MTCNN算法及代码笔记

2017年12月26日 21:48:33 AI之路

论文：Joint Face Detection and Alignment using Multi-task Cascaded Convolutional Networks
论文链接：https://arxiv.org/abs/1604.02878
官方代码链接：https://github.com/kpzhang93/MTCNN_face_detection_alignment
其他代码实现（MXNet）：https://github.com/pangyupo/mxnet_mtcnn_face_detection

这篇博客先介绍MTCNN算法，再介绍代码，结合起来看对算法的理解会更加深入。

算法部分

MTCNN（Multi-task Cascaded Convolutional Networks）算法是用来同时实现face detection和alignment，也就是人脸检测和对齐。文章一方面引入了cascaded structure，另一方面提出一种新的 online hard sample mining。文章的核心思想是原文的这一句话：**our framework adopts a cascaded structure with three stages of carefully designed deep convolutional networks that predict face and landmark location in a coarse-to-fine manner.** 因此该算法的cascaded structure 主要包含三个子网络：**Proposal Network(P-Net)、Refine Network(R-Net)、Output Network(O-Net)**，如Fig1所示，这3个stage对人脸的处理是按照一种由粗到细的方式，也就是原文中说的 a coarse-to-fine manner，在代码中体现得比较明显。另外要注意的是在Figure1中一开始对图像做了multi scale的resize，构成了图像金字塔，然后这些不同scale的图像作为3个stage的输入进行训练，目的是为了可以检测不同scale的人脸。

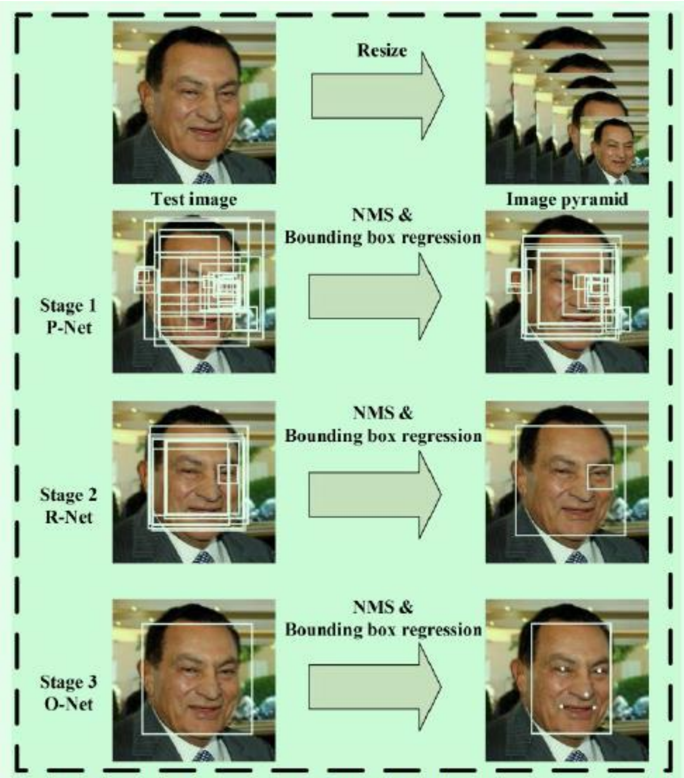

Fig. 1. Pipeline of our cascaded framework that includes three-stage multi-task deep convolutional networks. Firstly, candidate windows are produced through a fast Proposal Network (P-Net). After that, we refine these candidates in the next stage through a Refinement Network (R-Net). In the third stage, The Output Network (O-Net) produces final bounding box and facial landmarks position.

在Figure2中详细介绍了三个子网络的结构。
P-Net主要用来生成一些候选框（bounding box）。在训练的时候该网络的顶部有3条支路用来分别做人脸分类、人脸框的回归和人脸关键点定位；在测试的时候这一步的输出只有N个bounding box的4个坐标信息和score，当然这4个坐标信息已经用回归支路的输出进行修正了，score可以看做是分类的输出（是人脸的概率），具体可以看代码。
R-Net主要用来去除大量的非人脸框。这一步的输入是前面P-Net生成的bounding box，每个bounding box的大小都是24*24，可以通过resize操作得到。同样在测试的时候这一步的输出只有M个bounding box的4个坐标信息和score，4个坐标信息也用回归支路的输出进行修正了

O-Net和R-Net有点像，只不过这一步还增加了landmark位置的回归。输入大小调整为48*48，输出包含P个bounding box的4个坐标信息、score和关键点信息。

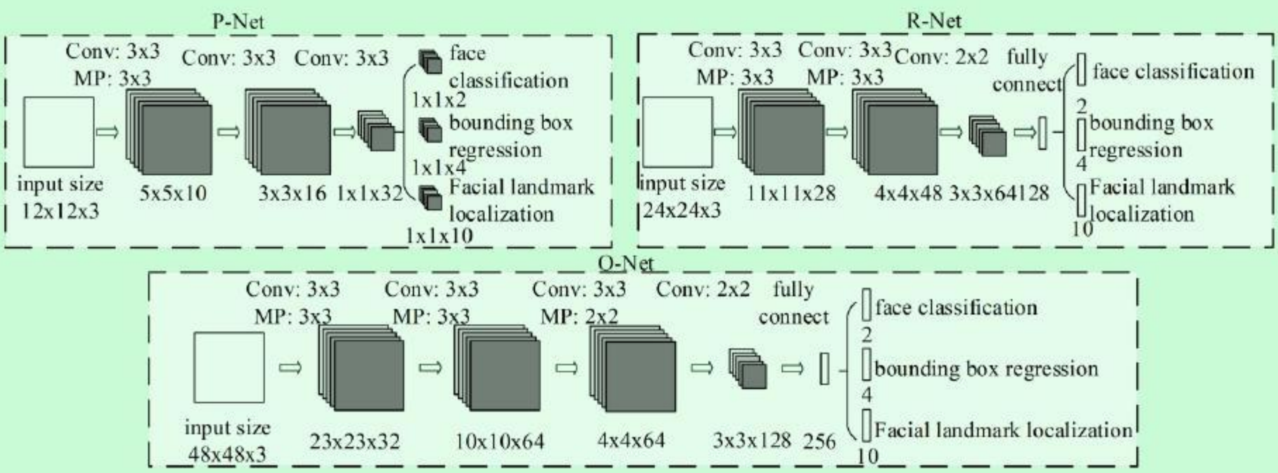

Fig. 2. The architectures of P-Net, R-Net, and O-Net, where "MP" means max pooling and "Conv" means convolution. The step size in convolution and pooling is 1 and 2, respectively.

在训练时候，每个stage的顶部都包含3条支路，接下来简单过一下这3条支路的损失函数。
face classification支路的损失函数如公式1所示，采用的是交叉熵，这是分类算法常用的损失函数，在这里是二分类。

$$L_i^{det} = -(y_i^{det}\log(p_i) + (1-y_i^{det})(1-\log(p_i))) \quad (1)$$

where p_i is the probability produced by the network that indicates a sample being a face. The notation $y_i^{det} \in \{0,1\}$ denotes the ground-truth label.

bounding box regression支路的损失函数如公式2所示，采用的是欧氏距离损失（L2 loss），这也是回归问题常用的损失函数。

$$L_i^{box} = \left\|\hat{y}_i^{box} - y_i^{box}\right\|_2^2 \quad (2)$$

where $\hat{y}_i^{box}$ regression target obtained from the network and y_i^{box} is the ground-truth coordinate. There are four coordinates, including left top, height and width, and thus $y_i^{box} \in \mathbb{R}^4$.

facial landmark localization支路的损失函数如公式3所示，同样采用的是欧氏距离损失（L2 loss）。

$$L_i^{landmark} = \left\|\hat{y}_i^{landmark} - y_i^{landmark}\right\|_2^2 \quad (3)$$

where $\hat{y}_i^{landmark}$ is the facial landmark's coordinate obtained from the network and $y_i^{landmark}$ is the ground-truth coordinate. There are five facial landmarks, including left eye, right eye, nose, left mouth corner, and right mouth corner, and thus $y_i^{landmark} \in \mathbb{R}^{10}$.

因为在训练的时候并不是对每个输入都计算上述的3个损失函数，因此定义了公式4用来控制对不同的输入计算不同的损失。可以在出，在P-Net和R-Net中，关键点的损失权重（α）要小于O-Net部分，这是因为前面2个stage重点在于过滤掉非人脸的bbox。β存在的意义是比如非人

脸输入，就只需要计算分类损失，而不需要计算回归和关键点的损失。

Then the overall learning target can be formulated as:

$$\min \sum_{i=1}^{N} \sum_{j\in\{det,box,landmark\}} \alpha_j \beta_i^j L_i^j \qquad (4)$$

where N is the number of training samples. α_j denotes on the task importance. We use $(\alpha_{det} = 1, \alpha_{box} = 0.5, \alpha_{landmark} = 0.5)$ in P-Net and R-Net, while $(\alpha_{det} = 1, \alpha_{box} = 0.5, \alpha_{landmark} = 1)$ in O-Net for more accurate facial landmarks localization. $\beta_i^j \in \{0,1\}$ is the sample type indicator. In

online hard sample mining 具体而言是这样做的：In particular, in each mini-batch, we sort the loss computed in the forward propagation phase from all samples and select the top 70% of them as hard samples. Then we only compute the gradient from the hard samples in the backward propagation phase.

实验结果：

训练过程中的4中不同的标注数据：here we use four different kinds of data annotation in our training process: **(i) Negatives**: Regions that the Intersec-tion-over-Union (IoU) ratio less than 0.3 to any ground-truth faces; **(ii) Positives**: IoU above 0.65 to a ground truth face; **(iii) Part faces**: IoU between 0.4 and 0.65 to a ground truth face; and **(iv) Landmark faces**: faces labeled 5 landmarks' positions.

Figure4是本文算法在FDDB、WIDER FACE、AFLW数据集上和其他算法的对比结果。

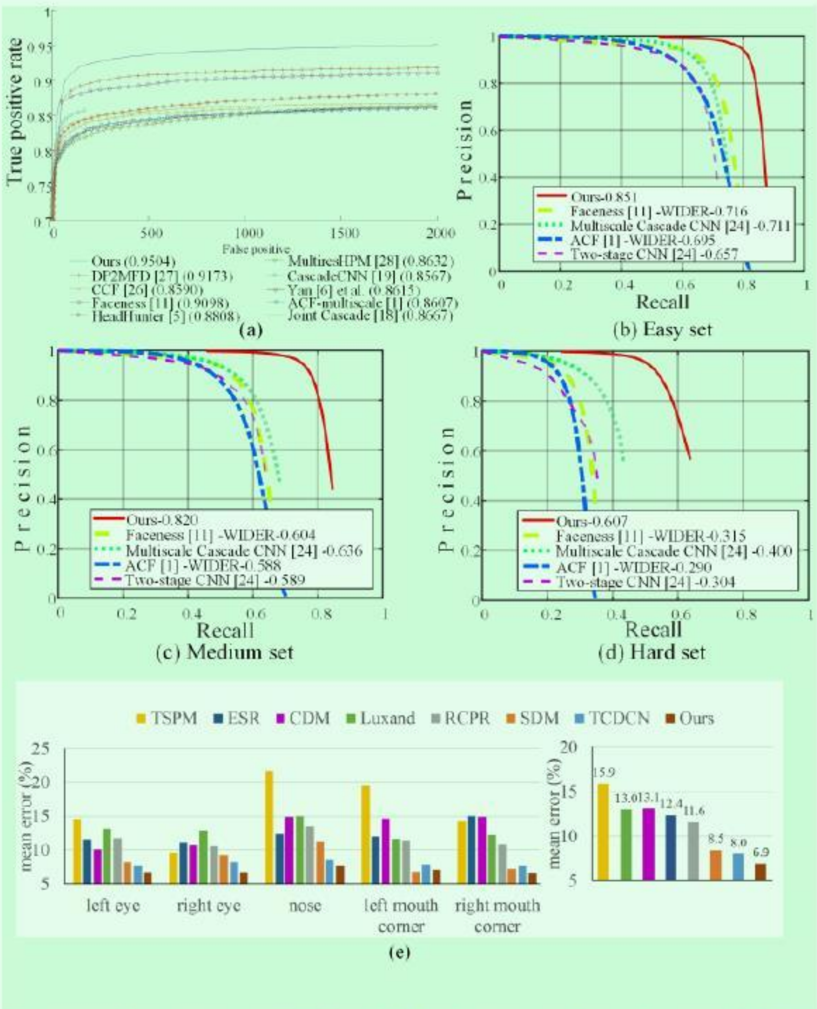

Fig. 4. (a) Evaluation on FDDB. (b-d) Evaluation on three subsets of WIDER FACE. The number following the method indicates the average accuracy. (e) Evaluation on AFLW for face alignment

代码部分

这里采用第三方的MXNet实现版本：https://github.com/pangyupo/mxnet_mtcnn_face_detection。如果感兴趣可以看原论文的代码：https://github.com/kpzhang93/MTCNN_face_detection_alignment。

该项目可以用来测试，主要包含三个脚本：main.py、mtcnn_detector.py、helper.py。main.py是代码的入口，接下来按跑代码时候的调用顺序来记录。

在main.py中，先import相关模块，其中最重要的是MtcnnDetector这个类，该类在mtcnn_detector.py脚本中定义。通过调用MtcnnDetector类的`__init__`函数可以初始化得到一个detector。

```
import mxnet as mx
from mtcnn_detector import MtcnnDetector
import cv2
import os
import time


detector = MtcnnDetector(model_folder='model', ctx=mx.cpu(0), num_worker = 4 , accurate_landmark = False)
```

MtcnnDetector类的`__init__`函数做了哪些工作呢？如下是mtcnn_detector.py脚本下的MtcnnDetector类的`__init__`函数。可以看出都是一些常规的初始化操作。比如models里面放的是训练好的模型路径，self.PNets、self.RNets、self.ONets分别表示论文中算法3个阶段的模型的初始化结果，self.LNet 是用来对关键点坐标进行进一步修正的网络，论文中未提及。这里要注意`__init__`函数的输入中factor表示和图像金字塔相关的一个参数，表示图像金字塔的每相邻两层之间的倍数关系是factor。threshold参数是一个包含3个值的列表，这3个值在算法的3个stage中将分别用到，可以看到这3个threshold值是递增的，是因为在3个stage中对一个bbox是否是人脸的置信度要求越来越高。

```
def __init__(self,
             model_folder='.',
             minsize = 20,
             threshold = [0.6, 0.7, 0.8],
             factor = 0.709,
             num_worker = 1,
             accurate_landmark = False,
             ctx=mx.cpu()):
            """
        Initialize the detector

        Parameters:
        ----------
            model_folder : string
                path for the models
            minsize : float number
                minimal face to detect
            threshold : float number
                detect threshold for 3 stages
            factor: float number
                scale factor for image pyramid
            num_worker: int number
                number of processes we use for first stage
            accurate_landmark: bool
                use accurate landmark localization or not

        """
        self.num_worker = num_worker
        self.accurate_landmark = accurate_landmark

        # load 4 models from folder
        models = ['det1', 'det2', 'det3','det4']
        models = [ os.path.join(model_folder, f) for f in models]

        self.PNets = []
        for i in range(num_worker):
            workner_net = mx.model.FeedForward.load(models[0], 1, ctx=ctx)
```

```
            self.PNets.append(workner_net)

        self.Pool = Pool(num_worker)

        self.RNet = mx.model.FeedForward.load(models[1], 1, ctx=ctx)
        self.ONet = mx.model.FeedForward.load(models[2], 1, ctx=ctx)
        self.LNet = mx.model.FeedForward.load(models[3], 1, ctx=ctx)

        self.minsize   = float(minsize)
        self.factor    = float(factor)
        self.threshold = threshold
```

于是，继续main.py中的思路，在初始化得到detector后，就可以读入图像，这里采用cv2模块读取图像数据，img是numpy array类型的数据。读完图像数据后，就将图像作为detector.detect_face函数的输入，在这个函数里面将执行所有3个stage算法的操作。得到的results是一个长度为2的tuple类型数据，其中results[0]是N*5的numpy array，表示人脸的bbox信息，其中N表示检测到的人脸数量，5表示每张人脸有4个坐标点（左上角的x，y和右下角的x，y）和1个置信度score。results[1]是N*10的numpy array，表示人脸关键点信息，其中N表示检测到的人脸数量，10表示5个关键点的x、y坐标信息。

```
img = cv2.imread('test2.jpg')

# run detector
results = detector.detect_face(img)
```

来详细看看detect_face这个函数的具体内容。整体上这个函数可以分成：初始部分、first stage、second stage、third stage、extend stage这5个部分。初始部分除了一些判断语句外，最重要的是生成一个scales列表，这个列表中放的就是一系列的scale值，表示依次将原图缩小成原图的scale倍，从而组成图像金字塔。首先minl是输入图像的长或宽的最小值，MIN_DET_SIZE表示最小的检测尺寸；self.minisize表示最小的人脸尺寸，是在`__init__`中初始化得到的。

```
def detect_face(self, img):
        """
            detect face over img
        Parameters:
        ----------
            img: numpy array, bgr order of shape (1, 3, n, m)
                input image
        Retures:
        -------
            bboxes: numpy array, n x 5 (x1,y2,x2,y2,score)
                bboxes
            points: numpy array, n x 10 (x1, x2 ... x5, y1, y2 ..y5)
                landmarks
        """

        # check input
        MIN_DET_SIZE = 12

        if img is None:
            return None

        # only works for color image
        if len(img.shape) != 3:
            return None

        # detected boxes
        total_boxes = []

        height, width, _ = img.shape
        minl = min( height, width)

        # get all the valid scales
        scales = []
        m = MIN_DET_SIZE/self.minsize
```

```
        minl *= m
        factor_count = 0
        while minl > MIN_DET_SIZE:
            scales.append(m*self.factor**factor_count)
            minl *= self.factor
            factor_count += 1
```

first stage部分是根据输入的scales列表，生成total_boxes这个二维的numpy ndarray。首先 self.slice_index这个函数将scales列表的index根据num_worker的值进行分组，比如scales列表一共包含9个数，num_worker的值为4，那么sliced_index就是[[0,1,2,3],[4,5,6,7],[8]]。 self.Pool.map()是生成bbox的重要函数，这里的self.Pool是python的multiprocessing库的Pool类，self.Pool.map()有两个输入，第一个输入是一个函数： detect_first_stage_warpper，第二个输入是一个迭代器：izip(repeat(img), self.PNets[:len(batch)], [scales[i] for i in batch], repeat(self.threshold[0]))。这个迭代器有izip函数生成，该函数有4个输入，迭代器的作用就是其生成的每个元素都由这4个输入组成，那么会生成多少个迭代元素呢？答案是len(batch)个，也就是num_worker个。
生成的local_boxes是一个长度为len(batch)的list，list中的每个numpy array表示对应scale的bbox信息，每个numpy array的shape为K*9，K就是bbox的数量，9包含4个坐标点信息，一个置信度score和4个用来调整前面4个坐标点的偏移信息。最后都并到total_boxes列表中，因此该列表一共包含len(scales)个尺度的numpy array，但是由于该列表中某些值是None，所以会有去掉None的操作。total_boxes = np.vstack(total_boxes)是将由numpy array组成的list按照列叠加成一个新的numpy array格式的total_boxes，这个新的total_boxes依然是2维的，每一行代表一个bbox，一共9列。
pick = nms(total_boxes[:, 0:5], 0.7, ‘Union’)操作是为了去掉一些重复框，后面会详细介绍这个nms操作。该函数返回的pick是一个list，list中的值是index，这些index是非重复的index；而这句话：total_boxes = total_boxes[pick] 则是将total_boxes中的这些非重复的框挑选出来。bbw和bbh分别是求bbox的宽和高。然后是refine the bbox这一步，就是调整total_boxes的坐标值。然后将total_boxes的尺寸调整为正方形：total_boxes = self.convert_to_square(total_boxes)，主要是基于人脸一般都是正方形的，最终得到的正方形的中心店还是原来矩形的中心点，边长是矩阵宽高的最大值。最后就是一个对四个坐标点的取整操作。也就是说total_boxes是N*5的numpy array，N表示bbox的数量。

```
        #############################################
    # first stage
    #############################################
        #for scale in scales:
        #    return_boxes = self.detect_first_stage(img, scale, 0)
        #    if return_boxes is not None:
        #        total_boxes.append(return_boxes)

        sliced_index = self.slice_index(len(scales))
        total_boxes = []
        for batch in sliced_index:
            local_boxes = self.Pool.map( detect_first_stage_warpper, \
                    izip(repeat(img), self.PNets[:len(batch)], [scales[i] for i in batch], repeat(self.threshold[0])) )
            total_boxes.extend(local_boxes)

        # remove the Nones
        total_boxes = [ i for i in total_boxes if i is not None]

        if len(total_boxes) == 0:
            return None

        total_boxes = np.vstack(total_boxes)

        if total_boxes.size == 0:
            return None

        # merge the detection from first stage
        pick = nms(total_boxes[:, 0:5], 0.7, 'Union')
        total_boxes = total_boxes[pick]

        bbw = total_boxes[:, 2] - total_boxes[:, 0] + 1
        bbh = total_boxes[:, 3] - total_boxes[:, 1] + 1

        # refine the bboxes
        total_boxes = np.vstack([total_boxes[:, 0]+total_boxes[:, 5] * bbw,
                                 total_boxes[:, 1]+total_boxes[:, 6] * bbh,
```

```
                                 total_boxes[:, 2]+total_boxes[:, 7] * bbw,
                                 total_boxes[:, 3]+total_boxes[:, 8] * bbh,
                                 total_boxes[:, 4]
                                 ])

        total_boxes = total_boxes.T
        total_boxes = self.convert_to_square(total_boxes)
        total_boxes[:, 0:4] = np.round(total_boxes[:, 0:4])
```

nms算法在object detection算法中还是比较常见的，讲解一下这里的实现。nms函数在helper.py脚本中，主要作用是去除重复框。首先输入boxes是一个N*5的numpy array，N表示bbox的数量，overlap_threshold是阈值。area = (x2 - x1 + 1) * (y2 - y1 + 1) 是求取每个bbox的面积，idxs = np.argsort(score)是对bbox的score按从小到大的顺序排序得到idxs。然后在while循环中，每次都从idxs的末尾开始取值，并将index放到pick列表中。 xx1 = np.maximum(x1[i], x1[idxs[:last]])、 yy1 = np.maximum(y1[i], y1[idxs[:last]])、xx2 = np.minimum(x2[i], x2[idxs[:last]])、yy2 = np.minimum(y2[i], y2[idxs[:last]]) 这4行是计算两个框的交集的左上角坐标（xx1，yy1）和右下角坐标（xx2，yy2），不管有无交集，都可以得到这4个值。w = np.maximum(0, xx2 - xx1 + 1)和 h = np.maximum(0, yy2 - yy1 + 1) 这两行将计算bbox的宽度和高度，如果宽度或高度是负值（也就是说不存在这样的bbox，再往前追溯，就是两个框没有交集，因此生成的左上角坐标（xx1，yy1）和右下角（xx2，yy2）构成不了一个框），这样的话就用0值代替。因为mode默认采用’Union’，所以overlap的计算采用overlap = inter / (area[i] + area[idxs[:last]] - inter)，这个公式表达的意思就是两个框的交集面积除以并集的面积。这里捎带解释下当mode采用’Min’时候的情况（后面stage3的时候用到），这个时候overlap的计算公式如下：overlap = inter / np.minimum(area[i], area[idxs[:last]])，这个时候分母变成了两个框中面积最小的那个框的面积，显然这样得到的overlap值比前面mode=’Union’时候的要小。idxs = np.delete(idxs, np.concatenate(([last], np.where(overlap > overlap_threshold)[0]))) 这一行就是将idxs中overlap满足阈值的bbox的index删除。首先是np.where(overlap > overlap_threshold)[0]，得到的是长度为1的tuple的第一个值，是一个numpy array，里面包含的是满足这个条件表达式的bbox的index。然后np.concatenate(([last], np.where(overlap > overlap_threshold)[0]))这个concatenate操作就是将原来socre最大的那个bbox的index和现在满足条件的bbox的index合并成一个numpy array。idxs = np.delete(idxs, np.concatenate(([last], np.where(overlap > overlap_threshold)[0]))). 则是通过调用np.delete函数从idxs中删掉指定bbox的index，然后继续赋值给idxs，这样idxs中就只剩下那些和之前最大score的bbox的overlap比较小的bbox的index，从而构成循环。最后返回的是pick这个列表，因此就达到了nms算法的目的。

```
def nms(boxes, overlap_threshold, mode='Union'):
    """
    non max suppression

  Parameters:
  ----------
    box: numpy array n x 5
      input bbox array
    overlap_threshold: float number
      threshold of overlap
    mode: float number
      how to compute overlap ratio, 'Union' or 'Min'
  Returns:
  -------
    index array of the selected bbox
  """
    # if there are no boxes, return an empty list
    if len(boxes) == 0:
        return []

    # if the bounding boxes integers, convert them to floats
    if boxes.dtype.kind == "i":
        boxes = boxes.astype("float")

    # initialize the list of picked indexes
    pick = []

    # grab the coordinates of the bounding boxes
    x1, y1, x2, y2, score = [boxes[:, i] for i in range(5)]

    area = (x2 - x1 + 1) * (y2 - y1 + 1)
    idxs = np.argsort(score)

    # keep looping while some indexes still remain in the indexes list
```

```
    while len(idxs) > 0:
        # grab the last index in the indexes list and add the index value to the list of picked indexes
        last = len(idxs) - 1
        i = idxs[last]
        pick.append(i)

        xx1 = np.maximum(x1[i], x1[idxs[:last]])
        yy1 = np.maximum(y1[i], y1[idxs[:last]])
        xx2 = np.minimum(x2[i], x2[idxs[:last]])
        yy2 = np.minimum(y2[i], y2[idxs[:last]])

        # compute the width and height of the bounding box
        w = np.maximum(0, xx2 - xx1 + 1)
        h = np.maximum(0, yy2 - yy1 + 1)

        inter = w * h
        if mode == 'Min':
            overlap = inter / np.minimum(area[i], area[idxs[:last]])
        else:
            overlap = inter / (area[i] + area[idxs[:last]] - inter)

        # delete all indexes from the index list that have
        idxs = np.delete(idxs, np.concatenate(([last], np.where(overlap > overlap_threshold)[0])))

    return pick
```

second stage，先调用self.pad函数生成 dy, edy, dx, edx, y, ey, x, ex, tmpw, tmph。这几个变量的含义如下：dy, dx : numpy array, n x 1, start point of the bbox in target image. edy, edx : numpy array, n x 1, end point of the bbox in target image. y, x : numpy array, n x 1, start point of the bbox in original image. ex, ex : numpy array, n x 1, end point of the bbox in original image. tmph, tmpw: numpy array, n x 1, height and width of the bbox. 这个函数在生成这些变量的时候还会做一些检查操作，避免尺寸超过图像大小或者尺寸是负值等。
然后是一个循环，遍历所有的bbox，并根据每个bbox的尺寸和坐标信息从图像中扣出相应的bbox，保存在一个临时变量tmp中，最后调用adjust_input函数将输入resize到24*24，并从（h,w,c）转换成（1,c,h,w），也就是不仅交换了通道，还新增加了一维变成了4维，另外还做了归一化操作。最后得到的input_buf的维度是（N,3,24,24），N表示bbox的数量。准备好输入数据之后，就调用output = self.RNet.predict(input_buf)进行预测，输出output是一个长度为2的list，其中output[0]是大小为N*4的numpy array，表示N个bbox的回归信息；output[1]是大小为N*2的numpy array，表示N个bbox的类别信息。passed = np.where(output[1][:, 1] > self.threshold[1])这一行是通过比较某个bbox属于人脸的概率和阈值来判断该bbox是否是人脸。通过这一步就可以过滤掉大部分的非人脸bbox。然后通过passed = np.where(output[1][:, 1] > self.threshold[1])将人脸概率信息也添加到total_boxes中，相当于score。reg = output[0][passed]则是将那些概率符合预期的bbox的回归信息赋值给reg。接下来的nms操作前面已经介绍过了，主要是用来去除重复框的。total_boxes = self.calibrate_box(total_boxes, reg[pick]) 这一行就是根据回归信息reg来调整total_boxes中bbox的坐标信息，大致的计算是total_boxes中的bbox的4个坐标值分别加上bbox的宽或高和reg的乘积（宽和x相乘，高和y相乘）。因为调整后的bbox的尺寸可能不是正方形，因此再次调用total_boxes = self.convert_to_square(total_boxes)将输入bbox的尺寸调整为正方形。最后的total_boxes[:, 0:4] = np.round(total_boxes[:, 0:4]) 就是将4个坐标值从float64转成整数。

```
        #############################################
    # second stage
    #############################################
        num_box = total_boxes.shape[0]

        # pad the bbox
        [dy, edy, dx, edx, y, ey, x, ex, tmpw, tmph] = self.pad(total_boxes, width, height)
        # (3, 24, 24) is the input shape for RNet
        input_buf = np.zeros((num_box, 3, 24, 24), dtype=np.float32)

        for i in range(num_box):
            tmp = np.zeros((tmph[i], tmpw[i], 3), dtype=np.uint8)
            tmp[dy[i]:edy[i]+1, dx[i]:edx[i]+1, :] = img[y[i]:ey[i]+1, x[i]:ex[i]+1, :]
            input_buf[i, :, :, :] = adjust_input(cv2.resize(tmp, (24, 24)))

        output = self.RNet.predict(input_buf)

        # filter the total_boxes with threshold
```

```
        passed = np.where(output[1][:, 1] > self.threshold[1])
        total_boxes = total_boxes[passed]

        if total_boxes.size == 0:
            return None

        total_boxes[:, 4] = output[1][passed, 1].reshape((-1,))
        reg = output[0][passed]

        # nms
        pick = nms(total_boxes, 0.7, 'Union')
        total_boxes = total_boxes[pick]
        total_boxes = self.calibrate_box(total_boxes, reg[pick])
        total_boxes = self.convert_to_square(total_boxes)
        total_boxes[:, 0:4] = np.round(total_boxes[:, 0:4])
```

third stage 和 second stage部分很像，需要注意的首先是third stage的输入图像大小是3*48*48，其次，output = self.ONet.predict(input_buf) 生成的output是一个长度为3的list，其中output[0]是N*10的numpy array，表示每个bbox的5个关键点的x、y坐标相关信息，剩下的output[1]和output[2]和second stage类似，分别表示回归信息和分类信息。然后是计算landmark point部分，因为前面得到的关键点的x、y坐标相关信息并不直接是x、y的值，而是一个scale值，最终的关键点的x、y值可以通过这个scale值和bbox的宽高相乘再累加到bbox的坐标得到，具体而言就是下面这两行代码：points[:, 0:5] = np.expand_dims(total_boxes[:, 0], 1) + np.expand_dims(bbw, 1) * points[:, 0:5]、points[:, 5:10] = np.expand_dims(total_boxes[:, 1], 1) + np.expand_dims(bbh, 1) * points[:, 5:10]。然后是nms算法，这里要注意的是调用nms算法时候，mode采用’Min’，可以看前面关于nms部分的介绍，另外在nms算法开始之前先对total_boxes中的4个坐标值利用回归信息（reg）进行修正。因为前面初始化的时候self.accurate_landmark是False，所以直接返回：return total_boxes, points。

```
        #############################################
    # third stage
    #############################################
        num_box = total_boxes.shape[0]

        # pad the bbox
        [dy, edy, dx, edx, y, ey, x, ex, tmpw, tmph] = self.pad(total_boxes, width, height)
        # (3, 48, 48) is the input shape for ONet
        input_buf = np.zeros((num_box, 3, 48, 48), dtype=np.float32)

        for i in range(num_box):
            tmp = np.zeros((tmph[i], tmpw[i], 3), dtype=np.float32)
            tmp[dy[i]:edy[i]+1, dx[i]:edx[i]+1, :] = img[y[i]:ey[i]+1, x[i]:ex[i]+1, :]
            input_buf[i, :, :, :] = adjust_input(cv2.resize(tmp, (48, 48)))

        output = self.ONet.predict(input_buf)

        # filter the total_boxes with threshold
        passed = np.where(output[2][:, 1] > self.threshold[2])
        total_boxes = total_boxes[passed]

        if total_boxes.size == 0:
            return None

        total_boxes[:, 4] = output[2][passed, 1].reshape((-1,))
        reg = output[1][passed]
        points = output[0][passed]

        # compute landmark points
        bbw = total_boxes[:, 2] - total_boxes[:, 0] + 1
        bbh = total_boxes[:, 3] - total_boxes[:, 1] + 1
        points[:, 0:5] = np.expand_dims(total_boxes[:, 0], 1) + np.expand_dims(bbw, 1) * points[:, 0:5]
        points[:, 5:10] = np.expand_dims(total_boxes[:, 1], 1) + np.expand_dims(bbh, 1) * points[:, 5:10]

        # nms
        total_boxes = self.calibrate_box(total_boxes, reg)
        pick = nms(total_boxes, 0.7, 'Min')
        total_boxes = total_boxes[pick]
```

```
        total_boxes = total_boxes[pick]
        points = points[pick]

        if not self.accurate_landmark:
            return total_boxes, points
```

如果在初始化的时候 self.accurate_landmark 值为True，那么将执行extended stage部分，这一部分主要是对关键点（points）做修正。patchw = np.maximum(total_boxes[:, 2]-total_boxes[:, 0]+1, total_boxes[:, 3]-total_boxes[:, 1]+1) 是求出每个bbox的宽高的最大值。patchw[np.where(np.mod(patchw,2) == 1)] += 1 是将patchw中的奇数值加1变成偶数值。for i in range(5): 表示遍历5个关键点，for j in range(num_box): 表示遍历所有的bbox。最后生成的input_buf是一个N*(3*5)*24*24的4维numpy array，然后进行 output = self.LNet.predict(input_buf) 操作得到output是一个长度为5的list，list的每个值是一个尺寸为N*2的numpy array，表示每个bbox的这个关键点的坐标值修正参数。

```
        #################################################
        # extended stage
        #################################################
        num_box = total_boxes.shape[0]
        patchw = np.maximum(total_boxes[:, 2]-total_boxes[:, 0]+1, total_boxes[:, 3]-total_boxes[:, 1]+1)
        patchw = np.round(patchw*0.25)

        # make it even
        patchw[np.where(np.mod(patchw,2) == 1)] += 1

        input_buf = np.zeros((num_box, 15, 24, 24), dtype=np.float32)
        for i in range(5):
            x, y = points[:, i], points[:, i+5]
            x, y = np.round(x-0.5*patchw), np.round(y-0.5*patchw)
            [dy, edy, dx, edx, y, ey, x, ex, tmpw, tmph] = self.pad(np.vstack([x, y, x+patchw-1, y+patchw-1]).T,
                                                                    width,
                                                                    height)
            for j in range(num_box):
                tmpim = np.zeros((tmpw[j], tmpw[j], 3), dtype=np.float32)
                tmpim[dy[j]:edy[j]+1, dx[j]:edx[j]+1, :] = img[y[j]:ey[j]+1, x[j]:ex[j]+1, :]
                input_buf[j, i*3:i*3+3, :, :] = adjust_input(cv2.resize(tmpim, (24, 24)))

        output = self.LNet.predict(input_buf)

        pointx = np.zeros((num_box, 5))
        pointy = np.zeros((num_box, 5))

        for k in range(5):
            # do not make a large movement
            tmp_index = np.where(np.abs(output[k]-0.5) > 0.35)
            output[k][tmp_index[0]] = 0.5

            pointx[:, k] = np.round(points[:, k] - 0.5*patchw) + output[k][:, 0]*patchw
            pointy[:, k] = np.round(points[:, k+5] - 0.5*patchw) + output[k][:, 1]*patchw

        points = np.hstack([pointx, pointy])
        points = points.astype(np.int32)

        return total_boxes, points
```

到此为止，也就是执行完 results = detector.detect_face(img) 这一行后，该算法的主要内容就讲完了，得到的results是一个长度为2的tuple，其中result[0]是人脸框的坐标和置信度信息，是一个N*5的numpy array；result[1]是人脸关键点信息，是一个N*10的numpy array。剩下的main.py的内容就是和展示结果相关的代码，比如 cv2.rectangle(draw, (int(b[0]), int(b[1])), (int(b[2]), int(b[3])), (255, 255, 255)) 是将人脸框画在原图上，cv2.circle(draw, (p[i], p[i + 5]), 1, (0, 0, 255), 2) 是将关键点信息画在原图上，这里不再赘述了。需要注意的是生成chips部分的内容不是必须的。综上，mtcnn的测试代码就介绍完了。

```
    if results is not None:
```

```
    total_boxes = results[0]
    points = results[1]

    # extract aligned face chips
    chips = detector.extract_image_chips(img, points, 144, 0.37)
    for i, chip in enumerate(chips):
        cv2.imshow('chip_'+str(i), chip)
        cv2.imwrite('chip_'+str(i)+'.png', chip)

    draw = img.copy()
    for b in total_boxes:
        cv2.rectangle(draw, (int(b[0]), int(b[1])), (int(b[2]), int(b[3])), (255, 255, 255))

    for p in points:
        for i in range(5):
            cv2.circle(draw, (p[i], p[i + 5]), 1, (0, 0, 255), 2)

    cv2.imshow("detection result", draw)
    cv2.waitKey(0)
```