evaluation mode

     model.eval()
```

```
[ ]: FasterRCNN(
       (transform): GeneralizedRCNNTransform(
           Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
           Resize(min_size=(800,), max_size=1333, mode='bilinear')
       )
       (backbone): BackboneWithFPN(
         (body): IntermediateLayerGetter(
           (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3),
     bias=False)
           (bn1): FrozenBatchNorm2d(64, eps=0.0)
           (relu): ReLU(inplace=True)
           (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1,
     ceil_mode=False)
```

```
      (layer1): Sequential(
        (0): Bottleneck(
          (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (bn1): FrozenBatchNorm2d(64, eps=0.0)
          (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
          (bn2): FrozenBatchNorm2d(64, eps=0.0)
          (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (bn3): FrozenBatchNorm2d(256, eps=0.0)
          (relu): ReLU(inplace=True)
          (downsample): Sequential(
            (0): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (1): FrozenBatchNorm2d(256, eps=0.0)
          )
        )
        (1): Bottleneck(
          (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (bn1): FrozenBatchNorm2d(64, eps=0.0)
          (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
          (bn2): FrozenBatchNorm2d(64, eps=0.0)
          (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (bn3): FrozenBatchNorm2d(256, eps=0.0)
          (relu): ReLU(inplace=True)
        )
        (2): Bottleneck(
          (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (bn1): FrozenBatchNorm2d(64, eps=0.0)
          (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
          (bn2): FrozenBatchNorm2d(64, eps=0.0)
          (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (bn3): FrozenBatchNorm2d(256, eps=0.0)
          (relu): ReLU(inplace=True)
        )
      )
      (layer2): Sequential(
        (0): Bottleneck(
          (conv1): Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (bn1): FrozenBatchNorm2d(128, eps=0.0)
          (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
          (bn2): FrozenBatchNorm2d(128, eps=0.0)
          (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (bn3): FrozenBatchNorm2d(512, eps=0.0)
          (relu): ReLU(inplace=True)
          (downsample): Sequential(
            (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
            (1): FrozenBatchNorm2d(512, eps=0.0)
          )
        )
        (1): Bottleneck(
          (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (bn1): FrozenBatchNorm2d(128, eps=0.0)
          (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
          (bn2): FrozenBatchNorm2d(128, eps=0.0)
          (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (bn3): FrozenBatchNorm2d(512, eps=0.0)
          (relu): ReLU(inplace=True)
        )
        (2): Bottleneck(
          (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (bn1): FrozenBatchNorm2d(128, eps=0.0)
          (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
          (bn2): FrozenBatchNorm2d(128, eps=0.0)
          (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (bn3): FrozenBatchNorm2d(512, eps=0.0)
          (relu): ReLU(inplace=True)
        )
        (3): Bottleneck(
          (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (bn1): FrozenBatchNorm2d(128, eps=0.0)
          (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
          (bn2): FrozenBatchNorm2d(128, eps=0.0)
          (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (bn3): FrozenBatchNorm2d(512, eps=0.0)
          (relu): ReLU(inplace=True)
        )
```

```
      )
      (layer3): Sequential(
        (0): Bottleneck(
          (conv1): Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (bn1): FrozenBatchNorm2d(256, eps=0.0)
          (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
          (bn2): FrozenBatchNorm2d(256, eps=0.0)
          (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (bn3): FrozenBatchNorm2d(1024, eps=0.0)
          (relu): ReLU(inplace=True)
          (downsample): Sequential(
            (0): Conv2d(512, 1024, kernel_size=(1, 1), stride=(2, 2),
bias=False)
            (1): FrozenBatchNorm2d(1024, eps=0.0)
          )
        )
        (1): Bottleneck(
          (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (bn1): FrozenBatchNorm2d(256, eps=0.0)
          (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
          (bn2): FrozenBatchNorm2d(256, eps=0.0)
          (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (bn3): FrozenBatchNorm2d(1024, eps=0.0)
          (relu): ReLU(inplace=True)
        )
        (2): Bottleneck(
          (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (bn1): FrozenBatchNorm2d(256, eps=0.0)
          (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
          (bn2): FrozenBatchNorm2d(256, eps=0.0)
          (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (bn3): FrozenBatchNorm2d(1024, eps=0.0)
          (relu): ReLU(inplace=True)
        )
        (3): Bottleneck(
          (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (bn1): FrozenBatchNorm2d(256, eps=0.0)
          (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
          (bn2): FrozenBatchNorm2d(256, eps=0.0)
          (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (bn3): FrozenBatchNorm2d(1024, eps=0.0)
          (relu): ReLU(inplace=True)
        )
        (4): Bottleneck(
          (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (bn1): FrozenBatchNorm2d(256, eps=0.0)
          (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
          (bn2): FrozenBatchNorm2d(256, eps=0.0)
          (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (bn3): FrozenBatchNorm2d(1024, eps=0.0)
          (relu): ReLU(inplace=True)
        )
        (5): Bottleneck(
          (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (bn1): FrozenBatchNorm2d(256, eps=0.0)
          (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
          (bn2): FrozenBatchNorm2d(256, eps=0.0)
          (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (bn3): FrozenBatchNorm2d(1024, eps=0.0)
          (relu): ReLU(inplace=True)
        )
      )
      (layer4): Sequential(
        (0): Bottleneck(
          (conv1): Conv2d(1024, 512, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (bn1): FrozenBatchNorm2d(512, eps=0.0)
          (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
          (bn2): FrozenBatchNorm2d(512, eps=0.0)
          (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (bn3): FrozenBatchNorm2d(2048, eps=0.0)
          (relu): ReLU(inplace=True)
          (downsample): Sequential(
            (0): Conv2d(1024, 2048, kernel_size=(1, 1), stride=(2, 2),
bias=False)
            (1): FrozenBatchNorm2d(2048, eps=0.0)
          )
        )
        (1): Bottleneck(
          (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (bn1): FrozenBatchNorm2d(512, eps=0.0)
          (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
          (bn2): FrozenBatchNorm2d(512, eps=0.0)
          (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (bn3): FrozenBatchNorm2d(2048, eps=0.0)
          (relu): ReLU(inplace=True)
        )
        (2): Bottleneck(
          (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (bn1): FrozenBatchNorm2d(512, eps=0.0)
          (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
          (bn2): FrozenBatchNorm2d(512, eps=0.0)
          (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (bn3): FrozenBatchNorm2d(2048, eps=0.0)
          (relu): ReLU(inplace=True)
        )
      )
    )
    (fpn): FeaturePyramidNetwork(
      (inner_blocks): ModuleList(
        (0): Conv2dNormActivation(
          (0): Conv2d(256, 256, kernel_size=(1, 1), stride=(1, 1))
        )
        (1): Conv2dNormActivation(
          (0): Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1))
        )
        (2): Conv2dNormActivation(
          (0): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1))
        )
        (3): Conv2dNormActivation(
          (0): Conv2d(2048, 256, kernel_size=(1, 1), stride=(1, 1))
        )
      )
      (layer_blocks): ModuleList(
        (0-3): 4 x Conv2dNormActivation(
          (0): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
        )
      )
      (extra_blocks): LastLevelMaxPool()
    )
  )
  (rpn): RegionProposalNetwork(
    (anchor_generator): AnchorGenerator()
    (head): RPNHead(
      (conv): Sequential(
        (0): Conv2dNormActivation(
          (0): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
          (1): ReLU(inplace=True)
        )
      )
      (cls_logits): Conv2d(256, 3, kernel_size=(1, 1), stride=(1, 1))
      (bbox_pred): Conv2d(256, 12, kernel_size=(1, 1), stride=(1, 1))
    )
  )
  (roi_heads): RoIHeads(
    (box_roi_pool): MultiScaleRoIAlign(featmap_names=['0', '1', '2', '3'],
output_size=(7, 7), sampling_ratio=2)
    (box_head): TwoMLPHead(
      (fc6): Linear(in_features=12544, out_features=1024, bias=True)
      (fc7): Linear(in_features=1024, out_features=1024, bias=True)
    )
    (box_predictor): FastRCNNPredictor(
      (cls_score): Linear(in_features=1024, out_features=91, bias=True)
      (bbox_pred): Linear(in_features=1024, out_features=364, bias=True)
    )
  )
)
```

```
[ ]: # Here we are just showing a simple example in which we are detecting object in␣
      ↪one image.
     # For that we need to download one specific image in which we want to detect␣
      ↪objects
     # Following command will download image 000000037988.jpg from val2017 dataset

     !wget 'http://images.cocodataset.org/val2017/000000037988.jpg'
```

```
--2023-04-28 10:37:20--  http://images.cocodataset.org/val2017/000000037988.jpg
Resolving images.cocodataset.org (images.cocodataset.org)… 54.231.171.185,
54.231.166.209, 52.217.98.172, …
Connecting to images.cocodataset.org
```

```
(images.cocodataset.org)|54.231.171.185|:80… connected.
HTTP request sent, awaiting response… 200 OK
Length: 67967 (66K) [image/jpeg]
Saving to: ‘000000037988.jpg’

000000037988.jpg    100%[===================>]  66.37K  --.-KB/s    in 0.1s

2023-04-28 10:37:20 (536 KB/s) - ‘000000037988.jpg’ saved [67967/67967]
```

[ ]:
```python
img = Image.open ("/content/000000037988.jpg")
```

[ ]:
```python
img
```

[ ]:

Most deep learning frameworks, including PyTorch, require input data to be in the form of tensors. Therefore, we need to convert above PIL image object to PyTorch tensor.

[ ]:
```python
# convert PIL image object to PyTorch tensor

transform = T. ToTensor ()
img = transform(img)
```

What do the following commands mean?

transform = T.ToTensor(): This line creates an instance of the ToTensor transform from the torchvision.transforms module. This transform converts a PIL image object to a PyTorch tensor. Specifically, it normalizes the pixel values to the range [0, 1] and converts the image to a tensor with shape (C, H, W), where C is the number of color channels (e.g., 3 for RGB images) and H and W are the height and width of the image.

img = transform(img): This line applies the ToTensor transform to the PIL image object img. The transformed image is stored in the variable img.

```
[ ]: with torch.no_grad():
       pred = model([img])
```

Here we are using pre trained faster RCNN. Tgerefore, we dont need to update the waights. And if we are not updating weights then there is no need to compute the gradients. Hence We need to disable gradient computation during inference because we don't need to update the model's parameters during this phase.

What do the following commands mean?

with torch.no__grad(): This is a context manager provided by PyTorch that disables gradient computation during the execution of the indented code block. In other words, any computation that involves gradients, such a backpropagation, is not performed within this block.

pred = model([img]): This line of code calls the PyTorch model's forward method with the input image tensor img as its argument. The forward method returns the output of the model, which in this case is the predicted bounding boxes and labels for any objects detected in the input image. The output is stored in the variable pred.

```
[ ]: pred
```

```
[ ]: [{'boxes': tensor([[143.5724,   0.8510, 235.5486, 120.4438],
               [308.2853, 143.5528, 408.5497, 447.0229],
               [283.7333, 246.5087, 318.0598, 338.2904],
               [350.5255,  60.6031, 361.8595,  72.1602],
               [379.0357,   0.0000, 439.4783,  44.5615]]),
       'labels': tensor([ 1,  1, 43, 37,  1]),
       'scores': tensor([0.9996, 0.9995, 0.9994, 0.9989, 0.9984])}]
```

What do the following commands mean?

'boxes': tensor(...): This is a tensor of size (N, 4) that contains the coordinates of the bounding boxes for N objects detected in the image.

'labels': tensor(...): This is a tensor of size (N,) that contains the class labels for the N objects detected in the image.

'scores': tensor(...): This is a tensor of size (N,) that contains the confidence scores for the N objects detected in the image.

```
[ ]: pred[0].keys()
[ ]: dict_keys(['boxes', 'labels', 'scores'])
```

What do the following commands mean?

pred[0].keys() returns the keys of a Python dictionary that contains the output of a Faster R-CNN model for object detection on an input image.

```
[ ]: scores=pred[0]['scores']
```

```
[ ]: scores>0.9
```

```
[ ]: tensor([True, True, True, True, True])
```

```
[ ]: torch.argwhere(scores > 0.9)
```

```
[ ]: tensor([[0],
             [1],
             [2],
             [3],
             [4]])
```

```
[ ]: num = torch. argwhere (scores > 0.9).shape [0]
```

```
[ ]: num
```

```
[ ]: 5
```

What do the following commands mean?

torch.argwhere(scores > 0.9) returns a tensor containing the indices of all elements in scores that are greater than 0.9.

```
[ ]: bboxes=pred[0]['boxes']
```

```
[ ]: bboxes
```

```
[ ]: tensor([[143.5724,   0.8510, 235.5486, 120.4438],
             [308.2853, 143.5528, 408.5497, 447.0229],
             [283.7333, 246.5087, 318.0598, 338.2904],
             [350.5255,  60.6031, 361.8595,  72.1602],
             [379.0357,   0.0000, 439.4783,  44.5615]])
```

```
[ ]: labels=pred[0]['labels']
```

```
[ ]: labels
```

```
[ ]: tensor([ 1,  1, 43, 37,  1])
```

What do the following commands mean?

pred[0]['boxes']: returns a 2D PyTorch tensor that contains the bounding boxes of the detected objects in the input image.

Each row in the tensor represents the coordinates of a bounding box in the format (xmin, ymin, xmax, ymax), where (xmin, ymin) are the coordinates of the top-left corner of the bounding box, and (xmax, ymax) are the coordinates of the bottom-right corner of the bounding box.

0.3 COCO Dataset Classes

Now we need to define COCO dataset Classes.

[ ]:
```
coco_names = ["person" , "bicycle" , "car" , "motorcycle" , "airplane" , "bus"␣
 ↪, "train" , "truck" , "boat" , "traffic light" , "fire hydrant" , "street␣
 ↪sign" , "stop sign" , "parking meter" , "bench" , "bird" , "cat" , "dog" ,␣
 ↪"horse" , "sheep" , "cow" , "elephant" , "bear" , "zebra" , "giraffe" ,␣
 ↪"hat" , "backpack" , "umbrella" , "shoe" , "eye glasses" , "handbag" , "tie"␣
 ↪, "suitcase" ,
"frisbee" , "skis" , "snowboard" , "sports ball" , "kite" , "baseball bat" ,
"baseball glove" , "skateboard" , "surfboard" , "tennis racket" , "bottle" ,
"plate" , "wine glass" , "cup" , "fork" , "knife" , "spoon" , "bowl" ,
"banana" , "apple" , "sandwich" , "orange" , "broccoli" , "carrot" , "hot dog" ,
"pizza" , "donut" , "cake" , "chair" , "couch" , "potted plant" , "bed" ,
"mirror" , "dining table" , "window" , "desk" , "toilet" , "door" , "tv" ,
"laptop" , "mouse" , "remote" , "keyboard" , "cell phone" , "microwave" ,
"oven" , "toaster" , "sink" , "refrigerator" , "blender" , "book" ,
"clock" , "vase" , "scissors" , "teddy bear" , "hair drier" , "toothbrush" ,␣
 ↪"hair brush"]
```

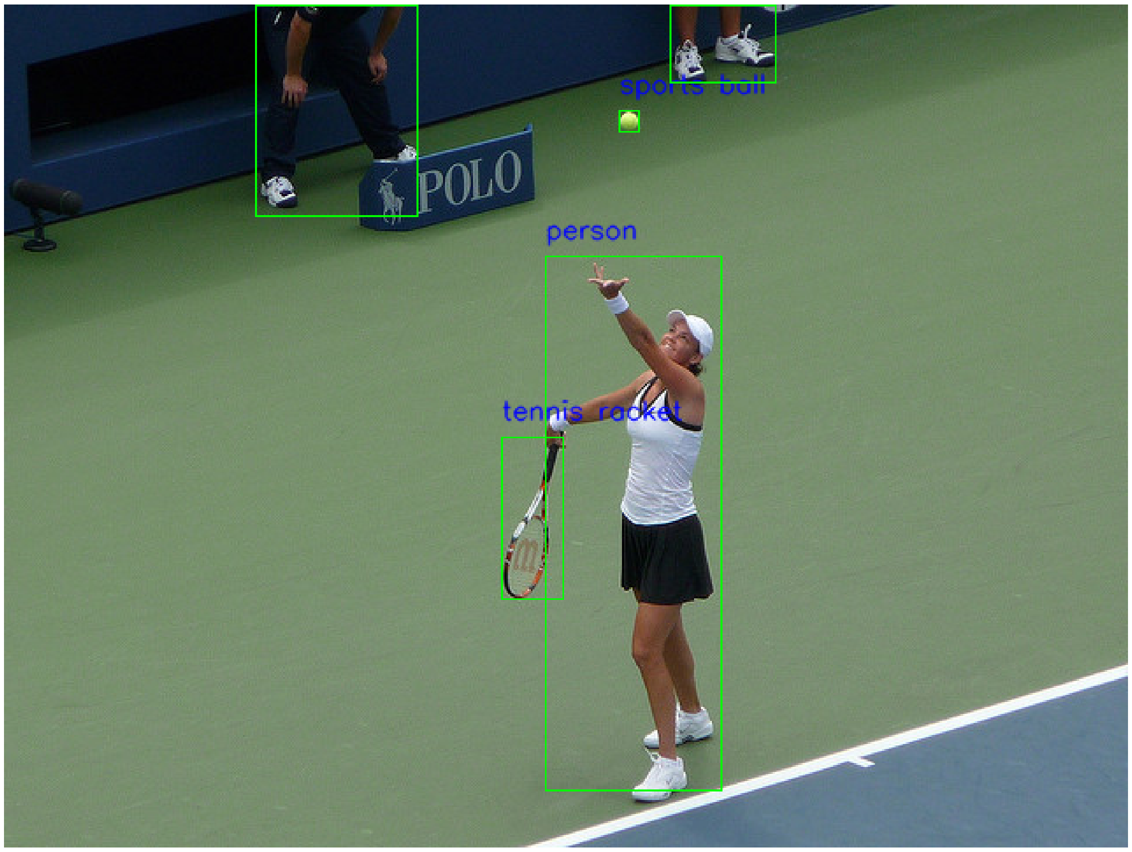
What do the following commands mean?

here, If a neural network outputs a class index of 1, we know that the predicted object is a "bicycle" since "bicycle" is the second element in the coco_names list.

[ ]:
```
font = cv2.FONT_HERSHEY_SIMPLEX
```

[ ]:
```
igg = cv2.imread ("/content/000000037988.jpg")
for i in range(num):
  x1, y1, x2, y2=bboxes[i].numpy().astype("int")
  class_name = coco_names[labels.numpy ()[i] - 1]
  igg = cv2.rectangle(igg, (x1 , y1),(x2,y2),(0,255,0),1)
  igg = cv2.putText (igg , class_name , (x1 , y1 - 10) , font , 0.5 , (255, 0 ,␣
 ↪0) , 1 , cv2. LINE_AA)
```

[ ]:
```
cv2_imshow(igg)
```


sports ball
POLO
person
tennis racket