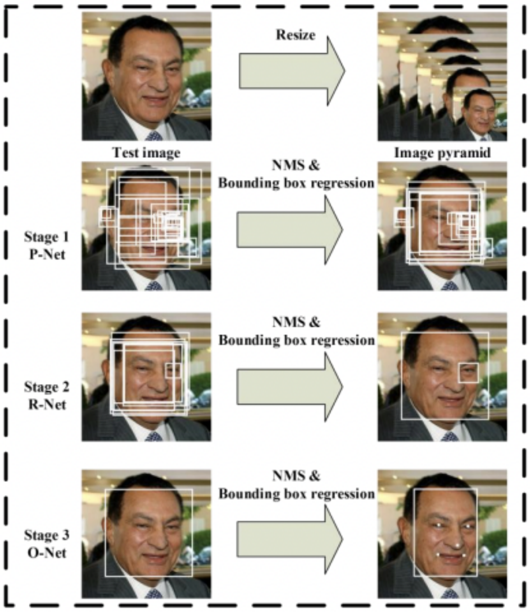
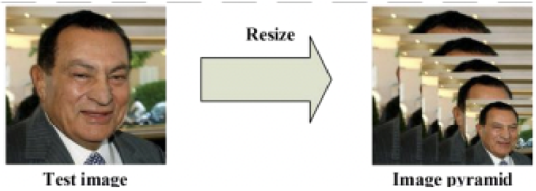
MTCNN



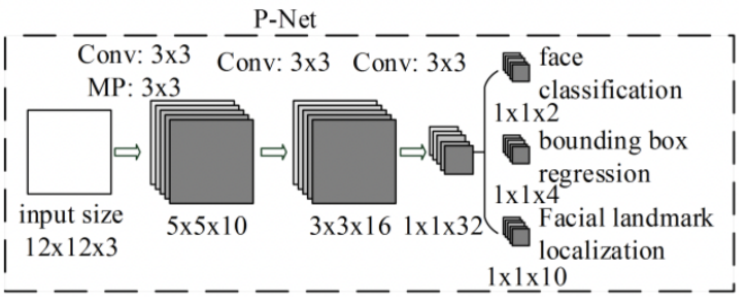
MTCNN은 총 3개의 스테이지로 구성되어있으며, Resize, P-Net, R-Net, O-Net을 의미함. MT는 multi-task를 의미하며 face classification, face landmark localization, bounding box regression 세 개의 태스크를 함께 학습하는 joint learning 방식을 사용하고 있음. joint learning을 통해 기존의 다른 방법에 비해 정확도가 높고 속도가 빠르다고 함.

Step0. Resize



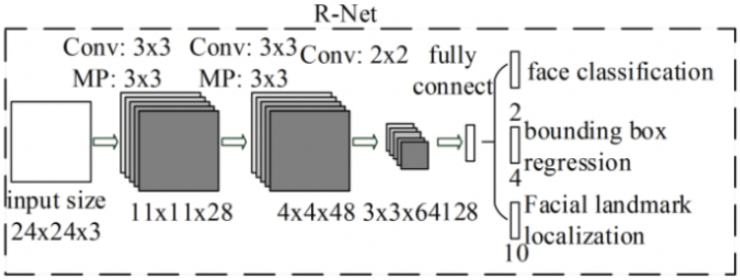
P-Net을 적용하기 전, input 이미지를 각기 다른 scale로 resize하여 image pyramid를 만든다.
→ 다양한 사이즈의 얼굴을 더 잘 detection하기 위함.

Step1. P-Net (Proposal Network)



P-Net은 이미지에서 얼굴을 찾아내는 데 중점을 둔 Network임.
fully connected layer가 없고, conv layer로 이루어진 것이 특징.
P-Net을 통해 수 많은 bbox regression vector와 후보 영역을 얻게되는데 이들을 NMS(non-maximum suppression) 알고리즘으로 높은 정확도의 후보 영역만 남도록 추려낸다.
face classification, bbox regression, face landmark localization 값을 다음 단계로 보낸다.

Step2. R-Net(Refine Network)



P-Net에서 찾아낸 후보 영역을 추려내는 데 중점을 둔 network.
P-Netdml 구조와 유사하지만 끝에 fully connected layer가 추가되어있음.

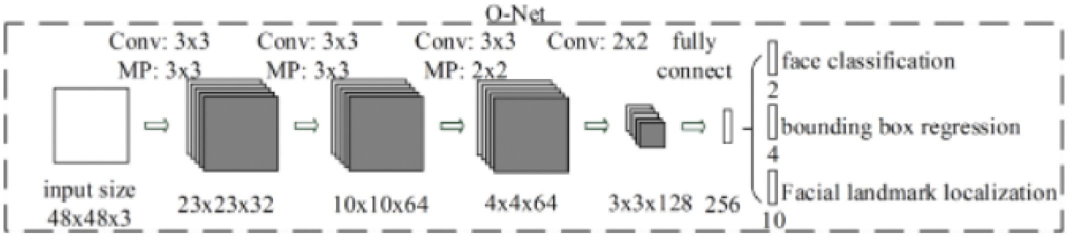
R-Net에서도 bbox regression vector와 후보 영역을 얻어내고, 이들을 NMS알고리즘으로 높은 정확도의 후보 영역만 남도록 추려냄. fcl가 있기에 조금 더 정확한 값을 추려낼 수 있다고 추측할 수 있음.
P-Net과 마찬가지로 남아있는 후보영역의 값을 다음 단계로 보낸다.

각 단계에서 얻어내는 값

- face classification (2개)
    - Ydet = GT에서 얼굴이 있는지 여부 (있으면 1, 없으면 0)
    - p = 얼굴이 있을 확률
- bbox regression (4개)
    - 예측한 bbox 왼쪽 상단 x,y 좌표
    - 예측한 bbox 너비와 높이
- face landmark localization (10개)

- 왼쪽 눈 x,y 좌표
- 오른쪽 눈 x,y 좌표
- 코 x,y 좌표
- 입 왼쪽 끝 x,y
- 입 오른쪽 끝 x,y

Step3. O-Net(Output Network)



O-Net은 이전 단계에서 찾아낸 후보 영역에서 face landmark를 찾아내는 데 중점을 둔 network.
→ 최종적인 face calssification, bbox regression, face landmark localization 값을 얻게 된다.

학습 데이터

학습에는 4가지 종류의 데이터가 필요하다.

1. Negatives: IoU 비율이 0.3 미만인 영역
2. Positives: GT에 대해 0.65 이상의 IoU
3. Part faces: GT에 대해 0.4~0.65 사이의 IoU
4. Landmark faces: 5가지 랜드마크 포인트로 라벨된 얼굴

Loss Function 과 Joint Train의 효과

- Face Classification loss

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

ydet(GT에서 얼굴이 있는지 여부(있을때 1, 없을때 0)) 와 p(얼굴이 있을 확률)의 cross entropy 값을 통해 Face Classification loss를 구함.

- Bounding box Regression loss

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

bbox의 왼쪽상단 x,y좌표, 높이, 너비로 이루어진 4개의 값과 GT의 값의 차이를 Euclidean norm 으로 구한다.

- Facial landmark Localization loss

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

얼굴의 다섯개의 포인트(두 눈, 코, 입의 양쪽 끝 부분)의 x,y 좌표값과 GT의 값의 차이를 Euclidean norm 으로 구한다.

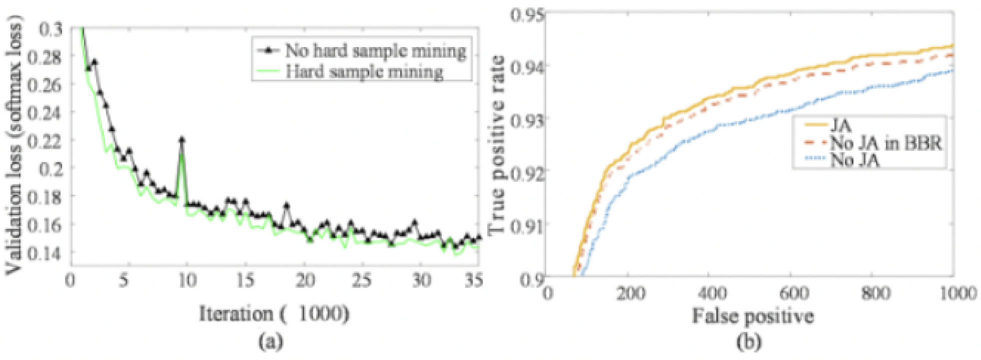
- Multi-source training loss

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

  - **αj**: 각 단계마다 각 loss의 가중치를 다르게 주기위해 사용.
    해당 논문에서는 P-Net과 R-Net에 (αdet = 1, αbox = 0.5, αlandmark = 0.5)를 주어 detection에 중점을 두었다.

    O-Net에서는 (αdet = 1, αbox = 0.5, αlandmark = 1)를 주어 facial landmark localization에 중점을 두었다.
  - **βj**: 특정 샘플이 선택되었는지 여부(0,1)
  - **Lj:**은 위에서 구한 각 목적별 loss.

- joint train



joint training을 통해 큰 폭으로 성능이 높아짐.

파라미터

- weights_file: 가중치를 불러오는 파일. MTCNN을 구성하는 P,R,O network의 가중치를 의미.
- min_face_size: 얼굴로 추정할 최소 크기 default = 20

- steps_threshold: P,R,O network에 사용하는 threshold를 의미. 각 모델에서 나온 결과의 신뢰도를 봐서 threshold % 이상이면 결과값을 사용함. default = [0.6, 0.7, 0.6]
- scale_factor: 스케일링에 사용할 값. default = 0.709

mtcnn code

```
import cv2
import numpy as np
import pkg_resources

from mtcnn.exceptions import InvalidImage
from mtcnn.network.factory import NetworkFactory

__author__ = "Iván de Paz Centeno"


class StageStatus(object):
    """
    Keeps status between MTCNN stages
    """

    def __init__(self, pad_result: tuple = None, width=0, height=0):
        self.width = width
        self.height = height
        self.dy = self.edy = self.dx = self.edx = self.y = self.ey = self.x = self.ex = self.tmpw = self.tmph = []

        if pad_result is not None:
            self.update(pad_result)

    def update(self, pad_result: tuple):
        s = self
        s.dy, s.edy, s.dx, s.edx, s.y, s.ey, s.x, s.ex, s.tmpw, s.tmph = pad_result


class MTCNN(object):
    """
    Allows to perform MTCNN Detection ->
        a) Detection of faces (with the confidence probability)
        b) Detection of keypoints (left eye, right eye, nose, mouth_left, mouth_right)
    """

    def __init__(self, weights_file: str = None, min_face_size: int = 20, steps_threshold: list = None,
                 scale_factor: float = 0.709):
        """
        Initializes the MTCNN.
        :param weights_file: file uri with the weights of the P, R and O networks from MTCNN. By default it will load
        the ones bundled with the package.
        :param min_face_size: minimum size of the face to detect
        :param steps_threshold: step's thresholds values
        :param scale_factor: scale factor
        """
        if steps_threshold is None:
            steps_threshold = [0.6, 0.7, 0.7]

        if weights_file is None:
            weights_file = pkg_resources.resource_stream('mtcnn', 'data/mtcnn_weights.npy')

        self._min_face_size = min_face_size
        self._steps_threshold = steps_threshold
        self._scale_factor = scale_factor
        self._pnet, self._rnet, self._onet = NetworkFactory().build_P_R_O_nets_from_file(weights_file)

    @property
    def min_face_size(self):
        return self._min_face_size

    @min_face_size.setter
    def min_face_size(self, mfc=20):
        try:
            self._min_face_size = int(mfc)
        except ValueError:
            self._min_face_size = 20

    def __compute_scale_pyramid(self, m, min_layer):
        scales = []
        factor_count = 0

        while min_layer >= 12:
            scales += [m * np.power(self._scale_factor, factor_count)]
            min_layer = min_layer * self._scale_factor
            factor_count += 1

        return scales

    @staticmethod
    def __scale_image(image, scale: float):
        """
        Scales the image to a given scale.
        :param image:
        :param scale:
        :return:
        """
        height, width, _ = image.shape

        width_scaled = int(np.ceil(width * scale))
        height_scaled = int(np.ceil(height * scale))

        im_data = cv2.resize(image, (width_scaled, height_scaled), interpolation=cv2.INTER_AREA)

        # Normalize the image's pixels
        im_data_normalized = (im_data - 127.5) * 0.0078125

        return im_data_normalized

    @staticmethod
    def __generate_bounding_box(imap, reg, scale, t):

        # use heatmap to generate bounding boxes
        stride = 2
        cellsize = 12

        imap = np.transpose(imap)
        dx1 = np.transpose(reg[:, :, 0])
        dy1 = np.transpose(reg[:, :, 1])
        dx2 = np.transpose(reg[:, :, 2])
        dy2 = np.transpose(reg[:, :, 3])

        y, x = np.where(imap >= t)

        if y.shape[0] == 1:
            dx1 = np.flipud(dx1)
            dy1 = np.flipud(dy1)
            dx2 = np.flipud(dx2)
            dy2 = np.flipud(dy2)

        score = imap[(y, x)]
        reg = np.transpose(np.vstack([dx1[(y, x)], dy1[(y, x)], dx2[(y, x)], dy2[(y, x)]]))
        if reg.size == 0:
            reg = np.empty(shape=(0, 3))

        bb = np.transpose(np.vstack([y, x]))

        q1 = np.fix((stride * bb + 1) / scale)
        q2 = np.fix((stride * bb + cellsize) / scale)
        boundingbox = np.hstack([q1, q2, np.expand_dims(score, 1), reg])

        return boundingbox, reg

    @staticmethod
    def __nms(boxes, threshold, method):
        """
        Non Maximum Suppression.
        :param boxes: np array with bounding boxes.
        :param threshold:
        :param method: NMS method to apply. Available values ('Min', 'Union')
        :return:
        """
        if boxes.size == 0:
            return np.empty((0, 3))

        x1 = boxes[:, 0]
        y1 = boxes[:, 1]
        x2 = boxes[:, 2]
        y2 = boxes[:, 3]
        s = boxes[:, 4]

        area = (x2 - x1 + 1) * (y2 - y1 + 1)
        sorted_s = np.argsort(s)

        pick = np.zeros_like(s, dtype=np.int16)
        counter = 0
        while sorted_s.size > 0:
            i = sorted_s[-1]
            pick[counter] = i
            counter += 1
            idx = sorted_s[0:-1]

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

            w = np.maximum(0.0, xx2 - xx1 + 1)
            h = np.maximum(0.0, yy2 - yy1 + 1)

            inter = w * h

            if method is 'Min':
                o = inter / np.minimum(area[i], area[idx])
            else:
                o = inter / (area[i] + area[idx] - inter)

            sorted_s = sorted_s[np.where(o <= threshold)]

        pick = pick[0:counter]

        return pick

    @staticmethod
    def __pad(total_boxes, w, h):
        # compute the padding coordinates (pad the bounding boxes to square)
        tmpw = (total_boxes[:, 2] - total_boxes[:, 0] + 1).astype(np.int32)
        tmph = (total_boxes[:, 3] - total_boxes[:, 1] + 1).astype(np.int32)
        numbox = total_boxes.shape[0]
        dx = np.ones(numbox, dtype=np.int32)
        dy = np.ones(numbox, dtype=np.int32)
        edx = tmpw.copy().astype(np.int32)
        edy = tmph.copy().astype(np.int32)

        x = total_boxes[:, 0].copy().astype(np.int32)
        y = total_boxes[:, 1].copy().astype(np.int32)
        ex = total_boxes[:, 2].copy().astype(np.int32)
        ey = total_boxes[:, 3].copy().astype(np.int32)

        tmp = np.where(ex > w)
        edx.flat[tmp] = np.expand_dims(-ex[tmp] + w + tmpw[tmp], 1)
        ex[tmp] = w

        tmp = np.where(ey > h)
        edy.flat[tmp] = np.expand_dims(-ey[tmp] + h + tmph[tmp], 1)
        ey[tmp] = h

        tmp = np.where(x < 1)
        dx.flat[tmp] = np.expand_dims(2 - x[tmp], 1)
        x[tmp] = 1

        tmp = np.where(y < 1)
        dy.flat[tmp] = np.expand_dims(2 - y[tmp], 1)
        y[tmp] = 1

        return dy, edy, dx, edx, y, ey, x, ex, tmpw, tmph

    @staticmethod
    def __rerec(bbox):
        # convert bbox to square
        height = bbox[:, 3] - bbox[:, 1]
        width = bbox[:, 2] - bbox[:, 0]
        max_side_length = np.maximum(width, height)
        bbox[:, 0] = bbox[:, 0] + width * 0.5 - max_side_length * 0.5
        bbox[:, 1] = bbox[:, 1] + height * 0.5 - max_side_length * 0.5
        bbox[:, 2:4] = bbox[:, 0:2] + np.transpose(np.tile(max_side_length, (2, 1)))
        return bbox

    @staticmethod
    def __bbreg(boundingbox, reg):
        # calibrate bounding boxes
        if reg.shape[1] == 1:
            reg = np.reshape(reg, (reg.shape[2], reg.shape[3]))

        w = boundingbox[:, 2] - boundingbox[:, 0] + 1
        h = boundingbox[:, 3] - boundingbox[:, 1] + 1
        b1 = boundingbox[:, 0] + reg[:, 0] * w
        b2 = boundingbox[:, 1] + reg[:, 1] * h
        b3 = boundingbox[:, 2] + reg[:, 2] * w
        b4 = boundingbox[:, 3] + reg[:, 3] * h
        boundingbox[:, 0:4] = np.transpose(np.vstack([b1, b2, b3, b4]))
        return boundingbox

    def detect_faces(self, img) -> list:
        """
        Detects bounding boxes from the specified image.
        :param img: image to process
        :return: list containing all the bounding boxes detected with their keypoints.
        """
        if img is None or not hasattr(img, "shape"):
            raise InvalidImage("Image not valid.")

        height, width, _ = img.shape
        stage_status = StageStatus(width=width, height=height)

        m = 12 / self._min_face_size
        min_layer = np.amin([height, width]) * m
        scales = self.__compute_scale_pyramid(m, min_layer)

        stages = [self.__stage1, self.__stage2, self.__stage3]
        result = [scales, stage_status]

        # We pipe here each of the stages
        for stage in stages:
            result = stage(img, result[0], result[1])

        [total_boxes, points] = result

        bounding_boxes = []

        for bounding_box, keypoints in zip(total_boxes, points.T):
            x = max(0, int(bounding_box[0]))
            y = max(0, int(bounding_box[1]))
            width = int(bounding_box[2] - x)
            height = int(bounding_box[3] - y)
            bounding_boxes.append({
                'box': [x, y, width, height],
                'confidence': bounding_box[-1],
                'keypoints': {
                    'left_eye': (int(keypoints[0]), int(keypoints[5])),
                    'right_eye': (int(keypoints[1]), int(keypoints[6])),
                    'nose': (int(keypoints[2]), int(keypoints[7])),
                    'mouth_left': (int(keypoints[3]), int(keypoints[8])),
                    'mouth_right': (int(keypoints[4]), int(keypoints[9])),
                }
            })

        return bounding_boxes

    def __stage1(self, image, scales: list, stage_status: StageStatus):
        """
        First stage of the MTCNN.
        :param image:
        :param scales:
        :param stage_status:
        :return:
        """
        total_boxes = np.empty((0, 9))
        status = stage_status

        for scale in scales:
            scaled_image = self.__scale_image(image, scale)

            img_x = np.expand_dims(scaled_image, 0)
            img_y = np.transpose(img_x, (0, 2, 1, 3))

            out = self._pnet.predict(img_y)

            out0 = np.transpose(out[0], (0, 2, 1, 3))
            out1 = np.transpose(out[1], (0, 2, 1, 3))

            boxes, _ = self.__generate_bounding_box(out1[0, :, :, 1].copy(),
                                                    out0[0, :, :, :].copy(), scale, self._steps_threshold[0])

            # inter-scale nms
            pick = self.__nms(boxes.copy(), 0.5, 'Union')
            if boxes.size > 0 and pick.size > 0:
                boxes = boxes[pick, :]
                total_boxes = np.append(total_boxes, boxes, axis=0)

        numboxes = total_boxes.shape[0]

        if numboxes > 0:
            pick = self.__nms(total_boxes.copy(), 0.7, 'Union')
            total_boxes = total_boxes[pick, :]

            regw = total_boxes[:, 2] - total_boxes[:, 0]
            regh = total_boxes[:, 3] - total_boxes[:, 1]

            qq1 = total_boxes[:, 0] + total_boxes[:, 5] * regw
            qq2 = total_boxes[:, 1] + total_boxes[:, 6] * regh
            qq3 = total_boxes[:, 2] + total_boxes[:, 7] * regw
            qq4 = total_boxes[:, 3] + total_boxes[:, 8] * regh

            total_boxes = np.transpose(np.vstack([qq1, qq2, qq3, qq4, total_boxes[:, 4]]))
            total_boxes = self.__rerec(total_boxes.copy())

            total_boxes[:, 0:4] = np.fix(total_boxes[:, 0:4]).astype(np.int32)
            status = StageStatus(self.__pad(total_boxes.copy(), stage_status.width, stage_status.height),
                                 width=stage_status.width, height=stage_status.height)

        return total_boxes, status

    def __stage2(self, img, total_boxes, stage_status: StageStatus):
        """
        Second stage of the MTCNN.
        :param img:
        :param total_boxes:
        :param stage_status:
        :return:
        """

        num_boxes = total_boxes.shape[0]
        if num_boxes == 0:
            return total_boxes, stage_status

        # second stage
        tempimg = np.zeros(shape=(24, 24, 3, num_boxes))

        for k in range(0, num_boxes):
            tmp = np.zeros((int(stage_status.tmph[k]), int(stage_status.tmpw[k]), 3))

            tmp[stage_status.dy[k] - 1:stage_status.edy[k], stage_status.dx[k] - 1:stage_status.edx[k], :] = \
                img[stage_status.y[k] - 1:stage_status.ey[k], stage_status.x[k] - 1:stage_status.ex[k], :]

            if tmp.shape[0] > 0 and tmp.shape[1] > 0 or tmp.shape[0] == 0 and tmp.shape[1] == 0:
                tempimg[:, :, :, k] = cv2.resize(tmp, (24, 24), interpolation=cv2.INTER_AREA)

            else:
                return np.empty(shape=(0,)), stage_status

        tempimg = (tempimg - 127.5) * 0.0078125
        tempimg1 = np.transpose(tempimg, (3, 1, 0, 2))

        out = self._rnet.predict(tempimg1)

        out0 = np.transpose(out[0])
        out1 = np.transpose(out[1])

        score = out1[1, :]

        ipass = np.where(score > self._steps_threshold[1])

        total_boxes = np.hstack([total_boxes[ipass[0], 0:4].copy(), np.expand_dims(score[ipass].copy(), 1)])

        mv = out0[:, ipass[0]]

        if total_boxes.shape[0] > 0:
            pick = self.__nms(total_boxes, 0.7, 'Union')
            total_boxes = total_boxes[pick, :]
            total_boxes = self.__bbreg(total_boxes.copy(), np.transpose(mv[:, pick]))
            total_boxes = self.__rerec(total_boxes.copy())
        return total_boxes, stage_status

    def __stage3(self, img, total_boxes, stage_status: StageStatus):
        """
        Third stage of the MTCNN.
        :param img:
        :param total_boxes:
        :param stage_status:
        :return:
        """
        num_boxes = total_boxes.shape[0]
        if num_boxes == 0:
            return total_boxes, np.empty(shape=(0,))

        total_boxes = np.fix(total_boxes).astype(np.int32)

        status = StageStatus(self.__pad(total_boxes.copy(), stage_status.width, stage_status.height),
                             width=stage_status.width, height=stage_status.height)

        tempimg = np.zeros((48, 48, 3, num_boxes))

        for k in range(0, num_boxes):

            tmp = np.zeros((int(status.tmph[k]), int(status.tmpw[k]), 3))

            tmp[status.dy[k] - 1:status.edy[k], status.dx[k] - 1:status.edx[k], :] = \
                img[status.y[k] - 1:status.ey[k], status.x[k] - 1:status.ex[k], :]

            if tmp.shape[0] > 0 and tmp.shape[1] > 0 or tmp.shape[0] == 0 and tmp.shape[1] == 0:
                tempimg[:, :, :, k] = cv2.resize(tmp, (48, 48), interpolation=cv2.INTER_AREA)
            else:
                return np.empty(shape=(0,)), np.empty(shape=(0,))

        tempimg = (tempimg - 127.5) * 0.0078125
        tempimg1 = np.transpose(tempimg, (3, 1, 0, 2))

        out = self._onet.predict(tempimg1)
        out0 = np.transpose(out[0])
        out1 = np.transpose(out[1])
        out2 = np.transpose(out[2])

        score = out2[1, :]

        points = out1

        ipass = np.where(score > self._steps_threshold[2])

        points = points[:, ipass[0]]

        total_boxes = np.hstack([total_boxes[ipass[0], 0:4].copy(), np.expand_dims(score[ipass].copy(), 1)])

        mv = out0[:, ipass[0]]

        w = total_boxes[:, 2] - total_boxes[:, 0] + 1
        h = total_boxes[:, 3] - total_boxes[:, 1] + 1

        points[0:5, :] = np.tile(w, (5, 1)) * points[0:5, :] + np.tile(total_boxes[:, 0], (5, 1)) - 1
        points[5:10, :] = np.tile(h, (5, 1)) * points[5:10, :] + np.tile(total_boxes[:, 1], (5, 1)) - 1

        if total_boxes.shape[0] > 0:
            total_boxes = self.__bbreg(total_boxes.copy(), np.transpose(mv))
            pick = self.__nms(total_boxes.copy(), 0.7, 'Min')
            total_boxes = total_boxes[pick, :]
            points = points[:, pick]

        return total_boxes, points
```