University of New Haven

DSCI-6671 Deep Learning

Final Project Report

Implementation of Faster R-CNN

Prepared by:

Srimanikanta Arjun Karimalammanavar

Hemendra Jampala

Prepared for:

Dr. Muhammad Aminul Islam

Introduction

Object Detection is a common Computer Vision problem which deals with identifying and locating object of certain classes in the image. Interpreting the object localization can be done in various ways, including creating a bounding box around the object or marking every pixel in the image which contains the object (called segmentation).

Object detection was studied even before the breakout popularity of CNNs in Computer Vision. While CNNs are capable of automatically extracting more complex and better features, taking a glance at the conventional methods can at worst be a small detour and at best an inspiration.

Object detection before Deep Learning was a several step process, starting with edge detection and feature extraction using techniques like SIFT, HOG etc. These images were then compared with existing object templates, usually at multi scale levels, to detect and localize objects present in the image.

Problem

The aim of our project is to implement Faster R-CNN network as friendly as possible to our readers.

State-of-the-art object detection networks depend on region proposal algorithms . . . . . . reduced running time of these detection networks . . .. introduce a *Region Proposal Network (RPN)* that shares full-image convolutional features enabling nearly cost-free region proposals.

- Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks
    - Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun

Method

The most widely used state of the art version of the R-CNN family — Faster R-CNN was first published in 2015.

In the R-CNN family of papers, the evolution between versions was usually in terms of computational efficiency (integrating the different training stages), reduction in test time, and improvement in performance (mAP). These networks usually consist of

a) A region proposal algorithm to generate “bounding boxes” or locations of possible objects in the image

b) A feature generation stage to obtain features of these objects, usually using a CNN

c) A classification layer to predict which class this object belongs to and

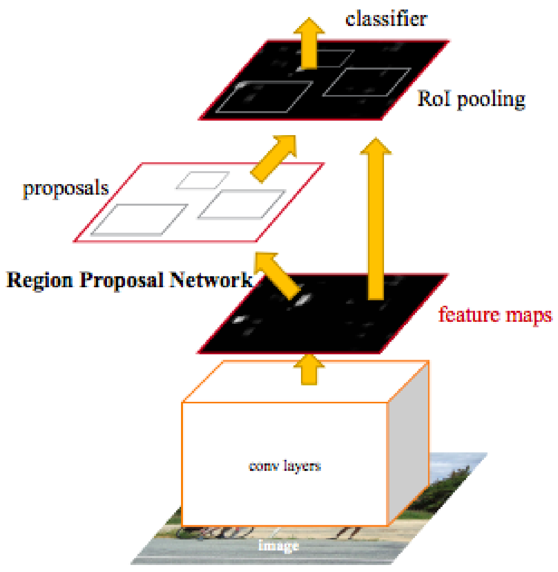
d) A regression layer to make the coordinates of the object bounding box more precise.

The only stand-alone portion of the network left in Fast R-CNN was the region proposal algorithm. Both R-CNN and Fast R-CNN use CPU based region proposal algorithms, E.g.- the Selective search algorithm which takes around 2 seconds per image and runs on CPU computation. The Faster R-CNN paper fixes this by using another convolutional network (the RPN) to generate the region proposals. This not only brings down the region proposal time from 2s to 10ms per image but also allows the region proposal stage to share layers with the following detection stages, causing an overall improvement in feature representation.

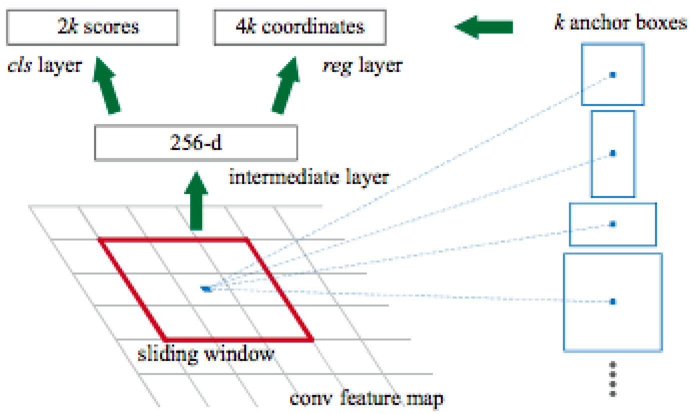
“Faster R-CNN” usually refers to a detection pipeline that uses the RPN as a region proposal algorithm, and Fast R-CNN as a detector network.

Region Proposal Network

To generate these so called “proposals” for the region where the object lies, a small **network** is slide over a convolutional feature map that is the output by the last convolutional layer.



Above is the architecture of Faster R-CNN. RPN generate the proposal for the objects. RPN has a specialized and unique architecture.



RPN has a classifier and a regressor. The authors have introduced the concept of anchors. Anchor is the central point of the sliding window. For ZF model, which was an extension of AlexNet, the dimensions are 256-d and for VGG-16, it was 512-d. Classifier determines the probability of a proposal having the target object. Regression regresses the coordinates of the proposals. For any image, scale and aspect-ratio are two important parameters. For those who do not know, aspect ratio = width of image/height of image, scale is the size of the image. The developers chose 3 scale and 3 aspect-ratio. So, total of 9 proposals are possible for each pixel, this is how the value of k is decided, k = 9 for this case, k being the number of anchors. For the whole image, number of anchors is W*H*K (50*50*9 = 22500).

But How Does It Work?

These anchors are assigned label based on two factors:

1. The anchors with highest Intersection-over-union overlap with a ground truth box.
2. The anchors with Intersection-Over-Union Overlap higher than 0.7.

Ultimately, RPN is an algorithm that needs to be trained. So, we have our Loss Function.

$$L(\{p_i\},\{t_i\}) = (1/N_{cls})\times\Sigma L_{cls}(p_i,p_i^*) + (\lambda/N_{reg})\times\Sigma p_i^* L_{reg}(t_i,t_i^*)$$

Loss Function

i → Index of anchor,

p → probability of being an object or not,

t →vector of 4 parameterized coordinates of predicted bounding box,

* represents ground truth box.

L_cls represents Log Loss over two classes.

$$L_{reg}(t_i, t_i^*) = R(t_i - t_i^*)$$

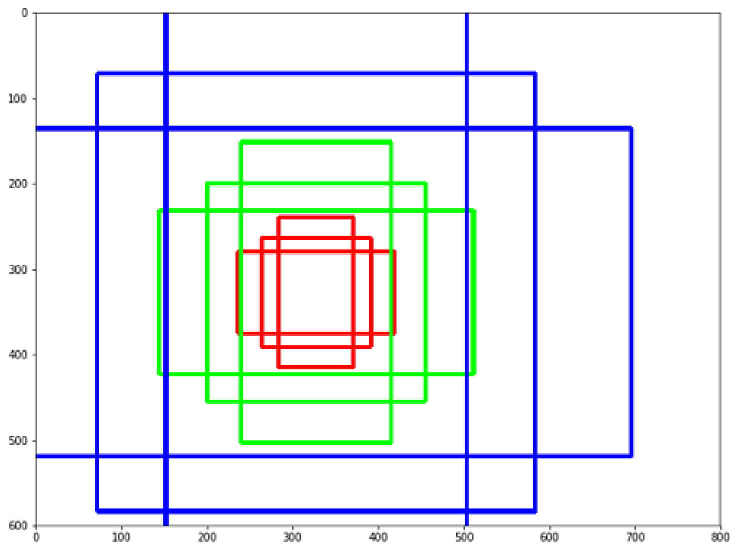
p* with regression term in the loss function ensures that if and only if object is identified as yes, then only regression will count, otherwise p* will be zero, so the regression term will become zero in the loss function.

N_cls and N_reg are the normalization. Default λ is 10 by default and is done to scale classifier and regressor on the same level.

Anchor Boxes

Anchor boxes are some of the most important concepts in Faster R-CNN. These are responsible for providing a predefined set of bounding boxes of different sizes and ratios that are used for reference when first predicting object locations for the RPN. These boxes are defined to capture the scale and aspect ratio of specific object classes you want to detect and are typically chosen based on object sizes in the training dataset. Anchor Boxes are typically centered at the sliding window.

The original implementation uses 3 scales and 3 aspect ratios, which means k=9. If the final feature map from feature extraction layer has width W and height H, then the total number of anchors generated will be W*H*k.



The Need for Anchor Boxes

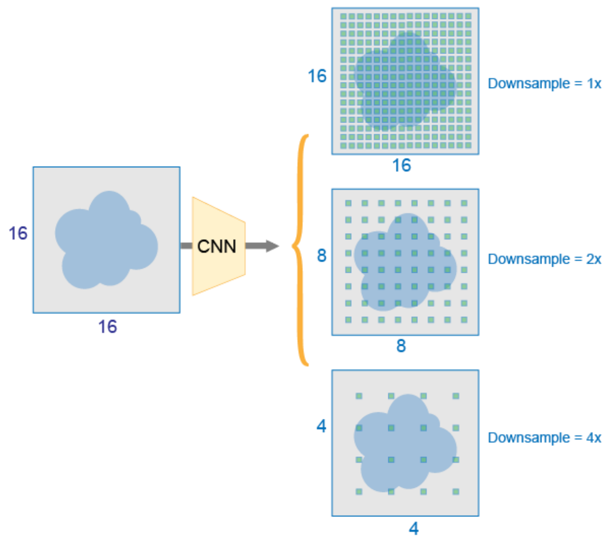
The main reason to use Anchor Boxes is so that we can evaluate all object predictions at once. They help speed up and improve efficiency for the detection portion of a deep learning neural network framework. Anchor boxes also help to detect multiple objects, objects of different scales, and overlapping objects without the need to scan an image with a sliding window that computes a separate prediction at every potential position as we saw in previous versions of R-CNN. This makes real time object detection possible.

As we mentioned before, the information from these anchor boxes is relayed to the regression and classification layer, where regression gives offsets from anchor boxes and classification gives the probability that each regressed anchor shows an object.

How do Anchor Boxes help in identifying an object?

Although anchors take the final feature map as input, the final anchors refer to the original image. This is made possible because of the convolution correspondence property of CNN's, thus enabling extracted features to be associated back to their location in that image. For a down sampling ratio d, the feature map will have dimensions W/d * H/d. In other words, in an image, each anchor point will be separated by d spatial pixels since we have just one at each spatial location of feature map. A value of 4 or 16 is common for d, which also corresponds to the *stride* between tiled anchor boxes. This is a tunable parameter in the configuration of Faster R-CNN.

Anchor boxes at each spatial location, mark an object as *foreground* or *background* depending on its IOU threshold with the ground truth. All the anchors are placed in a mini-batch and trained using softmax cross entropy to learn the classification loss and smooth L1 loss for regression. We use smooth L1 loss as regular L1 loss function is not differentiable at 0.



Non-maximum Suppression (NMS)

NMS is the second stage of filtering used to get rid of overlapping boxes, because even after filtering by thresholding over the classes scores, we still end up with a lot of overlapping boxes.

Overview of NMS:

1. Select the box that has the highest score.
2. Compute its overlap with all other boxes and remove boxes that overlap it more than the IOU threshold.
3. Go back to step 1 and iterate until there is no more boxes with a lower score than the current selected box.

This will remove all boxes that have a large overlap with the selected boxes. Only the “best” boxes remain. There are further improvements made to this method which is called Soft_NMS.

Object detection: Faster R-CNN (RPN + Fast R-CNN)

The Faster R-CNN architecture consists of the RPN as a region proposal algorithm and the Fast R-CNN as a detector network.

Fast R-CNN as a detector for Faster R-CNN

The Fast R-CNN detector also consists of a CNN backbone, an ROI pooling layer and fully connected layers followed by two sibling branches for classification and bounding box regression.

- The input image is first passed through the backbone CNN to get the feature map (Feature size: 50, 50, 512). Besides test time efficiency, another key reason using an RPN as a proposal generator makes sense is the advantages of **weight sharing between the RPN backbone and the Fast R-CNN detector backbone**.
- Next, the bounding box proposals from the RPN are used to pool features from the backbone feature map. This is done by the ROI pooling layer. The ROI pooling layer, in essence, works by a) Taking the region corresponding to a proposal from the backbone feature map; b) Dividing this region into a fixed number of sub-windows; c) Performing max-pooling over these sub-windows to give a fixed size output. To understand the details of the ROI pooling layer and its advantages, read Fast R-CNN.
- The output from the ROI pooling layer has a size of (N, 7, 7, 512) where N is the number of proposals from the region proposal algorithm. After passing them through two fully connected layers, the features are fed into the sibling classification and regression branches.
- Note that these classification and detection branches are different from those of the RPN. Here the classification layer has C units for each of the classes in the

detection task (including a catch-all background class). The features are passed through a SoftMax layer to get the classification scores — the probability of a proposal belonging to each class. The regression layer coefficients are used to improve the predicted bounding boxes. Here the regressor is size agnostic, (unlike the RPN) but is specific to each class. That is, all the classes have individual regressors with 4 parameters each corresponding to C*4 output units in the regression layer.

4 – Step Alternating Training

In order to force the network to share the weights of the CNN backbone between the RPN and the detector, the authors use a 4-step training method:

a) The RPN is trained independently as described above. The backbone CNN for this task is initialized with weights from a network trained for an ImageNet classification task and is then fine-tuned for the region proposal task.

b) The Fast R-CNN detector network is also trained independently. The backbone CNN for this task is initialized with weights from a network trained for an ImageNet classification task and is then fine-tuned for the object detection task. The RPN weights are fixed and the proposals from the RPN are used to train the Faster R-CNN.

c) The RPN is now initialized with weights from this Faster R-CNN and fine-tuned for the region proposal task. This time, weights in the common layers between the RPN and detector remain fixed, and only the layers unique to the RPN are fine-tuned. This is the final RPN.

d) Once again using the new RPN, the Fast R-CNN detector is fine-tuned. Again, only the layers unique to the detector network are fine-tuned and the common layer weights are fixed.

Experiments

Dataset

The dataset chosen was Pascal VOC 2007.

1. It provides standardized image data sets for object class recognition.
2. Provides a common set of tools for accessing the data sets and annotations.
3. Enables evaluation and comparison of different methods.

It is fundamentally used for supervised learning tasks in that a training set of images along with annotations are provided. The twenty object classes are :

- Person: person
- Animal: bird, cat, cow, dog, horse, sheep
- Vehicle: aeroplane, bicycle, boat, bus, car, motorbike, train
- Indoor: bottle, chair, dining table, potted plant, sofa, tv/monitor

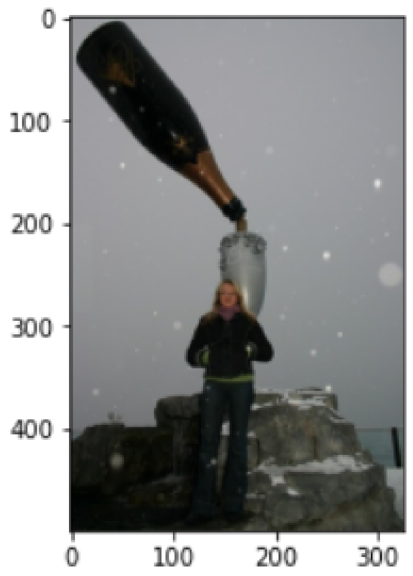
The training data provided consists of a set of images; each image has an annotation file giving a bounding box and object class label for each object in one of the twenty classes present in the image. Note that multiple objects from multiple classes may be present in the same image.

Data preparation

We have input images in the shape (H, W, C). We have the bounding box coordinates and the class of the object in the annotation file provided along with set of images. We use Computer vision library to read and view images. Let's look at the implementation with the example of one image,

```
#Diplay Input Image
input_img = cv2.imread(r'C:\Users\Owner\Downloads\000498.jpg')
input_img = cv2.cvtColor(input_img, cv2.COLOR_BGR2RGB)
print("Input Image Shape (H,W,F): ",input_img.shape)
plt.imshow(input_img)
plt.show()
```

```
Input Image Shape (H,W,F):  (500, 326, 3)
```

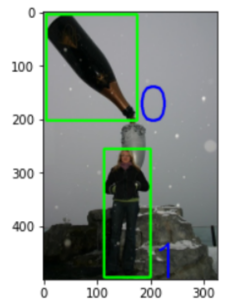


We have our sample image as (500, 326, 3) which shows that our image has 3 channels (i.e., RGB image) with height of 500 and width of 326. Then, we have the bounding box and object class label information from the .xml files,

```
#We have the bounding box information in annotations xml file
# Object information: a set of bounding boxes [ymin, xmin, ymax, xmax] and their labels
bounding_box = np.array([[5, 6, 203, 176], [256, 114, 496, 201]])
labels = np.array([0, 1]) # 0: bottle, 1: person
```

We then combine the bounding box coordinates, class labels with the sample image to see the placement of the bounding boxes.

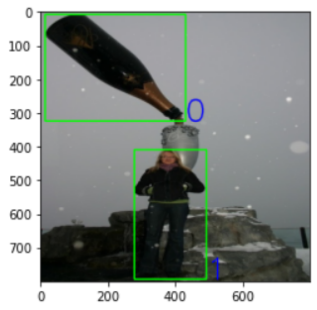
```
#Display bounding box and lables with respect to objects
img_clone = np.copy(input_img)
for i in range(len(bounding_box)):
    cv2.rectangle(img_clone, (bounding_box[i][1], bounding_box[i][0]), (bounding_box[i][3], bounding_box[i][2]), color=
    cv2.putText(img_clone, str(int(labels[i])), (bounding_box[i][3], bounding_box[i][2]), cv2.FONT_HERSHEY_SIMPLEX, 3,
plt.imshow(img_clone)
plt.show()
```



We can see that we have two objects in this image bottle belonging to label 0 and person belonging to label 1.

We resize the image to 800*800 pixels, in turn we need to reshape our bounding box.

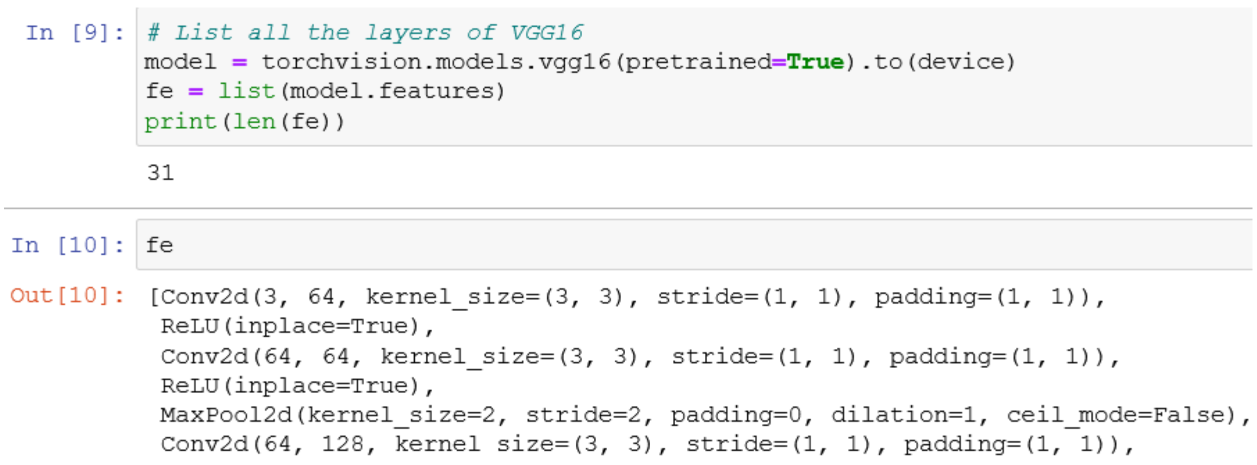
```
# display bounding box and labels after resized
img_clone = np.copy(img)
bbox_clone = bbox.astype(int)
for i in range(len(bbox)):
    cv2.rectangle(img_clone, (bbox[i][1], bbox[i][0]), (bbox[i][3], bbox[i][2]), color=(0, 255, 0), thickness=3) # Draw
    cv2.putText(img_clone, str(int(labels[i])), (bbox[i][3], bbox[i][2]), cv2.FONT_HERSHEY_SIMPLEX, 3, (0,0,255),thickne
plt.imshow(img_clone)
plt.show()
```



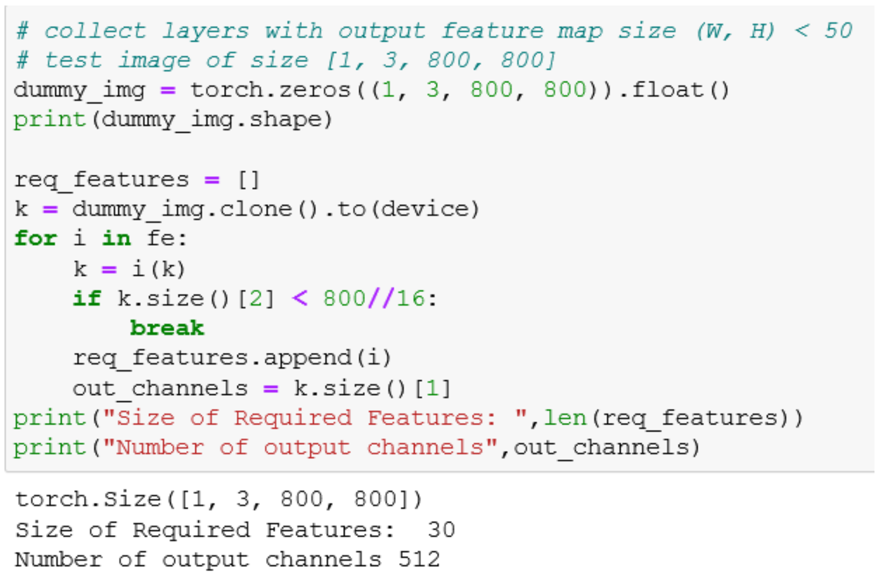
Feature Extraction

The VGG16 network is used as a feature extraction module here, this acts as a backbone for both the RPN network and Fast R-CNN network. Since, the input of the network is 800, the output of the feature extraction module should have feature map size of (800//16 = 50). So, we need to check where the VGG16 module is achieving this feature map size and trim the network accordingly. This can be done in the following way,

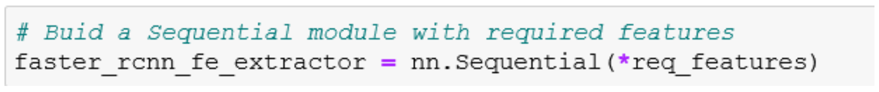
- Create a dummy image and set the volatile to be False.
- List all the layers of the VGG16
- Pass the image through the layers and subset the list when the output size of the image (feature map) is below the required level (800//16)
- Convert this list into a Sequential module.



```
In [9]:  # List all the layers of VGG16
         model = torchvision.models.vgg16(pretrained=True).to(device)
         fe = list(model.features)
         print(len(fe))

         31

In [10]: fe

Out[10]: [Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)),
          ReLU(inplace=True),
          Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)),
          ReLU(inplace=True),
          MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False),
          Conv2d(64, 128, kernel size=(3, 3), stride=(1, 1), padding=(1, 1)),
```



```
# collect layers with output feature map size (W, H) < 50
# test image of size [1, 3, 800, 800]
dummy_img = torch.zeros((1, 3, 800, 800)).float()
print(dummy_img.shape)

req_features = []
k = dummy_img.clone().to(device)
for i in fe:
    k = i(k)
    if k.size()[2] < 800//16:
        break
    req_features.append(i)
    out_channels = k.size()[1]
print("Size of Required Features: ",len(req_features))
print("Number of output channels",out_channels)

torch.Size([1, 3, 800, 800])
Size of Required Features:  30
Number of output channels 512
```



```
# Buid a Sequential module with required features
faster_rcnn_fe_extractor = nn.Sequential(*req_features)
```

Now this **faster_rcnn_fe_extractor** can be used as our backend to compute features.

```
# FatureMap  is created with vgg16 sequential model
transform = transforms.Compose([transforms.ToTensor()]) # Defing PyTorch Transform
imgTensor = transform(img).to(device)
imgTensor = imgTensor.unsqueeze(0)
out_map = faster_rcnn_fe_extractor(imgTensor)
print("Featuremap Output size:",out_map.size())
```

```
Featuremap Output size: torch.Size([1, 512, 50, 50])
```

Anchor Boxes

Our steps to create anchor boxes are:

1. Generate anchor at a feature map location.
2. Generate anchor at all the feature map location.
3. Assign the labels and location of objects (with respect to the anchor) to every anchor.
4. Generate anchor at a feature map location.

- We will use anchor scales of 8, 16, 32 and ratios of 0.5, 1, 2 and sub-sampling of 16 (since we have pooled our image from 800*800 to 50*50). Now every pixel in the output feature maps corresponding 16*16 pixels in the image.
- We need to generate anchor boxes on top of this 16 * 16 pixels first and similarly do along x-axis and y-axis to get all the anchor boxes. This is done in the step 2.
- At each pixel location on the feature map, we need to generate 9 anchor boxes (number of anchor scales and number of ratios) and each anchor box will have 'y1', 'x1', 'y2', 'x2'. So, at each location anchor will have a shape of (9, 4). Let us begin with an empty array filled with zero values.

```
# Generate a total of  50*50*9 = 2500*9 = 22500 anchorboxes on an image.
#for an image of 50x50 2500 anchors on each anchor 9 anchor boxes(3ratios * 3Scales)
ratios = [0.5, 1, 2]
scales = [8, 16, 32]
sub_sample = 16
# x, y intervals to generate anchor box center
fe_size = (800//16)
ctr_x = np.arange(16, (fe_size+1) * 16, 16)
ctr_y = np.arange(16, (fe_size+1) * 16, 16)
print("CenterX:\n", len(ctr_x), ctr_x)
print("CenterY:\n", len(ctr_y), ctr_y)
```

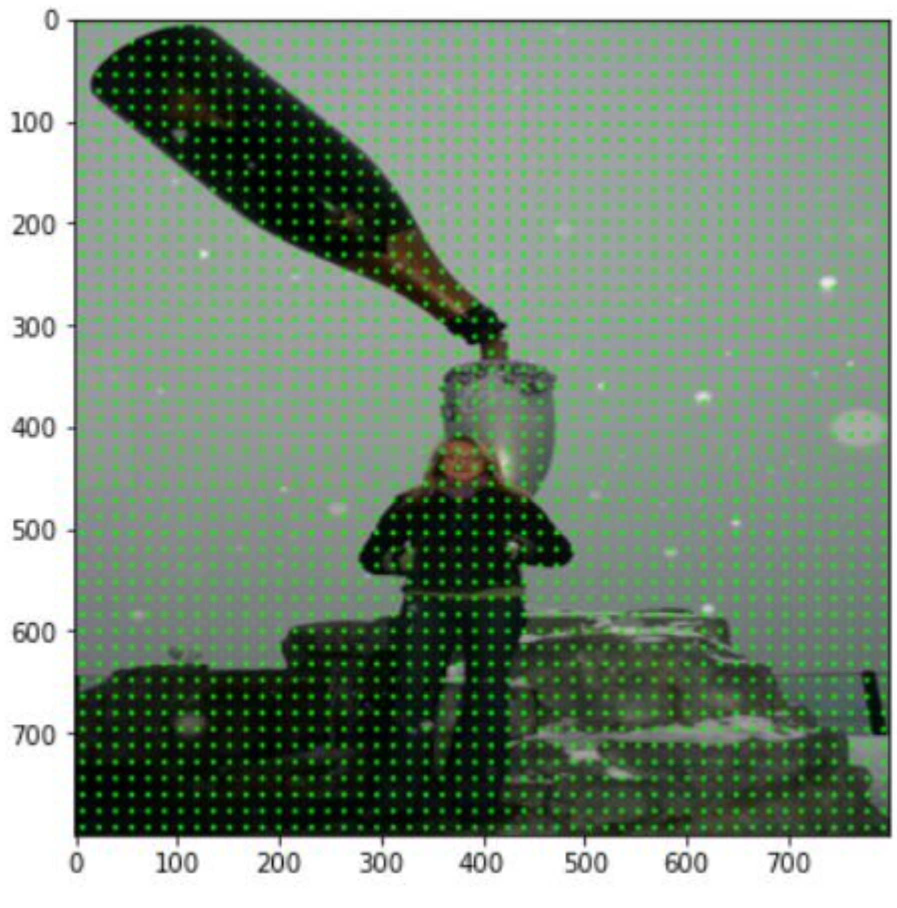
```
CenterX:
 50 [ 16  32  48  64  80  96 112 128 144 160 176 192 208 224 240 256 272 288
 304 320 336 352 368 384 400 416 432 448 464 480 496 512 528 544 560 576
 592 608 624 640 656 672 688 704 720 736 752 768 784 800]
CenterY:
 50 [ 16  32  48  64  80  96 112 128 144 160 176 192 208 224 240 256 272 288
 304 320 336 352 368 384 400 416 432 448 464 480 496 512 528 544 560 576
 592 608 624 640 656 672 688 704 720 736 752 768 784 800]
```

The coordinates of the center points to generate anchor boxes are generated looping through CenterX and CenterY.

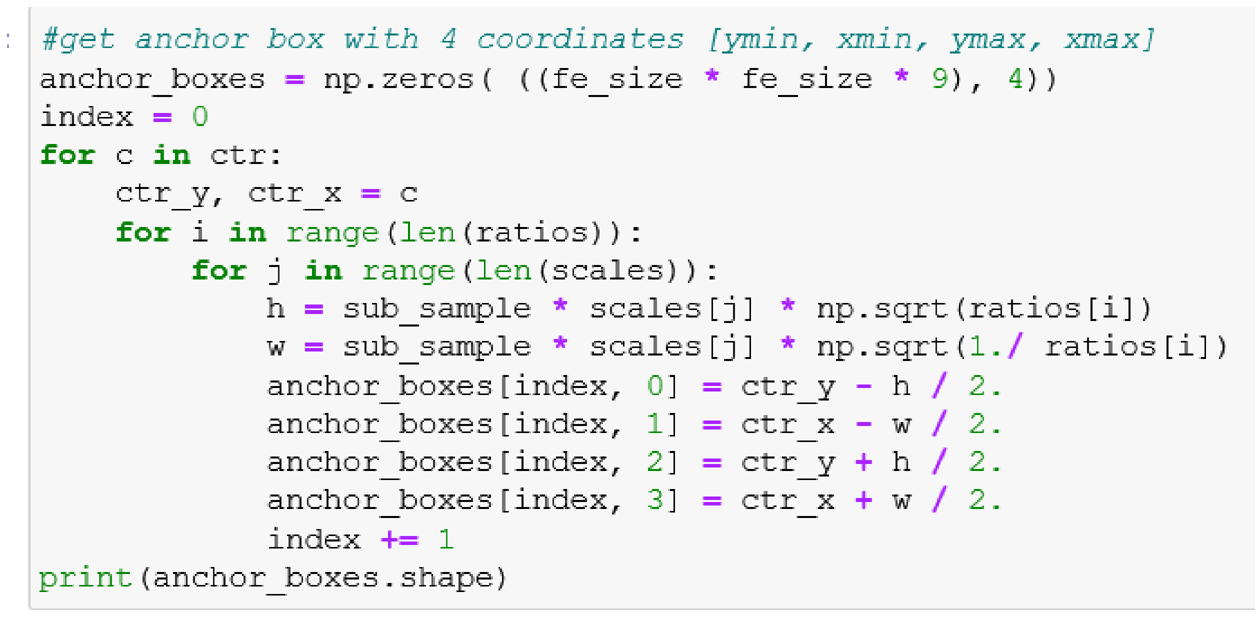
```
# coordinates of the 2500 center points to generate anchor boxes
index = 0
ctr = np.zeros((2500, 2))
for x in range(len(ctr_x)):
    for y in range(len(ctr_y)):
        ctr[index, 1] = ctr_x[x] - 8
        ctr[index, 0] = ctr_y[y] - 8
        index +=1
print(ctr.shape)
```

```
(2500, 2)
```

After generating the anchor points our image would look like,



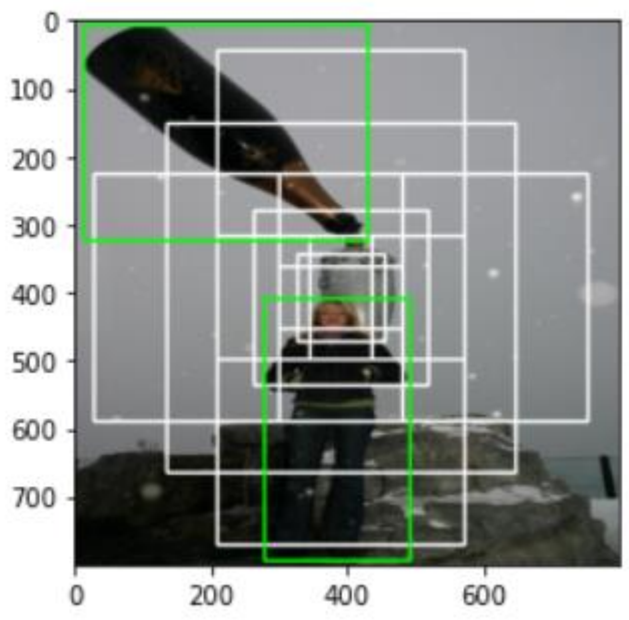
We create anchor boxes at each anchor points and a sample of set of anchor boxes generate at the center of the image is shown below.



```
#get anchor box with 4 coordinates [ymin, xmin, ymax, xmax]
anchor_boxes = np.zeros( ((fe_size * fe_size * 9), 4))
index = 0
for c in ctr:
    ctr_y, ctr_x = c
    for i in range(len(ratios)):
        for j in range(len(scales)):
            h = sub_sample * scales[j] * np.sqrt(ratios[i])
            w = sub_sample * scales[j] * np.sqrt(1./ ratios[i])
            anchor_boxes[index, 0] = ctr_y - h / 2.
            anchor_boxes[index, 1] = ctr_x - w / 2.
            anchor_boxes[index, 2] = ctr_y + h / 2.
            anchor_boxes[index, 3] = ctr_x + w / 2.
            index += 1
print(anchor_boxes.shape)
```

```
(22500, 4)
```

The above output shows us that 22500 (50*50*9) anchor boxes have been created with coordinates of shape 4 representing the y_min, x_min, y_max, x_max.



Valid anchor boxes are boxes with coordinates which has values between 0 and 800 and we have filtered out the invalid anchor boxes by this condition.

Assign the labels and location of objects (with respect to the anchor) to each and every anchor. Now since we have generated all the anchor boxes, we need to look at the objects inside the image and assign them to the specific anchor boxes which contain them. Faster R-CNN has some guidelines to assign labels to the anchor boxes.

We assign a positive label to two kind of anchors

a) The anchor/anchors with the highest Intersection-over-Union (IoU) overlap with a ground-truth-box or

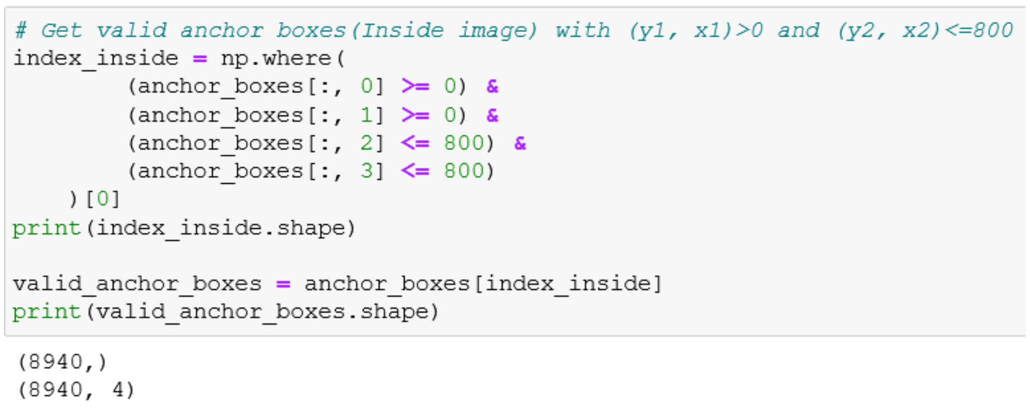
b) An anchor that has an IoU overlap higher than 0.7 with ground-truth box. Note that single ground-truth object may assign positive labels to multiple anchors.

c) We assign a negative label to a non-positive anchor if its IoU ratio is lower than 0.3 for all ground-truth boxes.

d) Anchors that are neither positive nor negative do not contribute to the training objective.

We will assign the labels and locations for the anchor boxes in the following ways.

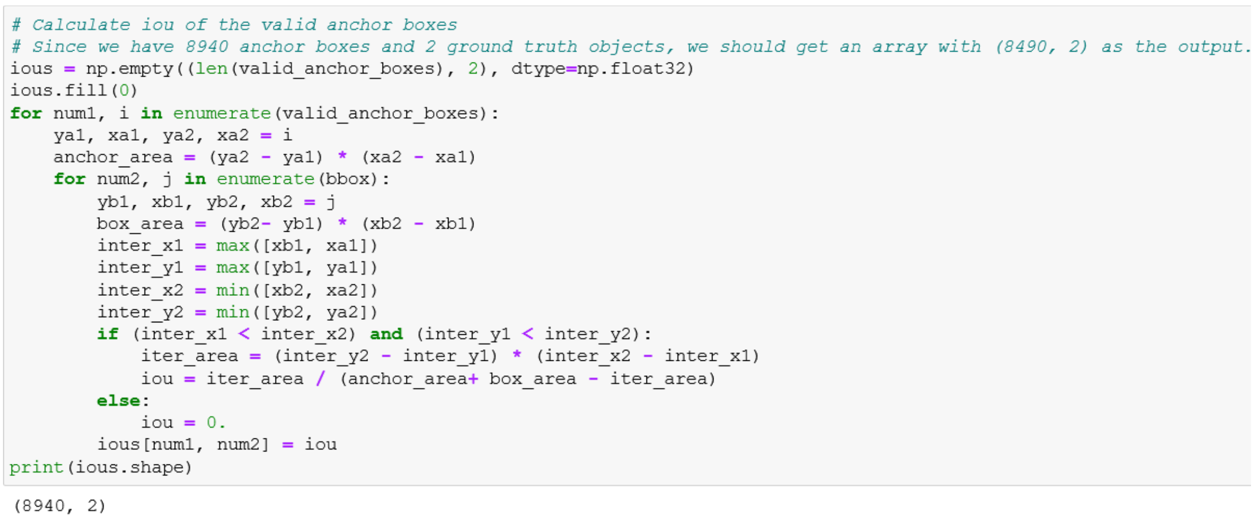
- Find the indexes of valid anchor boxes and create an array with these indices. Create a label array with shape index array filled with -1.
- Check whether one of the above condition a, b, c is satisfying or not and fill the label accordingly. In case of positive anchor box (label is 1), Note which ground truth object has resulted in this.
- Calculate the locations (**loc**) of ground truth associated with the anchor box with respect to the anchor box.
- Reorganize all anchor boxes by filling with -1 for all invalid anchor boxes and values we have calculated for all valid anchor boxes.
- Outputs should be labels with (N, 1) array and locations with (N, 4) array.
- Find the index of all valid anchor boxes.



```
# Get valid anchor boxes(Inside image) with (y1, x1)>0 and (y2, x2)<=800
index_inside = np.where(
        (anchor_boxes[:, 0] >= 0) &
        (anchor_boxes[:, 1] >= 0) &
        (anchor_boxes[:, 2] <= 800) &
        (anchor_boxes[:, 3] <= 800)
    )[0]
print(index_inside.shape)

valid_anchor_boxes = anchor_boxes[index_inside]
print(valid_anchor_boxes.shape)
```

```
(8940,)
(8940, 4)
```

Since we have 2 objects in our image, we calculate IoU for those 2 objects from our valid anchor boxes.



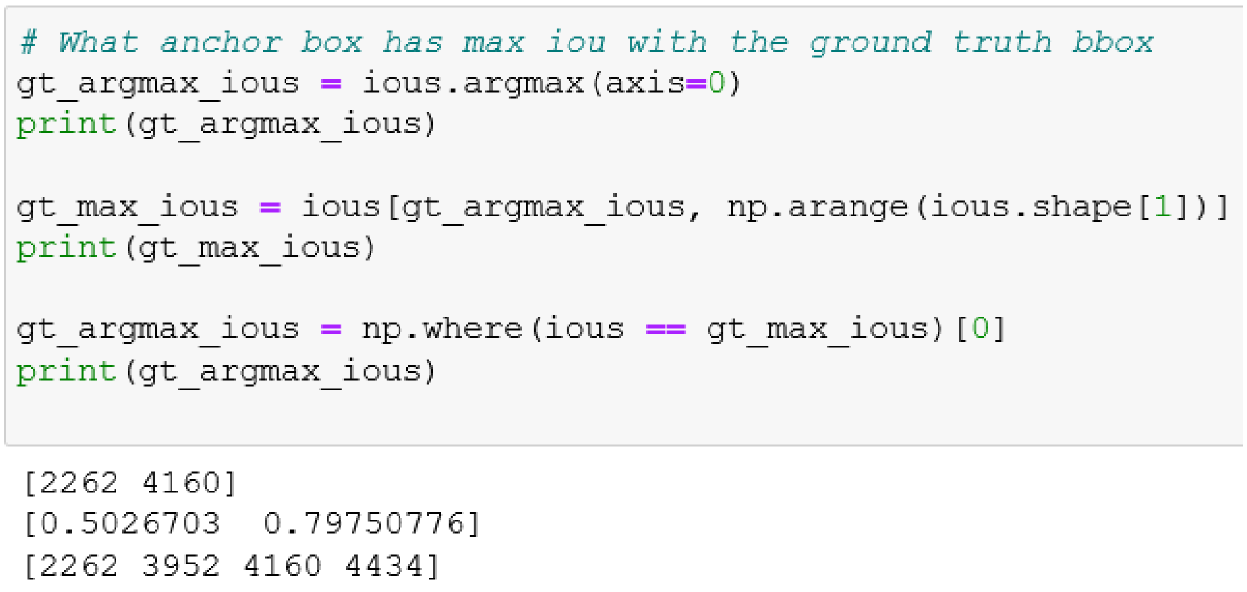
```
# Calculate iou of the valid anchor boxes
# Since we have 8940 anchor boxes and 2 ground truth objects, we should get an array with (8490, 2) as the output.
ious = np.empty((len(valid_anchor_boxes), 2), dtype=np.float32)
ious.fill(0)
for num1, i in enumerate(valid_anchor_boxes):
    ya1, xa1, ya2, xa2 = i
    anchor_area = (ya2 - ya1) * (xa2 - xa1)
    for num2, j in enumerate(bbox):
        yb1, xb1, yb2, xb2 = j
        box_area = (yb2- yb1) * (xb2 - xb1)
        inter_x1 = max([xb1, xa1])
        inter_y1 = max([yb1, ya1])
        inter_x2 = min([xb2, xa2])
        inter_y2 = min([yb2, ya2])
        if (inter_x1 < inter_x2) and (inter_y1 < inter_y2):
            iter_area = (inter_y2 - inter_y1) * (inter_x2 - inter_x1)
            iou = iter_area / (anchor_area+ box_area - iter_area)
        else:
            iou = 0.
        ious[num1, num2] = iou
print(ious.shape)
```

```
(8940, 2)
```

Considering the scenarios of a and b, we need to find two things here

- the highest IoU for each gt_box and its corresponding anchor box
- the highest IoU for each anchor box and its corresponding ground truth box

CASE - I



```
# What anchor box has max iou with the ground truth bbox
gt_argmax_ious = ious.argmax(axis=0)
print(gt_argmax_ious)

gt_max_ious = ious[gt_argmax_ious, np.arange(ious.shape[1])]
print(gt_max_ious)

gt_argmax_ious = np.where(ious == gt_max_ious)[0]
print(gt_argmax_ious)
```

```
[2262 4160]
[0.5026703  0.79750776]
[2262 3952 4160 4434]
```

CASE – II

```
# What ground truth bbox is associated with each anchor box
argmax_ious = ious.argmax(axis=1)
print(argmax_ious.shape)
print(argmax_ious)
max_ious = ious[np.arange(len(index_inside)), argmax_ious]
print(max_ious)
```

```
(8940,)
[0 0 0 ... 0 0 0]
[0.11933114 0.11933114 0.11933114 ... 0.         0.         0.        ]
```

Now we have three arrays,

- argmax_ious — Tells which ground truth object has max iou with each anchor.
- max_ious — Tells the max_iou with ground truth object with each anchor.
- gt_argmax_ious — Tells the anchors with the highest Intersection-over-Union (IoU) overlap with a ground-truth box.

Using argmax_ious and max_ious we can assign labels and locations to anchor boxes which satisify [b] and [c]. Using gt_argmax_ious we can assign labels and locations to anchor boxes which satisify [a].

Let’s put thresholds to some variables.

- Assign negative label (0) to all the anchor boxes which have max_iou less than negative threshold [c].
- Assign positive label (1) to all the anchor boxes which have highest IoU overlap with a ground-truth box [a]
- Assign positive label (1) to all the anchor boxes which have max_iou greater than positive threshold [b]

```
#Assign 1 to Object
# Use iou to assign 1 (objects) to two kind of anchors
# 1. The anchors with the highest iou overlap with a ground-truth-box
# 2. An anchor that has an IoU overlap higher than 0.7 with ground-truth box
pos_iou_threshold  = 0.7
neg_iou_threshold = 0.3
label[gt_argmax_ious] = 1
label[max_ious >= pos_iou_threshold] = 1

#Assign 0 to Background
#anchor if its IoU ratio is lower than 0.3 for all ground-truth boxes
label[max_ious < neg_iou_threshold] = 0
```

Training RPN

The Faster R-CNN paper phrases as follows Each mini-batch arises from a single image that contains many positive and negative example anchors, but this will bias towards negative samples as they are dominate. Instead, we randomly sample 256 anchors in an image to compute the loss function of a mini-batch, where the sampled positive and negative anchors have a ratio of up to 1:1. If there are fewer than 128 positive samples in an image, we pad the mini-batch with negative ones.

Now we need to randomly sample **n_pos** samples from the positive labels and ignore (-1) the remaining ones. In some cases we get less than **n_pos** samples, in that we will randomly sample (n_sample — n_pos) negitive samples (0) and assign ignore label to the remaining anchor boxes. This is done using the following code for positive and negative samples,

```
n_sample = 256
pos_ratio = 0.5
n_pos = pos_ratio * n_sample

pos_index = np.where(label == 1)[0]
if len(pos_index) > n_pos:
    disable_index = np.random.choice(pos_index, size=(len(pos_index) - n_pos), replace=False)
    label[disable_index] = -1

n_neg = n_sample * np.sum(label == 1)
neg_index = np.where(label == 0)[0]
if len(neg_index) > n_neg:
    disable_index = np.random.choice(neg_index, size=(len(neg_index) - n_neg), replace = False)
    label[disable_index] = -1
```

Assigning locations to anchor boxes

Now let's assign the locations to each anchor box with the ground truth object which has maximum iou. Note, we will assign anchor locations to all the valid anchor boxes irrespective of its label, later when we are calculating the losses, we can remove them with simple filters.

We already know which ground truth object has high iou with each anchor box, Now we need to find the locations of ground truth with respect to the anchor box location.

```
anchor_labels = np.empty((len(anchor_boxes),), dtype=label.dtype)
anchor_labels.fill(-1)
anchor_labels[index_inside] = label
print(anchor_labels.shape)

anchor_locations = np.empty((len(anchor_boxes),) + anchor_boxes.shape[1:], dtype=anchor_locs.dtype)
anchor_locations.fill(0)
anchor_locations[index_inside, :] = anchor_locs
print(anchor_locations.shape)
```

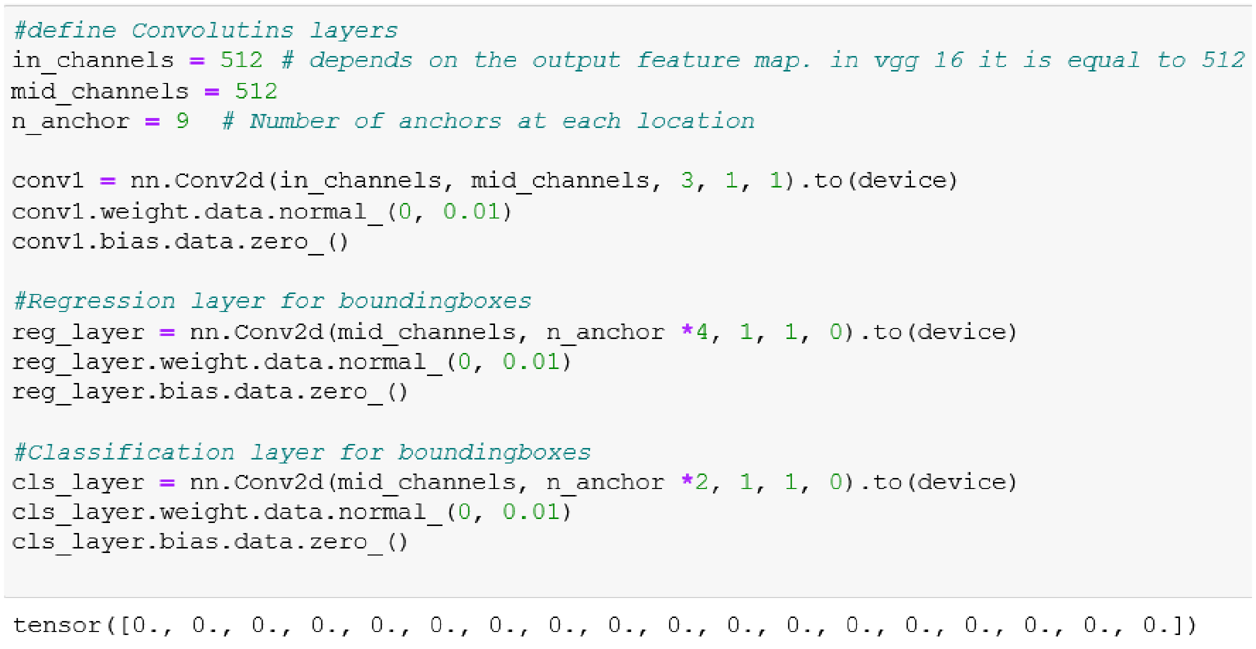
```
(22500,)
(22500, 4)
```

To generate region proposals, we slide a small network over the convolutional feature map output that we obtained in the feature extraction module. This small network takes as input an n x n spatial window of the input convolutional feature map. Each sliding window is mapped to a lower-dimensional feature [512 features]. This feature is fed into two sibling fully connected layers

- A box regression layer
- A box classification layer

we use n=3, as noted in Faster R-CNN paper. We can implement this Architecture using n x n convolutional layer followed by two siblings 1 x 1 convolutional layers.

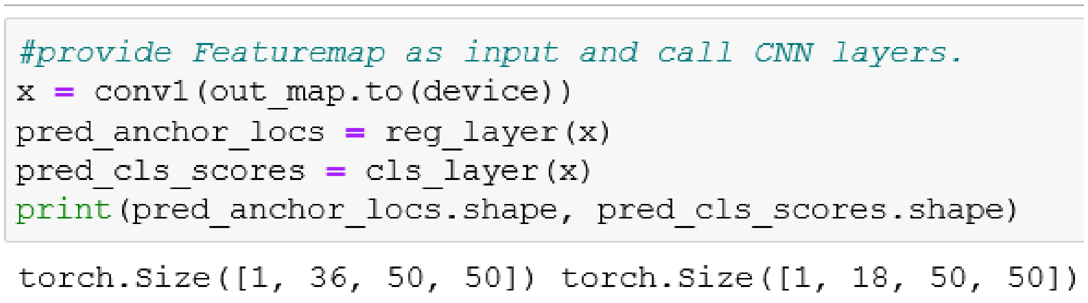
The paper tells that they initialized these layers with zero mean and 0.01 standard deviation for weights and zeros for base. Let’s do that.



```
#define Convolutins layers
in_channels = 512 # depends on the output feature map. in vgg 16 it is equal to 512
mid_channels = 512
n_anchor = 9  # Number of anchors at each location

conv1 = nn.Conv2d(in_channels, mid_channels, 3, 1, 1).to(device)
conv1.weight.data.normal_(0, 0.01)
conv1.bias.data.zero_()

#Regression layer for boundingboxes
reg_layer = nn.Conv2d(mid_channels, n_anchor *4, 1, 1, 0).to(device)
reg_layer.weight.data.normal_(0, 0.01)
reg_layer.bias.data.zero_()

#Classification layer for boundingboxes
cls_layer = nn.Conv2d(mid_channels, n_anchor *2, 1, 1, 0).to(device)
cls_layer.weight.data.normal_(0, 0.01)
cls_layer.bias.data.zero_()
```

```
tensor([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

Now the outputs we got in the feature extraction state should be sent to this network to predict locations of objects with respect to the anchor and the objectness score associated with it.

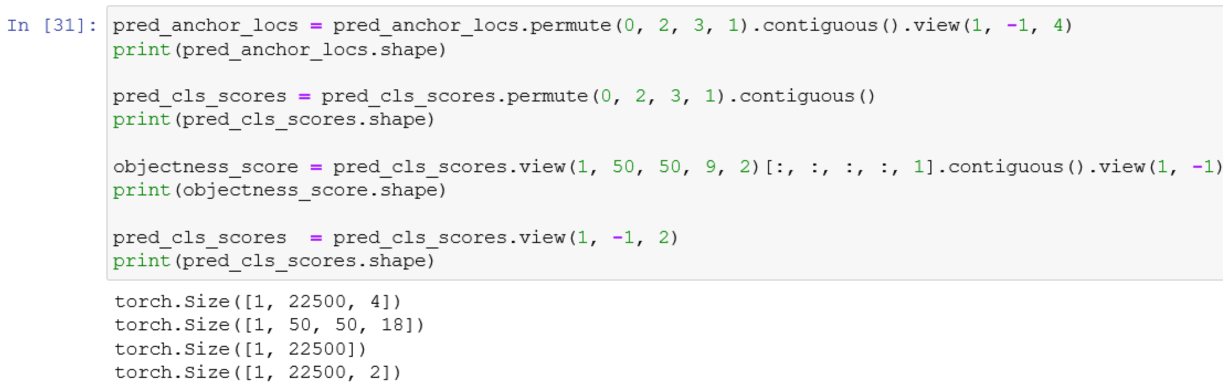


```
#provide Featuremap as input and call CNN layers.
x = conv1(out_map.to(device))
pred_anchor_locs = reg_layer(x)
pred_cls_scores = cls_layer(x)
print(pred_anchor_locs.shape, pred_cls_scores.shape)
```

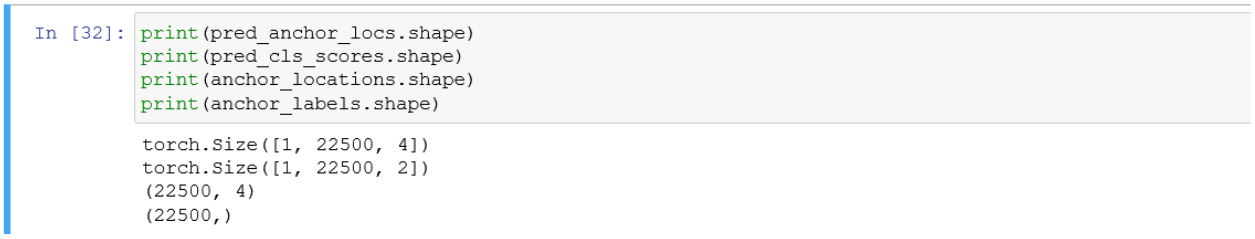
```
torch.Size([1, 36, 50, 50]) torch.Size([1, 18, 50, 50])
```

Lets reformat these a bit and make it align with our anchor targets we designed previously.

- pred_cls_scores and pred_anchor_locs are the output the RPN network and the losses to updates the weights
- pred_cls_scores and objectness_scores are used as inputs to the **proposal layer**, which generate a set of proposal which are further used by RoI network.



```
In [31]: pred_anchor_locs = pred_anchor_locs.permute(0, 2, 3, 1).contiguous().view(1, -1, 4)
         print(pred_anchor_locs.shape)

         pred_cls_scores = pred_cls_scores.permute(0, 2, 3, 1).contiguous()
         print(pred_cls_scores.shape)

         objectness_score = pred_cls_scores.view(1, 50, 50, 9, 2)[:, :, :, :, 1].contiguous().view(1, -1)
         print(objectness_score.shape)

         pred_cls_scores  = pred_cls_scores.view(1, -1, 2)
         print(pred_cls_scores.shape)

         torch.Size([1, 22500, 4])
         torch.Size([1, 50, 50, 18])
         torch.Size([1, 22500])
         torch.Size([1, 22500, 2])
```



```
In [32]: print(pred_anchor_locs.shape)
         print(pred_cls_scores.shape)
         print(anchor_locations.shape)
         print(anchor_labels.shape)

         torch.Size([1, 22500, 4])
         torch.Size([1, 22500, 2])
         (22500, 4)
         (22500,)
```

Region of Interest

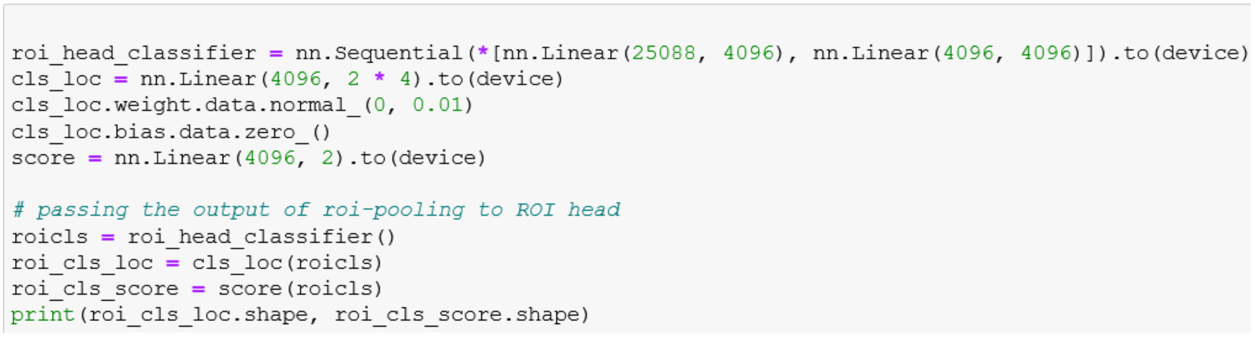
The Faster R-CNN says, RPN proposals highly overlap with each other. To reduced redundancy, we adopt non-maximum supression (NMS) on the proposal regions based on their cls scores. We fix the IoU threshold for NMS at 0.7, which leaves us about 2000 proposal regions per image. After an ablation study, the authors show that NMS does not harm the ultimate detection accuracy, but substantially reduces the number of proposals. After NMS, we use the top-N ranked proposal regions for detection.

The final region proposals is used as the input to the Fast R-CNN object which finally tries to predict the object locations (with respect to the proposed box) and class of the object (classification of each proposal) in this network. Then, we will determine the losses, We will calculate both the RPN loss and Fast R-CNN loss.

Feed forward layer

Now RPN output is feed as input to a classifier layer, which will further brach out to a classification head and regression in the defined network. roi_cls_loc and roi_cls_score are two ouput tensors from which we can get actual bounding boxes. We will compute the losses for both the RPN and Fast RCNN networks. This will complete the Faster R-CNN implementation.

Feed forward layer



```
roi_head_classifier = nn.Sequential(*[nn.Linear(25088, 4096), nn.Linear(4096, 4096)]).to(device)
cls_loc = nn.Linear(4096, 2 * 4).to(device)
cls_loc.weight.data.normal_(0, 0.01)
cls_loc.bias.data.zero_()
score = nn.Linear(4096, 2).to(device)

# passing the output of roi-pooling to ROI head
roicls = roi_head_classifier()
roi_cls_loc = cls_loc(roicls)
roi_cls_score = score(roicls)
print(roi_cls_loc.shape, roi_cls_score.shape)
```

Conclusion

This is a very intricate architecture to implement. We learnt the important aspects of object detection like ground truth, anchor points, anchors, anchor boxes, intersection over union, Region proposal network, Region of interest pooling.

References

1. Prakhar Ganesh – 2019/08/12 - Object detection: Simplified - https://towardsdatascience.com/object-detection-simplified-e07aa3830954
2. Shilpa Ananth – 2019/09/09 – Faster R-CNN for Object Detection - https://towardsdatascience.com/faster-r-cnn-for-object-detection-a-technical-summary-474c5b857b46
3. Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun - Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks - https://arxiv.org/pdf/1506.01497.pdf
4. Tanay Karmakar – Region Proposal Network – Backbone of Faster R-CNN - https://medium.com/egen/region-proposal-network-rpn-backbone-of-faster-r-cnn-4a744a38d7f9
5. Faster R-CNN – Using RPN for Object detection - https://www.alegion.com/faster-r-cnn
6. Blogspace - https://blog.paperspace.com/faster-r-cnn-explained-object-detection/
7. Medium - https://medium.com/@fractaldle/guide-to-build-faster-rcnn-in-pytorch-95b10c273439
8. Trylolabs - https://tryolabs.com/blog/2018/01/18/faster-r-cnn-down-the-rabbit-hole-of-modern-object-detection/
9. Deepsense.ai - https://blog.deepsense.ai/region-of-interest-pooling-explained/
10. ROI Pooling - https://blog.deepsense.ai/region-of-interest-pooling-explained/