def make\_yolov3\_model():

    input\_image = Input(shape=(None, None, 3))

    # Layer  0 => 4

    x = \_conv\_block(input\_image, [{'filter': 32, 'kernel': 3, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer\_idx': 0},

                                  {'filter': 64, 'kernel': 3, 'stride': 2, 'bnorm': True, 'leaky': True, 'layer\_idx': 1},

                                  {'filter': 32, 'kernel': 1, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer\_idx': 2},

                                  {'filter': 64, 'kernel': 3, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer\_idx': 3}])

    # Layer  5 => 8

    x = \_conv\_block(x, [{'filter': 128, 'kernel': 3, 'stride': 2, 'bnorm': True, 'leaky': True, 'layer\_idx': 5},

                        {'filter':  64, 'kernel': 1, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer\_idx': 6},

                        {'filter': 128, 'kernel': 3, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer\_idx': 7}])

    # Layer  9 => 11

    x = \_conv\_block(x, [{'filter':  64, 'kernel': 1, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer\_idx': 9},

                        {'filter': 128, 'kernel': 3, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer\_idx': 10}])

    # Layer 12 => 15

    x = \_conv\_block(x, [{'filter': 256, 'kernel': 3, 'stride': 2, 'bnorm': True, 'leaky': True, 'layer\_idx': 12},

                        {'filter': 128, 'kernel': 1, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer\_idx': 13},

                        {'filter': 256, 'kernel': 3, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer\_idx': 14}])

    # Layer 16 => 36

    for i in range(7):

        x = \_conv\_block(x, [{'filter': 128, 'kernel': 1, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer\_idx': 16+i\*3},

                            {'filter': 256, 'kernel': 3, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer\_idx': 17+i\*3}])

    skip\_36 = x

    # Layer 37 => 40

    x = \_conv\_block(x, [{'filter': 512, 'kernel': 3, 'stride': 2, 'bnorm': True, 'leaky': True, 'layer\_idx': 37},

                        {'filter': 256, 'kernel': 1, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer\_idx': 38},

                        {'filter': 512, 'kernel': 3, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer\_idx': 39}])

    # Layer 41 => 61

    for i in range(7):

        x = \_conv\_block(x, [{'filter': 256, 'kernel': 1, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer\_idx': 41+i\*3},

                            {'filter': 512, 'kernel': 3, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer\_idx': 42+i\*3}])

    skip\_61 = x

    # Layer 62 => 65

    x = \_conv\_block(x, [{'filter': 1024, 'kernel': 3, 'stride': 2, 'bnorm': True, 'leaky': True, 'layer\_idx': 62},

                        {'filter':  512, 'kernel': 1, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer\_idx': 63},

                        {'filter': 1024, 'kernel': 3, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer\_idx': 64}])

    # Layer 66 => 74

    for i in range(3):

        x = \_conv\_block(x, [{'filter':  512, 'kernel': 1, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer\_idx': 66+i\*3},

                            {'filter': 1024, 'kernel': 3, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer\_idx': 67+i\*3}])

    # Layer 75 => 79

    x = \_conv\_block(x, [{'filter':  512, 'kernel': 1, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer\_idx': 75},

                        {'filter': 1024, 'kernel': 3, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer\_idx': 76},

                        {'filter':  512, 'kernel': 1, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer\_idx': 77},

                        {'filter': 1024, 'kernel': 3, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer\_idx': 78},

                        {'filter':  512, 'kernel': 1, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer\_idx': 79}], skip=False)

    # Layer 80 => 82

    yolo\_82 = \_conv\_block(x, [{'filter': 1024, 'kernel': 3, 'stride': 1, 'bnorm': True,  'leaky': True,  'layer\_idx': 80},

                              {'filter':  255, 'kernel': 1, 'stride': 1, 'bnorm': False, 'leaky': False, 'layer\_idx': 81}], skip=False)

    # Layer 83 => 86

    x = \_conv\_block(x, [{'filter': 256, 'kernel': 1, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer\_idx': 84}], skip=False)

    x = UpSampling2D(2)(x)

    x = concatenate([x, skip\_61])

    # Layer 87 => 91

    x = \_conv\_block(x, [{'filter': 256, 'kernel': 1, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer\_idx': 87},

                        {'filter': 512, 'kernel': 3, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer\_idx': 88},

                        {'filter': 256, 'kernel': 1, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer\_idx': 89},

                        {'filter': 512, 'kernel': 3, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer\_idx': 90},

                        {'filter': 256, 'kernel': 1, 'stride': 1, 'bnorm': True, 'leaky': True, 'layer\_idx': 91}], skip=False)

    # Layer 92 => 94

    yolo\_94 = \_conv\_block(x, [{'filter': 512, 'kernel': 3, 'stride': 1, 'bnorm': True,  'leaky': True,  'layer\_idx': 92},

                              {'filter': 255, 'kernel': 1, 'stride': 1, 'bnorm': False, 'leaky': False, 'layer\_idx': 93}], skip=False)

    # Layer 95 => 98

    x = \_conv\_block(x, [{'filter': 128, 'kernel': 1, 'stride': 1, 'bnorm': True, 'leaky': True,   'layer\_idx': 96}], skip=False)

    x = UpSampling2D(2)(x)

    x = concatenate([x, skip\_36])

    # Layer 99 => 106

    yolo\_106 = \_conv\_block(x, [{'filter': 128, 'kernel': 1, 'stride': 1, 'bnorm': True,  'leaky': True,  'layer\_idx': 99},

                               {'filter': 256, 'kernel': 3, 'stride': 1, 'bnorm': True,  'leaky': True,  'layer\_idx': 100},

                               {'filter': 128, 'kernel': 1, 'stride': 1, 'bnorm': True,  'leaky': True,  'layer\_idx': 101},

                               {'filter': 256, 'kernel': 3, 'stride': 1, 'bnorm': True,  'leaky': True,  'layer\_idx': 102},

                               {'filter': 128, 'kernel': 1, 'stride': 1, 'bnorm': True,  'leaky': True,  'layer\_idx': 103},

                               {'filter': 256, 'kernel': 3, 'stride': 1, 'bnorm': True,  'leaky': True,  'layer\_idx': 104},

                               {'filter': 255, 'kernel': 1, 'stride': 1, 'bnorm': False, 'leaky': False, 'layer\_idx': 105}], skip=False)

    model = Model(input\_image, [yolo\_82, yolo\_94, yolo\_106])

    return model

class WeightReader:

    def \_\_init\_\_(self, weight\_file):

        with open(weight\_file, 'rb') as w\_f:

            major,    = struct.unpack('i', w\_f.read(4))

            minor,    = struct.unpack('i', w\_f.read(4))

            revision, = struct.unpack('i', w\_f.read(4))

            if (major\*10 + minor) >= 2 and major < 1000 and minor < 1000:

                w\_f.read(8)

            else:

                w\_f.read(4)

            transpose = (major > 1000) or (minor > 1000)

            binary = w\_f.read()

        self.offset = 0

        self.all\_weights = np.frombuffer(binary, dtype='float32')

    def read\_bytes(self, size):

        self.offset = self.offset + size

        return self.all\_weights[self.offset-size:self.offset]

    def load\_weights(self, model):

        for i in range(106):

            try:

                conv\_layer = model.get\_layer('conv\_' + str(i))

                print("loading weights of convolution #" + str(i))

                if i not in [81, 93, 105]:

                    norm\_layer = model.get\_layer('bnorm\_' + str(i))

                    size = np.prod(norm\_layer.get\_weights()[0].shape)

                    beta  = self.read\_bytes(size) # bias

                    gamma = self.read\_bytes(size) # scale

                    mean  = self.read\_bytes(size) # mean

                    var   = self.read\_bytes(size) # variance

                    weights = norm\_layer.set\_weights([gamma, beta, mean, var])

                if len(conv\_layer.get\_weights()) > 1:

                    bias   = self.read\_bytes(np.prod(conv\_layer.get\_weights()[1].shape))

                    kernel = self.read\_bytes(np.prod(conv\_layer.get\_weights()[0].shape))

                    kernel = kernel.reshape(list(reversed(conv\_layer.get\_weights()[0].shape)))

                    kernel = kernel.transpose([2,3,1,0])

                    conv\_layer.set\_weights([kernel, bias])

                else:

                    kernel = self.read\_bytes(np.prod(conv\_layer.get\_weights()[0].shape))

                    kernel = kernel.reshape(list(reversed(conv\_layer.get\_weights()[0].shape)))

                    kernel = kernel.transpose([2,3,1,0])

                    conv\_layer.set\_weights([kernel])

            except ValueError:

                print("no convolution #" + str(i))

    def reset(self):

        self.offset = 0

from google.colab import drive

from google.colab import files

drive.mount('/content/drive')

model = make\_yolov3\_model()

# load the model weights

weight\_reader = WeightReader('/content/drive/MyDrive/yolov2.weights')

# set the model weights into the model

weight\_reader.load\_weights(model)

model.summary()

def load\_image\_pixels(filename, shape):

    image = load\_img(filename)

    width, height = image.size

    image = load\_img(filename, target\_size=shape)

    image = img\_to\_array(image)

    image = image.astype('float32')

    image = image/255.0

    image = np.expand\_dims(image, 0)

    return image, width, height

input\_w, input\_h = 416, 416

image\_path = '/content/drive/MyDrive/Dataset/Car Images/Test Images/Acura TL Sedan 2012/00314.jpg'

image, image\_w, image\_h = load\_image\_pixels(image\_path, (input\_w, input\_h))

y\_pred = model.predict(image)

print([a.shape for a in y\_pred])

class BoundBox:

    def \_\_init\_\_(self, xmin, ymin, xmax, ymax, objness = None, classes = None):

        self.xmin = xmin

        self.ymin = ymin

        self.xmax = xmax

        self.ymax = ymax

        self.objness = objness

        self.classes = classes

        self.label = -1

        self.score = -1

    def get\_label(self):

        if self.label == -1:

            self.label = np.argmax(self.classes)

        return self.label

    def get\_score(self):

        if self.score == -1:

            self.score = self.classes[self.get\_label()]

        return self.score

def \_sigmoid(x):

    return 1. / (1. + np.exp(-x))

def decode\_netout(netout, anchors, obj\_thresh, net\_h, net\_w):

    grid\_h, grid\_w = netout.shape[:2]

    nb\_box = 3

    netout = netout.reshape((grid\_h, grid\_w, nb\_box, -1))

    nb\_class = netout.shape[-1] - 5

    boxes = []

    netout[..., :2]  = \_sigmoid(netout[..., :2])

    netout[..., 4:]  = \_sigmoid(netout[..., 4:])

    netout[..., 5:]  = netout[..., 4][..., np.newaxis] \* netout[..., 5:]

    netout[..., 5:] \*= netout[..., 5:] > obj\_thresh

    for i in range(grid\_h\*grid\_w):

        row = i / grid\_w

        col = i % grid\_w

        for b in range(nb\_box):

            # 4th element is objectness score

            objectness = netout[int(row)][int(col)][b][4]

            #objectness = netout[..., :4]

            if(objectness.all() <= obj\_thresh): continue

            # first 4 elements are x, y, w, and h

            x, y, w, h = netout[int(row)][int(col)][b][:4]

            x = (col + x) / grid\_w # center position, unit: image width

            y = (row + y) / grid\_h # center position, unit: image height

            w = anchors[2 \* b + 0] \* np.exp(w) / net\_w # unit: image width

            h = anchors[2 \* b + 1] \* np.exp(h) / net\_h # unit: image height

            # last elements are class probabilities

            classes = netout[int(row)][col][b][5:]

            #classes = 2

            box = BoundBox(x-w/2, y-h/2, x+w/2, y+h/2, objectness, classes)

            #box = BoundBox(x-w/2, y-h/2, x+w/2, y+h/2, None, classes)

            boxes.append(box)

    return boxes

def correct\_yolo\_boxes(boxes, image\_h, image\_w, net\_h, net\_w):

    if (float(net\_w)/image\_w) < (float(net\_h)/image\_h):

        new\_w = net\_w

        new\_h = (image\_h\*net\_w)/image\_w

    else:

        new\_h = net\_w

        new\_w = (image\_w\*net\_h)/image\_h

    for i in range(len(boxes)):

        x\_offset, x\_scale = (net\_w - new\_w)/2./net\_w, float(new\_w)/net\_w

        y\_offset, y\_scale = (net\_h - new\_h)/2./net\_h, float(new\_h)/net\_h

        boxes[i].xmin = 0

        boxes[i].xmax = 0

        boxes[i].ymin = 0

        boxes[i].ymax = 0

        boxes[i].xmin = math.isnan(int((boxes[i].xmin - x\_offset) / x\_scale \* image\_w))

        boxes[i].xmax = math.isnan(int((boxes[i].xmax - x\_offset) / x\_scale \* image\_w))

        boxes[i].ymin = math.isnan(int((boxes[i].ymin - y\_offset) / y\_scale \* image\_h))

        boxes[i].ymax = math.isnan(int((boxes[i].ymax - y\_offset) / y\_scale \* image\_h))

def \_interval\_overlap(interval\_a, interval\_b):

    x1, x2 = interval\_a

    x3, x4 = interval\_b

    if x3 < x1:

        if x4 < x1:

            return 0

        else:

            return min(x2,x4) - x1

    else:

        if x2 < x3:

             return 0

        else:

            return min(x2,x4) - x3

def bbox\_iou(box1, box2):

    intersect\_w = \_interval\_overlap([box1.xmin, box1.xmax], [box2.xmin, box2.xmax])

    intersect\_h = \_interval\_overlap([box1.ymin, box1.ymax], [box2.ymin, box2.ymax])

    intersect = intersect\_w \* intersect\_h

    w1, h1 = box1.xmax-box1.xmin, box1.ymax-box1.ymin

    w2, h2 = box2.xmax-box2.xmin, box2.ymax-box2.ymin

    union = w1\*h1 + w2\*h2 - intersect

    if union > 0:

      return float(intersect) / union

    else:

      return float(intersect)

def do\_nms(boxes, nms\_thresh):

    if len(boxes) > 0:

        nb\_class = len(boxes[0].classes)

    else:

        return

    for c in range(nb\_class):

        sorted\_indices = np.argsort([-box.classes[c] for box in boxes])

        print(nb\_class)

        print(c)

        for i in range(len(sorted\_indices)):

            index\_i = sorted\_indices[i]

            if boxes[index\_i].classes[c] == 0: continue

            for j in range(i+1, len(sorted\_indices)):

                index\_j = sorted\_indices[j]

                if bbox\_iou(boxes[index\_i], boxes[index\_j]) >= nms\_thresh:

                    boxes[index\_j].classes[c] = 0

# get all of the results above a threshold

def get\_boxes(boxes, labels, thresh):

    v\_boxes, v\_labels, v\_scores = list(), list(), list()

    # enumerate all boxes

    for box in boxes:

        # enumerate all possible labels

        for i in range(len(labels)):

        # check if the threshold for this label is high enough

            if box.classes[i] > thresh:

                v\_boxes.append(box)

                v\_labels.append(labels[i])

                v\_scores.append(box.classes[i]\*100)

        # don't break, many labels may trigger for one box

    return v\_boxes, v\_labels, v\_scores

# draw all results

def draw\_boxes(filename, v\_boxes, v\_labels, v\_scores):

    # load the image

    data = plt.imread(filename)

    # plot the image

    plt.imshow(data)

    # get the context for drawing boxes

    ax = plt.gca()

    # plot each box

    for i in range(len(v\_boxes)):

        box = v\_boxes[i]

        # get coordinates

        y1, x1, y2, x2 = box.ymin, box.xmin, box.ymax, box.xmax

        # calculate width and height of the box

        width, height = x2 - x1, y2 - y1

        # create the shape

        rect = Rectangle((x1, y1), width, height, fill=False, color='red') # draw the box

        ax.add\_patch(rect)

        # draw text and score in top left corner

        label = "%s (%.3f)" % (v\_labels[i], v\_scores[i])

        plt.text(x1, y1, label, color='red') # show the plot

    plt.show()

anchors = [[116,90, 156,198, 373,326], [30,61, 62,45, 59,119], [10,13, 16,30, 33,23]]

# define the probability threshold for detected objects

class\_threshold = 0.6

boxes = list()

for i in range(len(y\_pred)):

  # decode the output of the network

  boxes += decode\_netout(y\_pred[i][0], anchors[i], class\_threshold, input\_h, input\_w)

len(y\_pred)

len(boxes)

correct\_yolo\_boxes(boxes, image\_h, image\_w, input\_h, input\_w)

#for i  in range(6):

  #boxes = boxes[i]

# suppress non-maximal boxes

do\_nms(boxes, 0.5)

labels = ["person", "bicycle", "car", "motorbike", "aeroplane", "bus", "train", "truck", \

              "boat", "traffic light", "fire hydrant", "stop sign", "parking meter", "bench", \

              "bird", "cat", "dog", "horse", "sheep", "cow", "elephant", "bear", "zebra", "giraffe", \

              "backpack", "umbrella", "handbag", "tie", "suitcase", "frisbee", "skis", "snowboard", \

              "sports ball", "kite", "baseball bat", "baseball glove", "skateboard", "surfboard", \

              "tennis racket", "bottle", "wine glass", "cup", "fork", "knife", "spoon", "bowl", "banana", \

              "apple", "sandwich", "orange", "broccoli", "carrot", "hot dog", "pizza", "donut", "cake", \

              "chair", "sofa", "pottedplant", "bed", "diningtable", "toilet", "tvmonitor", "laptop", "mouse", \

              "remote", "keyboard", "cell phone", "microwave", "oven", "toaster", "sink", "refrigerator", \

              "book", "clock", "vase", "scissors", "teddy bear", "hair drier", "toothbrush"]

# get the details of the detected objects

v\_boxes, v\_labels, v\_scores = get\_boxes(boxes, labels, class\_threshold)

# summarize what we found

for i in range(len(v\_boxes)):

    print(v\_labels[i], v\_scores[i])

draw\_boxes(image\_path, v\_boxes, v\_labels, v\_scores)

imgplot = plt.imshow(boximg)

plt.show()

