从零开始学习 **SSD**

GiantPandaCV 公众号

GiantPandaCV-BBuf

2020-04-14

简介

本电子书要解析的 SSD 代码库来自 github 一个非常火的 Pytorch 实现，已经有 3.5KStar。地址为：https://github.com/amdegroot/ssd.pytorch/

GiantPandaCV 公众号由专注于技术的一群 95 后创建，专注于机器学习、深度学习、计算机视觉、图像处理等领域。每天更新一到两篇相关推文，希望在传播知识、分享知识的同时能够启发你。

欢迎大家关注我们的公众号 GiantPandaCV:

1. 数据下载

在工程的根目录下执行下面的命令下载 VOC2007 的 trainval & test，以及 VOC2012 的 trainval。

```
sh data/scripts/VOC2007.sh
sh data/scripts/VOC2012.sh
```

把下载下来的数据整理一下，将 VOC2007 和 VOC2012 放在同一目录下，具体目录结构为：

```
├── data
│   ├── VOC
```

```
│           ├── VOCdevkit
│                ├── VOC2007
│                ├── VOC2012
```

2. 制作 **pytorch** 可以读取的数据集

这部分的代码都在 `data/voc0712.py` 文件里面。代码主要实现了 2 个 `class`，第一个 `class` 是 `VOCAnnotationTransform()`，主要功能是为了提取 VOC 数据集中每张原图的 xml 文件的 bbox 坐标进行归一化，并将类别转化为字典格式，最后把数据组合起来。形状最后类似于：`[[x_min,y_min,x_max,y_max, c], ...]`。第一部分的代码如下：

```
"""VOC Dataset Classes
Original author: Francisco Massa
https://github.com/fmassa/vision/blob/voc_dataset/torchvision/datasets/voc.py
Updated by: Ellis Brown, Max deGroot
"""
from .config import HOME
import os.path as osp
import sys
import torch
import torch.utils.data as data
import cv2
import numpy as np
if sys.version_info[0] == 2:
    import xml.etree.cElementTree as ET
else:
    import xml.etree.ElementTree as ET
# VOC 的数据类别
VOC_CLASSES = (  # always index 0
    'aeroplane', 'bicycle', 'bird', 'boat',
    'bottle', 'bus', 'car', 'cat', 'chair',
    'cow', 'diningtable', 'dog', 'horse',
    'motorbike', 'person', 'pottedplant',
    'sheep', 'sofa', 'train', 'tvmonitor')
# VOC 数据所在的目录
# note: if you used our download scripts, this should be right
VOC_ROOT = osp.join(HOME, "data/VOCdevkit/")


class VOCAnnotationTransform(object):
    """
```

```
    将 VOC 的 annotation 中的 bbox 坐标转化为归一化的值：
    将类别转化为用索引来表示的字典形式；
    参数列表：
            class_to_ind: 类别的索引字典。
            keep_difficult: 是否保留 difficult=1 的物体。
    """

    def __init__(self, class_to_ind=None, keep_difficult=False):
        self.class_to_ind = class_to_ind or dict(
            zip(VOC_CLASSES, range(len(VOC_CLASSES))))
        self.keep_difficult = keep_difficult

    def __call__(self, target, width, height):
        """
        参数列表：
                target: xml 被读取的一个 ET.Element 对象
                width: 图片宽度
                height: 图片高度
        返回值：
                一个 list，list 中的每个元素是 [bbox coords, class name]
        """
        res = []
        for obj in target.iter('object'):
            # 判断目标是否 difficult
            difficult = int(obj.find('difficult').text) == 1
            if not self.keep_difficult and difficult:
                continue
            # 读取 xml 中所需的信息
            name = obj.find('name').text.lower().strip()
            bbox = obj.find('bndbox')
            # xml 文件中 bbox 的表示
            pts = ['xmin', 'ymin', 'xmax', 'ymax']
            bndbox = []
            for i, pt in enumerate(pts):
                cur_pt = int(bbox.find(pt).text) - 1
                # 归一化，x/w，y/h
                cur_pt = cur_pt / width if i % 2 == 0 else cur_pt / height
                bndbox.append(cur_pt)
            # 提取类别名称对应的 index
            label_idx = self.class_to_ind[name]
            bndbox.append(label_idx)
            res += [bndbox]  # [xmin, ymin, xmax, ymax, label_ind]
            # img_id = target.find('filename').text[:-4]
```

```
            return res # [[xmin, ymin, xmax, ymax, label_ind], ... ]
```

第 2 个 class 主要根据上面的 `VOCAnnotationTransform()` 和 VOC 数据集的文件结构，读取图片，bbox 和 `lable`，构建 VOC 数据集的 DataLoader。这部分的代码仍在 `data/voc0712.py` 文件中，具体如下：

```
class VOCDetection(data.Dataset):
    # target_transform 传入上面的 VOCAnnotationTransform() 类
    def __init__(self, root,
                 image_sets=[('2007', 'trainval'), ('2012', 'trainval')],
                 transform=None, target_transform=VOCAnnotationTransform(),
                 dataset_name='VOC0712'):
        self.root = root
        self.image_set = image_sets
        self.transform = transform
        self.target_transform = target_transform
        self.name = dataset_name
        # bbox 和 label
        self._annopath = osp.join('%s', 'Annotations', '%s.xml')
        # 图片路径
        self._imgpath = osp.join('%s', 'JPEGImages', '%s.jpg')
        self.ids = list()
        for (year, name) in image_sets:
            rootpath = osp.join(self.root, 'VOC' + year)
            for line in open(osp.join(rootpath, 'ImageSets', 'Main', name +
             ↪  '.txt')):
                self.ids.append((rootpath, line.strip()))

    # 可以自定义的函数
    def __getitem__(self, index):
        im, gt, h, w = self.pull_item(index)

        return im, gt

    def __len__(self):
        return len(self.ids)

    def pull_item(self, index):
        img_id = self.ids[index]
        # label 信息
        target = ET.parse(self._annopath % img_id).getroot()
        # 读取图片信息
```

```
        img = cv2.imread(self._imgpath % img_id)
        # 图片的长宽通道数
        height, width, channels = img.shape
        # 标签执行 VOCAnnotationTransform() 操作
        if self.target_transform is not None:
            target = self.target_transform(target, width, height)
        # 数据（包括标签）是否需要执行 transform（数据增强）操作
        if self.transform is not None:
            target = np.array(target)
            # 执行了数据增强操作
            img, boxes, labels = self.transform(img, target[:, :4],
↪   target[:, 4])
            # 把图片转化为 RGB
            img = img[:, :, (2, 1, 0)]
            # 把 bbox 和 label 合并为 shape(N, 5)
            target = np.hstack((boxes, np.expand_dims(labels, axis=1)))
        return torch.from_numpy(img).permute(2, 0, 1), target, height, width
        # return torch.from_numpy(img), target, height, width

    def pull_image(self, index):
        '''
        以 PIL 图像的方式返回下标为 index 的 PIL 格式原始图像
        '''
        img_id = self.ids[index]
        return cv2.imread(self._imgpath % img_id, cv2.IMREAD_COLOR)

    def pull_anno(self, index):
        '''
        返回索引为 index 的图像的 xml 标注信息对象
       shape: [img_id, [(label, bbox coords),...]]
       例子: ('001718', [('dog', (96, 13, 438, 332))])
        '''
        img_id = self.ids[index]
        anno = ET.parse(self._annopath % img_id).getroot()
        gt = self.target_transform(anno, 1, 1)
        return img_id[1], gt

    def pull_tensor(self, index):
        '''
        以 Tensor 的形式返回索引为 index 的原始图像，调用 unsqueeze_ 函数
        return torch.Tensor(self.pull_image(index)).unsqueeze_(0)
```

下面提供了一段代码可以测试上面的两个类的效果。

```
Data = VOCDetection(VOC_ROOT)
data_loader = data.DataLoader(Data, batch_size=1,
                                    num_workers=0,
                                    shuffle=True,
                                    pin_memory=True)
print('the data length is:', len(data_loader))

# 类别 to index
class_to_ind = dict(zip(VOC_CLASSES, range(len(VOC_CLASSES))))

# index to class，转化为类别名称
ind_to_class = ind_to_class ={v:k for k, v in class_to_ind.items()}

# 加载数据
for datas in data_loader:
    img, target,h, w = datas
    img = img.squeeze(0).permute(1,2,0).numpy().astype(np.uint8)
    target = target[0].float()

    # 把 bbox 的坐标还原为原图的数值
    target[:,0] *= w.float()
    target[:,2] *= w.float()
    target[:,1] *= h.float()
    target[:,3] *= h.float()

    # 取整
    target = np.int0(target.numpy())
    # 画出图中类别名称
    for i in range(target.shape[0]):
        # 画矩形框
        img =cv2.rectangle(img, (target[i,0],target[i,1]),(target[i, 2],
↪   target[i, 3]), (0,0,255), 2)
        # 标明类别名称
        img =cv2.putText(img, ind_to_class[target[i,4]],(target[i,0],
↪   target[i,1]-25),
                         cv2.FONT_HERSHEY_SIMPLEX, .5, (255, 255, 0), 1)
    # 显示
    cv2.imshow('imgs', img)
    cv2.waitKey(0);
    cv2.destroyAllWindows()
    break
```

3. 数据增强

这部分介绍的代码都在 utils/augmentations.py 里面了。我们首先看一下 SSD 数据增强的 python 类代码：

```
class PhotometricDistort(object):
    """
    图片亮度，对比度和色调变化的方式合并为一个类
    """
    def __init__(self):
        self.pd = [
            RandomContrast(),
            ConvertColor(transform='HSV'),
            RandomSaturation(),
            RandomHue(),
            ConvertColor(current='HSV', transform='BGR'),
            RandomContrast()
        ]
        self.rand_brightness = RandomBrightness()
        self.rand_light_noise = RandomLightingNoise()

    def __call__(self, image, boxes, labels):
       # 使用图像的副本来做数据增强操作
        im = image.copy()
        # 亮度扰动增强
        im, boxes, labels = self.rand_brightness(im, boxes, labels)
        # 如果随机到 1(只可能随机到 0,1) 就不执行 pd 的最后一个操作
        if random.randint(2):
            distort = Compose(self.pd[:-1])
        else:
            # 随机到 0
            distort = Compose(self.pd[1:])
        # 执行一系列 pd 中的操作
        im, boxes, labels = distort(im, boxes, labels)
        # 最后再执行一个图片更换通道，形成颜色变化
        return self.rand_light_noise(im, boxes, labels)


# 结合所有的图片增广方法形成的类
class SSDAugmentation(object):
    def __init__(self, size=300, mean=(104, 117, 123)):
        self.mean = mean
        self.size = size
```

```
        self.augment = Compose([
            ConvertFromInts(),  # 转化为 float32
            ToAbsoluteCoords(), # 转化为原图坐标
            PhotometricDistort(), # 图片增强方式
            Expand(self.mean),  # 扩充
            RandomSampleCrop(), # 裁剪
            RandomMirror(), # 镜像
            ToPercentCoords(), # 转化为归一化后的坐标
            Resize(self.size), # Resize
            ToAbsoluteCoords(), # 转为原图坐标
            #SubtractMeans(self.mean), # 减去均值
        ])

    def __call__(self, image, boxes, labels):
        return self.augment(image, boxes, labels)
```

接下来我们就分别看看 SSDAugmentation 类的每一个增强子类是如何实现的。

3.1 Compose 类

从 SSDAugmentation 类来看，代码中有很多图片增强方式，如对比度，亮度，色度，那么如何将这些增强方法组合起来呢？就用这个类来实现。代码如下：

```
class Compose(object):
    """ 将不同的增强方法组合在一起
    参数:
        transforms (List[Transform]): list of transforms to compose.
    例子:
        >>> augmentations.Compose([
        >>>     transforms.CenterCrop(10),
        >>>     transforms.ToTensor(),
        >>> ])
    """

    def __init__(self, transforms):
        self.transforms = transforms

    def __call__(self, img, boxes=None, labels=None):
        for t in self.transforms:
            img, boxes, labels = t(img, boxes, labels)
        return img, boxes, labels
```

3.2 数据类型转换

在数据进行增强之前需要把图片的 uchar 类型转换为 float 类型。代码实现如下：

```
class ConvertFromInts(object):
    def __call__(self, image, boxes=None, labels=None):
        return image.astype(np.float32), boxes, labels
```

3.3 转回原图坐标

```
class ToAbsoluteCoords(object):
    # 把归一化后的 box 变回原图
    def __call__(self, image, boxes=None, labels=None):
        height, width, channels = image.shape
        boxes[:, 0] *= width
        boxes[:, 2] *= width
        boxes[:, 1] *= height
        boxes[:, 3] *= height

        return image, boxes, labels
```

3.4 图片色彩空间转换

在进行亮度，对比度，色度调整之前需要把色彩空间转换为 HSV 空间。最后还需要把 HSV 颜色空间转回 RGB 颜色空间。这部分的代码为：

```
class ConvertColor(object):
    # RGB 和 HSV 颜色空间互转
    def __init__(self, current='BGR', transform='HSV'):
        self.transform = transform
        self.current = current

    def __call__(self, image, boxes=None, labels=None):
        if self.current == 'BGR' and self.transform == 'HSV':
            image = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
        elif self.current == 'HSV' and self.transform == 'BGR':
            image = cv2.cvtColor(image, cv2.COLOR_HSV2BGR)
        else:
            raise NotImplementedError
        return image, boxes, labels
```

3.5 饱和度变化

饱和度变化需要在 HSV 颜色空间下改变 S 的数值，同时 S 的范围是 [0,1]，需要限定在里面。关于 HSV 表示什么，可以看一下我这篇推文：OpenCV 图像处理专栏一 | 盘点常见颜色空间互转。这部分的代码实现如下：

```
class RandomSaturation(object):
    # 随机饱和度变化，需要输入图片格式为 HSV
    def __init__(self, lower=0.5, upper=1.5):
        self.lower = lower
        self.upper = upper
        assert self.upper >= self.lower, "contrast upper must be >= lower."
        assert self.lower >= 0, "contrast lower must be non-negative."

    def __call__(self, image, boxes=None, labels=None):
        if random.randint(2):
            image[:, :, 1] *= random.uniform(self.lower, self.upper)

        return image, boxes, labels
```

3.6 色调变化

Hue 变化需要在 HSV 空间下，改变 H 的数值，H 的取值范围是 0-360。代码如下：

```
class RandomHue(object):
    def __init__(self, delta=18.0):
        assert delta >= 0.0 and delta <= 360.0
        self.delta = delta

    def __call__(self, image, boxes=None, labels=None):
        if random.randint(2):
            image[:, :, 0] += random.uniform(-self.delta, self.delta)
            image[:, :, 0][image[:, :, 0] > 360.0] -= 360.0
            image[:, :, 0][image[:, :, 0] < 0.0] += 360.0
            return image, boxes, labels
```

3.7 对比度变化

图片的对比度变化，只需要在 RGB 颜色空间下，乘上一个 alpha 值。代码如下：

```
class RandomContrast(object):
    def __init__(self, lower=0.5, upper=1.5):
```

```
        self.lower = lower
        self.upper = upper
        assert self.upper >= self.lower, "contrast upper must be >= lower."
        assert self.lower >= 0, "contrast lower must be non-negative."

    # expects float image
    def __call__(self, image, boxes=None, labels=None):
        if random.randint(2):
            alpha = random.uniform(self.lower, self.upper)
            image *= alpha
        return image, boxes, labels
```

3.8 亮度变化

接下来的增强操作是亮度变化，亮度变化只需要在 RGB 空间下，加上一个 delta 值，代码如下：

```
class RandomBrightness(object):
    def __init__(self, delta=32):
        assert delta >= 0.0
        assert delta <= 255.0
        self.delta = delta

    def __call__(self, image, boxes=None, labels=None):
        if random.randint(2):
            delta = random.uniform(-self.delta, self.delta)
            image += delta
        return image, boxes, labels
```

3.9 颜色通道变化

针对图片的 RGB 空间，随机调换各通道的位置，实现不同灯光效果，代码如下：

```
class SwapChannels(object):
    """ 图像通道变换
     specified in the swap tuple.
    Args:
        swaps (int triple): final order of channels
            eg: (2, 1, 0)
    """

    def __init__(self, swaps):
        self.swaps = swaps
```

```
    def __call__(self, image):
        image = image[:, :, self.swaps]
        return image

class RandomLightingNoise(object):
    # 图片更换通道，形成的颜色变化
    def __init__(self):
        self.perms = ((0, 1, 2), (0, 2, 1),
                      (1, 0, 2), (1, 2, 0),
                      (2, 0, 1), (2, 1, 0))

    def __call__(self, image, boxes=None, labels=None):
        if random.randint(2):
            swap = self.perms[random.randint(len(self.perms))]
            shuffle = SwapChannels(swap)  # shuffle channels
            image = shuffle(image)
        return image, boxes, labels
```

3.10 图像扩充

设置一个大于原图尺寸的 size，填充指定的像素值 mean，然后把原图随机放入这个图片中，实现原图的扩充。

```
class Expand(object):
    # 随机扩充图片
    def __init__(self, mean):
        self.mean = mean

    def __call__(self, image, boxes, labels):
        if random.randint(2):
            return image, boxes, labels

        height, width, depth = image.shape
        ratio = random.uniform(1, 4)
        left = random.uniform(0, width*ratio - width)
        top = random.uniform(0, height*ratio - height)
        # 填充 mean 值
        expand_image = np.zeros(
            (int(height*ratio), int(width*ratio), depth),
            dtype=image.dtype)
        # 放入原图
```

```
        expand_image[:, :, :] = self.mean
        expand_image[int(top):int(top + height),
                     int(left):int(left + width)] = image
        image = expand_image
        # 同样相应的变化 boxes 的坐标
        boxes = boxes.copy()
        boxes[:, :2] += (int(left), int(top))
        boxes[:, 2:] += (int(left), int(top))

        return image, boxes, labels
```

3.11 图片的随机裁剪

图片随机裁剪在数据增强中有重要作用，这个算法的运行流程大致如下：

- 随机选取裁剪框的大小；
- 根据大小确定裁剪框的坐标；
- 分析裁剪框和图片内部 bounding box 的交并比；
- 筛选掉交并比不符合要求的裁剪框；
- 裁剪图片，并重新更新 bounding box 的位置坐标；

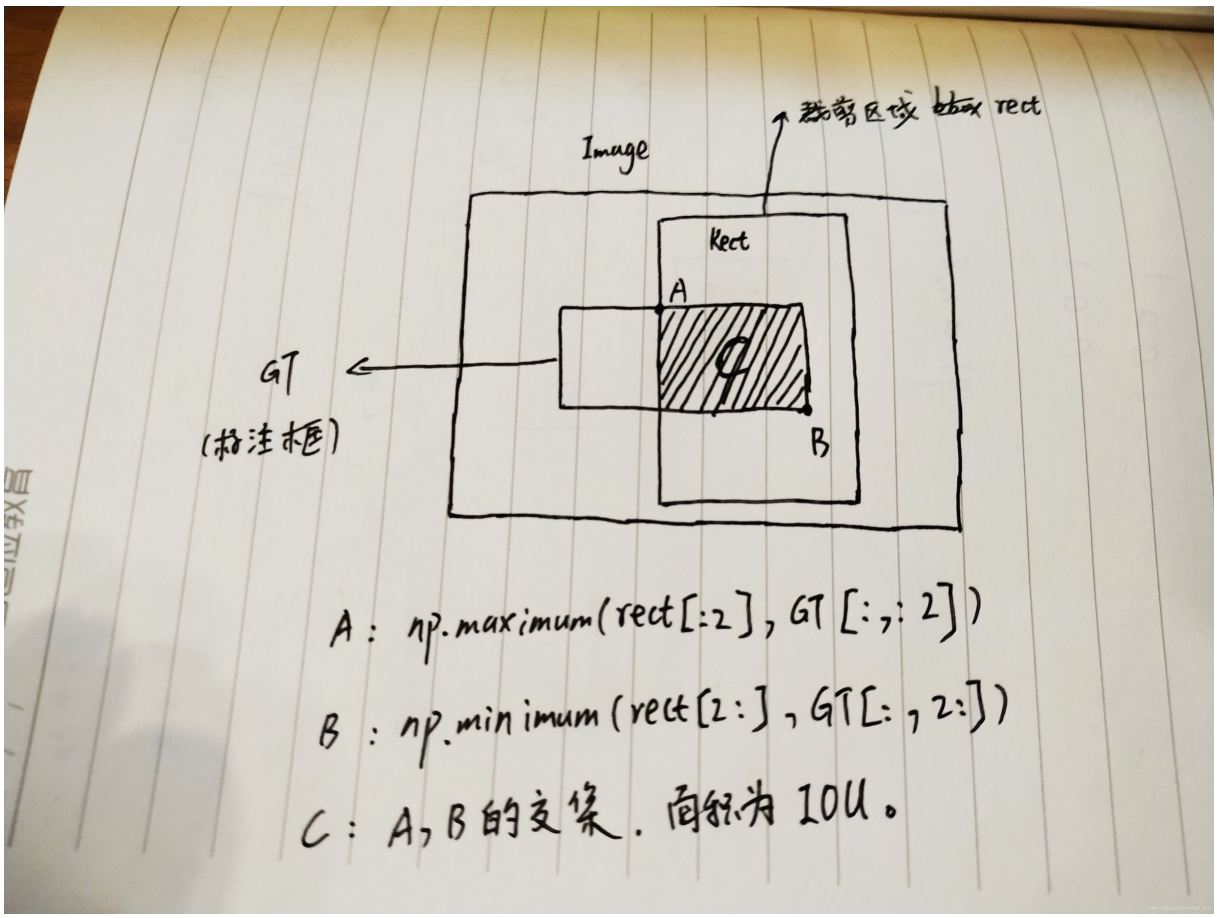


Figure 1: 演示图

```
class RandomSampleCrop(object):
    """Crop
    Arguments:
        img (Image): the image being input during training
        boxes (Tensor): the original bounding boxes in pt form
        labels (Tensor): the class labels for each bbox
        mode (float tuple): the min and max jaccard overlaps
    Return:
        (img, boxes, classes)
            img (Image): the cropped image
            boxes (Tensor): the adjusted bounding boxes in pt form
            labels (Tensor): the class labels for each bbox
    """
    def __init__(self):
        self.sample_options = (
            # 使用原图
```

```
            None,
            # 最小的 IOU，和最大的 IOU
            (0.1, None),
            (0.3, None),
            (0.7, None),
            (0.9, None),
            # randomly sample a patch
            (None, None),
        )

    def __call__(self, image, boxes=None, labels=None):
        # 原图的长宽
        height, width, _ = image.shape
        while True:
            # 随机选择一个切割模式
            mode = random.choice(self.sample_options)
            if mode is None:
                return image, boxes, labels

            min_iou, max_iou = mode
            if min_iou is None:
                min_iou = float('-inf')
            if max_iou is None:
                max_iou = float('inf')

            # 迭代 50 次
            for _ in range(50):
                current_image = image
                # 随机一个长宽
                w = random.uniform(0.3 * width, width)
                h = random.uniform(0.3 * height, height)

                # 判断长宽比在一定范围
                if h / w < 0.5 or h / w > 2:
                    continue

                left = random.uniform(width - w)
                top = random.uniform(height - h)

                # 切割的矩形大小，形状是 [x1,y1,x2,y2]
                rect = np.array([int(left), int(top), int(left+w),
↪   int(top+h)])
```

```
                # 计算切割的矩形和 gt 框的 iou 大小
                overlap = jaccard_numpy(boxes, rect)

                # 筛选掉不满足 overlap 条件的
                if overlap.min() < min_iou and max_iou < overlap.max():
                    continue

                # 从原图中裁剪矩形
                current_image = current_image[rect[1]:rect[3],
↪  rect[0]:rect[2],
                                              :]

                # 所有 GT 的中心点坐标
                centers = (boxes[:, :2] + boxes[:, 2:]) / 2.0

                # 这个地方的原理是判断每个 GT 的中心坐标是否在裁剪框 Rect 里面，
                # 如果超出了那么下面的 mask 就全为 0，那么 mask.any() 返回 false，
                # 也即是说这次裁剪失败了。
                m1 = (rect[0] < centers[:, 0]) * (rect[1] < centers[:, 1])


                m2 = (rect[2] > centers[:, 0]) * (rect[3] > centers[:, 1])

                # mask in that both m1 and m2 are true
                mask = m1 * m2

                # 是否有合法的盒子
                if not mask.any():
                    continue

                # 取走中心点落在裁剪区域中的 GT
                current_boxes = boxes[mask, :].copy()

                # 取出中心点落在裁剪区域中的 GT 对应的标签
                current_labels = labels[mask]

                # 获取 GT box 和切割矩形的交点 (左上角)  A 点
                current_boxes[:, :2] = np.maximum(current_boxes[:, :2],
                                                  rect[:2])
                # 调节坐标系，让 boxes 的左上角坐标变为切割后的坐标
                current_boxes[:, :2] -= rect[:2]

                current_boxes[:, 2:] = np.minimum(current_boxes[:, 2:],
```

```
                                                rect[2:])
            # 调节坐标系，让 boxes 的左上角坐标变为切割后的坐标
            current_boxes[:, 2:] -= rect[:2]
            # 返回结果
            return current_image, current_boxes, current_labels
```

3.12 图片镜像

这种方式比较简单，就是将图片进行左右翻转，实现数据增强。

```
class RandomMirror(object):
    # 随机镜像图片
    def __call__(self, image, boxes, labels):
        w = image.shape[1]
        if random.randint(2):
            # 图片翻转
            image = image[:, ::-1]

            # boxes 的坐标也需要相应改变
            boxes = boxes.copy()
            boxes[:, 0::2] = w - boxes[:, 2::-2]

        return image, boxes, labels
```

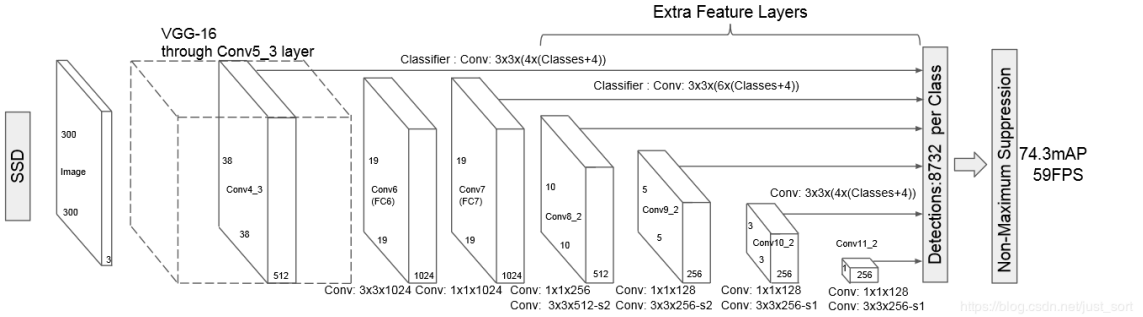
执行了上面所有的数据增强操作之后就得到了代码中最终的数据增强图了，整个数据增强部分的工作也就结束了，有点快哈哈。

3.13 一个疑问？

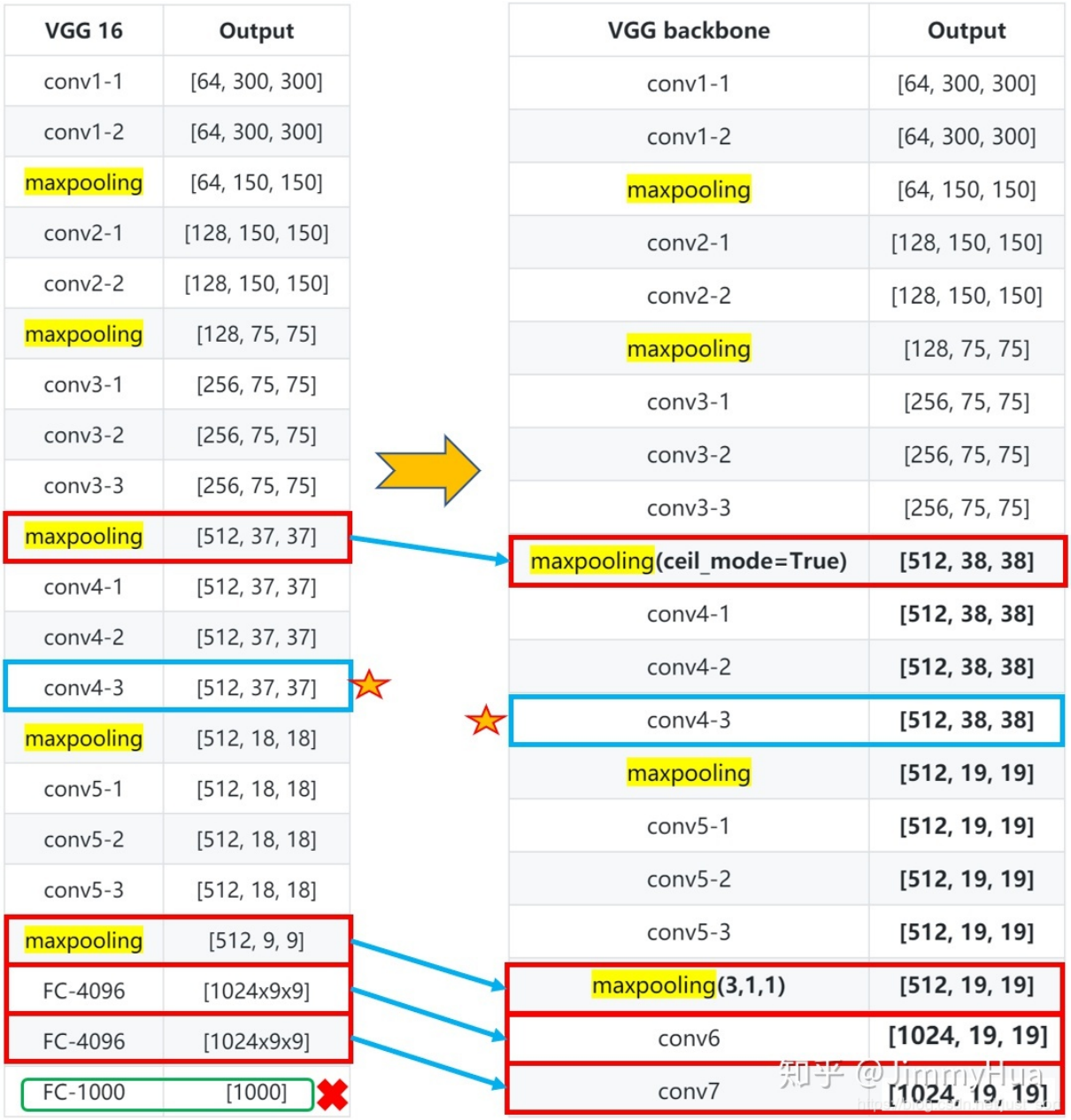
这里一个值得怀疑的地方是在亮度和对比度增强的时候没有将像素的值域限制在 0-255 范围内，似乎这一点是存在问题的？因此为了解决这一问题，我在每一个数据增强后面进行了打印，我发现在亮度增强和对比度增强之后像素值确实有超过了 255 的，但是最后减掉均值之后像素值的范围是在 0-255 的。我对这个过程充满了怀疑，按照常识亮度增强和对比度增强是需要 crop 的，这样难道不会影响最后的结果吗？欢迎加我好友讨论，微信号：hellotopython。

4. SSD 网络结构

为了比较好的对应 SSD 的结构来看代码，我们首先放出 SSD 的网络结构，如下图所示：



可以看到原始的 SSD 网络是以 VGG-16 作 Backbone（骨干网络）的。为了更加清晰看到相比于 VGG16，SSD 的网络使用了哪些变化，知乎上的一个帖子做了一个非常清晰的图，这里借用一下，原图地址为：https://zhuanlan.zhihu.com/p/79854543 。带有特征图维度信息的更清晰的骨干网络和 VGG16 的对比图如下：



VGG 16	Output
conv1-1	[64, 300, 300]
conv1-2	[64, 300, 300]
maxpooling	[64, 150, 150]
conv2-1	[128, 150, 150]
conv2-2	[128, 150, 150]
maxpooling	[128, 75, 75]
conv3-1	[256, 75, 75]
conv3-2	[256, 75, 75]
conv3-3	[256, 75, 75]
maxpooling	[512, 37, 37]
conv4-1	[512, 37, 37]
conv4-2	[512, 37, 37]
conv4-3	[512, 37, 37]
maxpooling	[512, 18, 18]
conv5-1	[512, 18, 18]
conv5-2	[512, 18, 18]
conv5-3	[512, 18, 18]
maxpooling	[512, 9, 9]
FC-4096	[1024x9x9]
FC-4096	[1024x9x9]
FC-1000	[1000]

VGG backbone	Output
conv1-1	[64, 300, 300]
conv1-2	[64, 300, 300]
maxpooling	[64, 150, 150]
conv2-1	[128, 150, 150]
conv2-2	[128, 150, 150]
maxpooling	[128, 75, 75]
conv3-1	[256, 75, 75]
conv3-2	[256, 75, 75]
conv3-3	[256, 75, 75]
maxpooling(ceil_mode=True)	**[512, 38, 38]**
conv4-1	**[512, 38, 38]**
conv4-2	**[512, 38, 38]**
conv4-3	**[512, 38, 38]**
maxpooling	**[512, 19, 19]**
conv5-1	**[512, 19, 19]**
conv5-2	**[512, 19, 19]**
conv5-3	**[512, 19, 19]**
maxpooling(3,1,1)	**[512, 19, 19]**
conv6	**[1024, 19, 19]**
conv7	**[1024, 19, 19]**

Figure 2: 带有特征图维度信息的更清晰的骨干网络和 VGG16 的对比图，左边是 VGG16

5. 源码解析

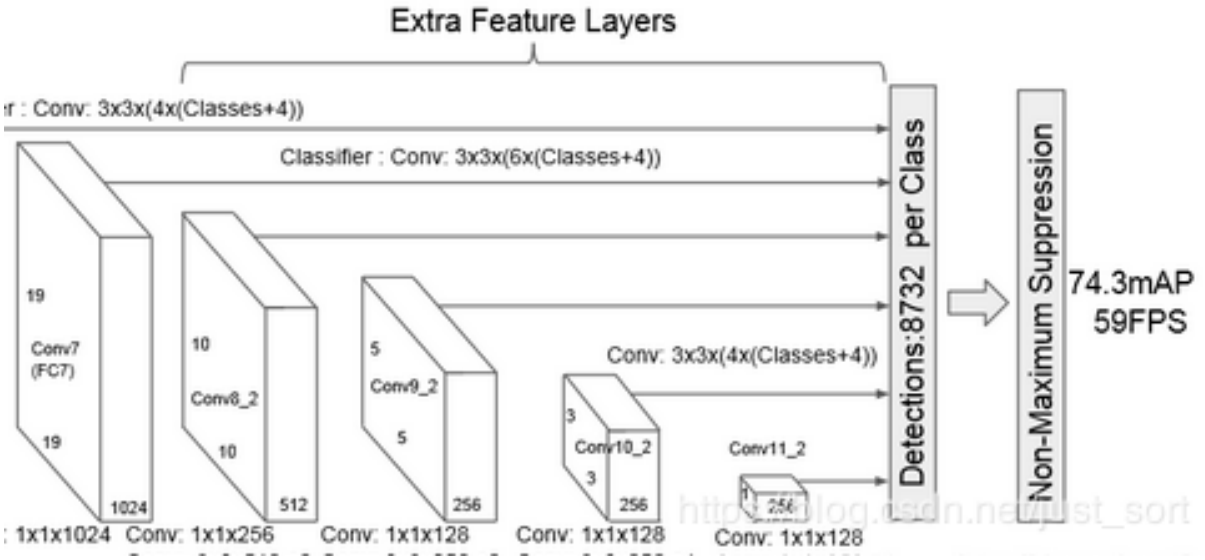
OK，现在我们就要开始从源码剖析 SSD 了。主要弄清楚三个方面，网络结构的搭建，Anchor，损失函数，标准化，L2 位置信息编解码，以及后处理 NMS 就算是理解这个源码了。

5.1 网络搭建

从上面的图中我们可以清晰的看到在以 VGG16 做骨干网络时，在 conv5 后丢弃了 VGG16 中的全连接层改为了 $1024 \times 3 \times 3$ 和 $1024 \times 1 \times 1$ 的卷积层。其中 conv4-1 卷积层前面的 maxpooling 层的 ceil_model=True，使得输出特征图长宽为 38×38。还有 conv5-3 后面的一层 maxpooling 层参数为 (kernelsize=3,stride=1,padding=1)，不进行下采样。然后在 fc7 后面接上多尺度提取的另外 4 个卷积层就构成了完整的 SSD 网络。这里 VGG16 修改后的代码如下，来自 ssd.py：

```
def vgg(cfg, i, batch_norm=False):
    layers = []
    in_channels = i
    for v in cfg:
        if v == 'M':
            layers += [nn.MaxPool2d(kernel_size=2, stride=2)]
        elif v == 'C':
            layers += [nn.MaxPool2d(kernel_size=2, stride=2, ceil_mode=True)]
        else:
            conv2d = nn.Conv2d(in_channels, v, kernel_size=3, padding=1)
            if batch_norm:
                layers += [conv2d, nn.BatchNorm2d(v), nn.ReLU(inplace=True)]
            else:
                layers += [conv2d, nn.ReLU(inplace=True)]
            in_channels = v
    pool5 = nn.MaxPool2d(kernel_size=3, stride=1, padding=1)
    conv6 = nn.Conv2d(512, 1024, kernel_size=3, padding=6, dilation=6)
    conv7 = nn.Conv2d(1024, 1024, kernel_size=1)
    layers += [pool5, conv6,
               nn.ReLU(inplace=True), conv7, nn.ReLU(inplace=True)]
    return layers
```

可以看到和我们上面的那张图是完全一致的。代码里面最后获得的 conv7 就是我们上面图里面的 fc7，特征维度是：$[None, 1024, 19, 19]$。现在可以开始搭建 SSD 网络后面的多尺度提取网络了。也就是网络结构图中的 Extra Feature Layers。我们从上一节的 SSD 结构图中截取一下这一部分，方便我们对照代码。



实现的代码如下（同样来自 ssd.py）：

```
def add_extras(cfg, i, batch_norm=False):
    # Extra layers added to VGG for feature scaling
    layers = []
    in_channels = i
    flag = False #flag 用来控制 kernel_size= 1 or 3
    for k, v in enumerate(cfg):
        if in_channels != 'S':
            if v == 'S':
                layers += [nn.Conv2d(in_channels, cfg[k + 1],
                           kernel_size=(1, 3)[flag], stride=2, padding=1)]
            else:
                layers += [nn.Conv2d(in_channels, v, kernel_size=(1,
 ↪  3)[flag])]
            flag = not flag
        in_channels = v
return layers
```

可以看到网络结构中除了魔改后的 VGG16 和 Extra Layers 还有 6 个横着的线，这代表的是对 6 个尺度的特征图进行卷积获得预测框的回归 (loc) 和类别 (cls) 信息，注意 SSD 将背景也看成类别了，所以对于 VOC 数据集类别数就是 20+1=21。这部分的代码为：

```
def multibox(vgg, extra_layers, cfg, num_classes):
    loc_layers = []# 多尺度分支的回归网络
    conf_layers = []# 多尺度分支的分类网络
    # 第一部分，vgg 网络的 Conv2d-4_3(21 层)， Conv2d-7_1(-2 层)
```

```
    vgg_source = [21, -2]
    for k, v in enumerate(vgg_source):
        # 回归 box*4(坐标)
        loc_layers += [nn.Conv2d(vgg[v].out_channels,
                                 cfg[k] * 4, kernel_size=3, padding=1)]
        # 置信度 box*(num_classes)
        conf_layers += [nn.Conv2d(vgg[v].out_channels,
                        cfg[k] * num_classes, kernel_size=3, padding=1)]
    # 第二部分，cfg 从第三个开始作为 box 的个数，而且用于多尺度提取的网络分别为 1,3,5,7 层
    for k, v in enumerate(extra_layers[1::2], 2):
        loc_layers += [nn.Conv2d(v.out_channels, cfg[k]
                                 * 4, kernel_size=3, padding=1)]
        conf_layers += [nn.Conv2d(v.out_channels, cfg[k]
                                  * num_classes, kernel_size=3, padding=1)]
    return vgg, extra_layers, (loc_layers, conf_layers)
# 用下面的测试代码测试一下
if __name__  == "__main__":
    vgg, extra_layers, (l, c) = multibox(vgg(base['300'], 3),
                                         add_extras(extras['300'], 1024),
                                         [4, 6, 6, 6, 4, 4], 21)
    print(nn.Sequential(*l))
    print('----------------------------')
    print(nn.Sequential(*c))
```

在 jupter notebook 输出信息为：

```
'''
loc layers:
'''
Sequential(
  (0): Conv2d(512, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (1): Conv2d(1024, 24, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (2): Conv2d(512, 24, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (3): Conv2d(256, 24, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (4): Conv2d(256, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (5): Conv2d(256, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
)
----------------------------
'''
conf layers:
'''
Sequential(
  (0): Conv2d(512, 84, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (1): Conv2d(1024, 126, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
```

```
  (2): Conv2d(512, 126, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (3): Conv2d(256, 126, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (4): Conv2d(256, 84, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (5): Conv2d(256, 84, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
)
```

5.2 Anchor 生成 (Prior_Box 层)

这个在目标检测算法之SSD中讲过了,这里不妨再回忆一下,SSD 从魔改后的VGG16的 conv4_3 开始一共使用了6个不同大小的特征图,大小分别为 (38,28),(19,19),(10,10),(5,5),(3,3),(1,1),但每个特征图上设置的先验框 (Anchor) 的数量不同。先验框的设置包含尺度和长宽比两个方面。对于先验框的设置，公式如下：$s_k = s_{min} + \frac{s_{max}-s_{min}}{m-1}(k-1), k \in [1, m]$，其中 M 指的是特征图个数，这里为5，因为第一层 conv4_3 的 Anchor 是单独设置的，s_k 代表先验框大小相对于特征图的比例，注意这里不是相对原图哦。最后，s_{min} 和 s_{max} 表示比例的最小值和最大值，论文中分别取 0.2 和 0.9。对于第一个特征图，它的先验框尺度比例设置为 $s_{min}/2 = 0.1$，则他的尺度为 $300 \times 0.1 = 30$，后面的特征图带入公式计算，并将其映射会原图 300 的大小可以得到，剩下的 5 个特征图的尺度 s_k 为 $60, 111, 162, 213, 264$。所以综合起来，6 个特征图的尺度 s_k 为 $30, 60, 111, 162, 213, 264$。有了 Anchor 的尺度，接下来设置 Anchor 的长宽，论文中长宽设置一般为 $a_r = 1, 2, 3, \frac{1}{2}, \frac{1}{3}$，根据面积和长宽比可以得到先验框的宽度和高度：$w_k^a = s_k\sqrt{a_r}, h_k^a = s_k/\sqrt{a_r}$。这里有一些值得注意的点，如下：

- 上面的 s_k 是相对于原图的大小。
- 默认情况下，每个特征图除了上面 5 个比例的 Anchor，还会设置一个尺度为 $s_k^{'} = \sqrt{s_k s_{k+1}}$ 且 $a_r = 1$ 的先验框，这样每个特征图都设置了两个长宽比为1但大小不同的正方形先验框。最后一个特征图需要参考一下 $s_{m+1} = 315$ 来计算 s_m。
- 在实现 conv4_3,conv10_2,conv11_2 层时仅使用 4 个先验框，不使用长宽比为 $3, \frac{1}{3}$ 的 Anchor。
- 每个单元的先验框中心点分布在每个单元的中心，即：$[\frac{i+0.5}{|f_k|}, \frac{j+0.5}{|f_k|}], i, j \in [0, |f_k|]$，其中 f_k 是特征图的大小。

从 Anchor 的值来看，越前面的特征图 Anchor 的尺寸越小，也就是说对小目标的效果越好。先验框的总数为 num_priors = 38x38x4+19x19x6+10x10x6+5x5x6+3x3x4+1x1x4=8732。

生成先验框的代码如下（来自 layers/functions/prior_box.py）

```python
class PriorBox(object):
    """Compute priorbox coordinates in center-offset form for each source
    feature map.
    """
    def __init__(self, cfg):
        super(PriorBox, self).__init__()
```

```
        self.image_size = cfg['min_dim']
        # number of priors for feature map location (either 4 or 6)
        self.num_priors = len(cfg['aspect_ratios'])
        self.variance = cfg['variance'] or [0.1]
        self.feature_maps = cfg['feature_maps']
        self.min_sizes = cfg['min_sizes']
        self.max_sizes = cfg['max_sizes']
        self.steps = cfg['steps']
        self.aspect_ratios = cfg['aspect_ratios']
        self.clip = cfg['clip']
        self.version = cfg['name']
        for v in self.variance:
            if v <= 0:
                raise ValueError('Variances must be greater than 0')

    def forward(self):
        mean = []
        # 遍历多尺度的 特征图: [38, 19, 10, 5, 3, 1]
        for k, f in enumerate(self.feature_maps):
            # 遍历每个像素
            for i, j in product(range(f), repeat=2):
                # k-th 层的 feature map 大小
                f_k = self.image_size / self.steps[k]
                # # 每个框的中心坐标
                cx = (j + 0.5) / f_k
                cy = (i + 0.5) / f_k

                # aspect_ratio: 1 当 ratio==1 的时候，会产生两个 box
                # r==1, size = s_k,  正方形
                s_k = self.min_sizes[k]/self.image_size
                mean += [cx, cy, s_k, s_k]

                # r==1, size = sqrt(s_k * s_(k+1)), 正方形
                # rel size: sqrt(s_k * s_(k+1))
                s_k_prime = sqrt(s_k * (self.max_sizes[k]/self.image_size))
                mean += [cx, cy, s_k_prime, s_k_prime]

                # 当 ratio != 1 的时候，产生的 box 为矩形
                for ar in self.aspect_ratios[k]:
                    mean += [cx, cy, s_k*sqrt(ar), s_k/sqrt(ar)]
                    mean += [cx, cy, s_k/sqrt(ar), s_k*sqrt(ar)]
        # 转化为 torch 的 Tensor
        output = torch.Tensor(mean).view(-1, 4)
```

```
        # 归一化，把输出设置在 [0,1]
        if self.clip:
            output.clamp_(max=1, min=0)
return output
```

5.3 网络结构

结合了前面介绍的魔改后的 VGG16，还有 Extra Layers，还有生成 Anchor 的 Priobox 策略，我们可以写出 SSD 的整体结构如下（代码在 ssd.py）：

```
class SSD(nn.Module):
    """Single Shot Multibox Architecture
    The network is composed of a base VGG network followed by the
    added multibox conv layers.  Each multibox layer branches into
        1) conv2d for class conf scores
        2) conv2d for localization predictions
        3) associated priorbox layer to produce default bounding
           boxes specific to the layer's feature map size.
    See: https://arxiv.org/pdf/1512.02325.pdf for more details.
    Args:
        phase: (string) Can be "test" or "train"
        size: input image size
        base: VGG16 layers for input, size of either 300 or 500
        extras: extra layers that feed to multibox loc and conf layers
        head: "multibox head" consists of loc and conf conv layers
    """

    def __init__(self, phase, size, base, extras, head, num_classes):
        super(SSD, self).__init__()
        self.phase = phase
        self.num_classes = num_classes
        # 配置 config
        self.cfg = (coco, voc)[num_classes == 21]
        # 初始化先验框
        self.priorbox = PriorBox(self.cfg)
        self.priors = Variable(self.priorbox.forward(), volatile=True)
        self.size = size

        # SSD network
        # backbone 网络
        self.vgg = nn.ModuleList(base)
        # Layer learns to scale the l2 normalized features from conv4_3
```

```
        # conv4_3 后面的网络，L2 正则化
        self.L2Norm = L2Norm(512, 20)
        self.extras = nn.ModuleList(extras)
        # 回归和分类网络
        self.loc = nn.ModuleList(head[0])
        self.conf = nn.ModuleList(head[1])

        if phase == 'test':
            self.softmax = nn.Softmax(dim=-1)
            self.detect = Detect(num_classes, 0, 200, 0.01, 0.45)

    def forward(self, x):
        """Applies network layers and ops on input image(s) x.
        Args:
            x: input image or batch of images. Shape: [batch,3,300,300].
        Return:
            Depending on phase:
            test:
                Variable(tensor) of output class label predictions,
                confidence score, and corresponding location predictions for
                each object detected. Shape: [batch,topk,7]
            train:
                list of concat outputs from:
                    1: confidence layers, Shape:
↪   [batch*num_priors,num_classes]
                    2: localization layers, Shape: [batch,num_priors*4]
                    3: priorbox layers, Shape: [2,num_priors*4]
        """
        sources = list()
        loc = list()
        conf = list()

        # apply vgg up to conv4_3 relu
        # vgg 网络到 conv4_3
        for k in range(23):
            x = self.vgg[k](x)
        # l2 正则化
        s = self.L2Norm(x)
        sources.append(s)

        # apply vgg up to fc7
        # conv4_3 到 fc
        for k in range(23, len(self.vgg)):
```

```
            x = self.vgg[k](x)
        sources.append(x)

        # apply extra layers and cache source layer outputs
        # extras 网络
        for k, v in enumerate(self.extras):
            x = F.relu(v(x), inplace=True)
            if k % 2 == 1:
                # 把需要进行多尺度的网络输出存入 sources
                sources.append(x)

        # apply multibox head to source layers
        # 多尺度回归和分类网络
        for (x, l, c) in zip(sources, self.loc, self.conf):
            loc.append(l(x).permute(0, 2, 3, 1).contiguous())
            conf.append(c(x).permute(0, 2, 3, 1).contiguous())

        loc = torch.cat([o.view(o.size(0), -1) for o in loc], 1)
        conf = torch.cat([o.view(o.size(0), -1) for o in conf], 1)
        if self.phase == "test":
            output = self.detect(
                loc.view(loc.size(0), -1, 4),                   # loc preds
                self.softmax(conf.view(conf.size(0), -1,
                             self.num_classes)),                # conf preds
               self.priors.type(type(x.data))                  # default boxes
            )
        else:
            output = (
                # loc 的输出，size:(batch, 8732, 4)
                loc.view(loc.size(0), -1, 4),
                # conf 的输出，size:(batch, 8732, 21)
                conf.view(conf.size(0), -1, self.num_classes),
                # 生成所有的候选框 size([8732, 4])
                self.priors
            )
        return output
    # 加载模型参数
    def load_weights(self, base_file):
        other, ext = os.path.splitext(base_file)
        if ext == '.pkl' or '.pth':
            print('Loading weights into state dict...')
            self.load_state_dict(torch.load(base_file,
                                 map_location=lambda storage, loc: storage))
```

```
            print('Finished!')
        else:
            print('Sorry only .pth and .pkl files supported.')
```

然后为了增加可读性，重新封装了一下，代码如下：

```
base = {
    '300': [64, 64, 'M', 128, 128, 'M', 256, 256, 256, 'C', 512, 512, 512,
    ↪  'M',
            512, 512, 512],
    '512': [],
}
extras = {
    '300': [256, 'S', 512, 128, 'S', 256, 128, 256, 128, 256],
    '512': [],
}
mbox = {
    '300': [4, 6, 6, 6, 4, 4],  # number of boxes per feature map location
    '512': [],
}


def build_ssd(phase, size=300, num_classes=21):
    if phase != "test" and phase != "train":
        print("ERROR: Phase: " + phase + " not recognized")
        return
    if size != 300:
        print("ERROR: You specified size " + repr(size) + ". However, " +
              "currently only SSD300 (size=300) is supported!")
        return
    # 调用 multibox，生成 vgg,extras,head
    base_, extras_, head_ = multibox(vgg(base[str(size)], 3),
                                     add_extras(extras[str(size)], 1024),
                                     mbox[str(size)], num_classes)
    return SSD(phase, size, base_, extras_, head_, num_classes)
```

5.4 Loss 解析

SSD 的损失函数包含两个部分，一个是定位损失 L_{loc}，一个是分类损失 L_{conf}，整个损失函数表达如下：$L(x,c,l,g) = \frac{1}{N}(L_{conf}(x,c) + \alpha L_{loc}(x,l,g))$ 其中，N 是先验框的正样本数量，c 是类别置信度预测值，l 是先验框对应的边界框预测值，g 是 ground truth 的位置参数，x 代表网络的预测值。对于位置损失，采用 Smooth L1 Loss，位置信息都是 encode 之

后的数值，后面会讲这个 encode 的过程。而对于分类损失，首先需要使用 `hard negtive mining` 将正负样本按照 `1:3` 的比例把负样本抽样出来，抽样的方法是：针对所有 batch 的 confidence，按照置信度误差进行降序排列，取出前 `top_k` 个负样本。损失函数可以用下图表示：

$$L(x,c,l,g)=\frac{1}{N}(L_{conf}(x,c)+\alpha L_{loc}(x,l,g)) \tag{1}$$

第i个预测框与第j个真实框关于类别k是否匹配: 0, 1

预测框 真实框

$$L_{loc}(x,l,g)=\sum_{i\in Pos}^{N}\sum_{m\in\{cx,cy,w,h\}}x_{ij}^{k}\text{smooth}_{\text{L1}}(l_i^m-\hat{g}_j^m)$$

预测框i与真实框j关于类别p匹配，则p的概率预测越高，损失越小

预测框其实没有物体，则预测为背景的概率越高，损失越小

概率通过Softmax产生

$$L_{conf}(x,c)=-\sum_{i\in Pos}^{N}x_{ij}^{p}log(\hat{c}_i^p)-\sum_{i\in Neg}log(\hat{c}_i^0)\quad\text{where}\quad\hat{c}_i^p=\frac{\exp(c_i^p)}{\sum_p\exp(c_i^p)} \tag{3}$$

SSD 图3

5.4.1 实现步骤

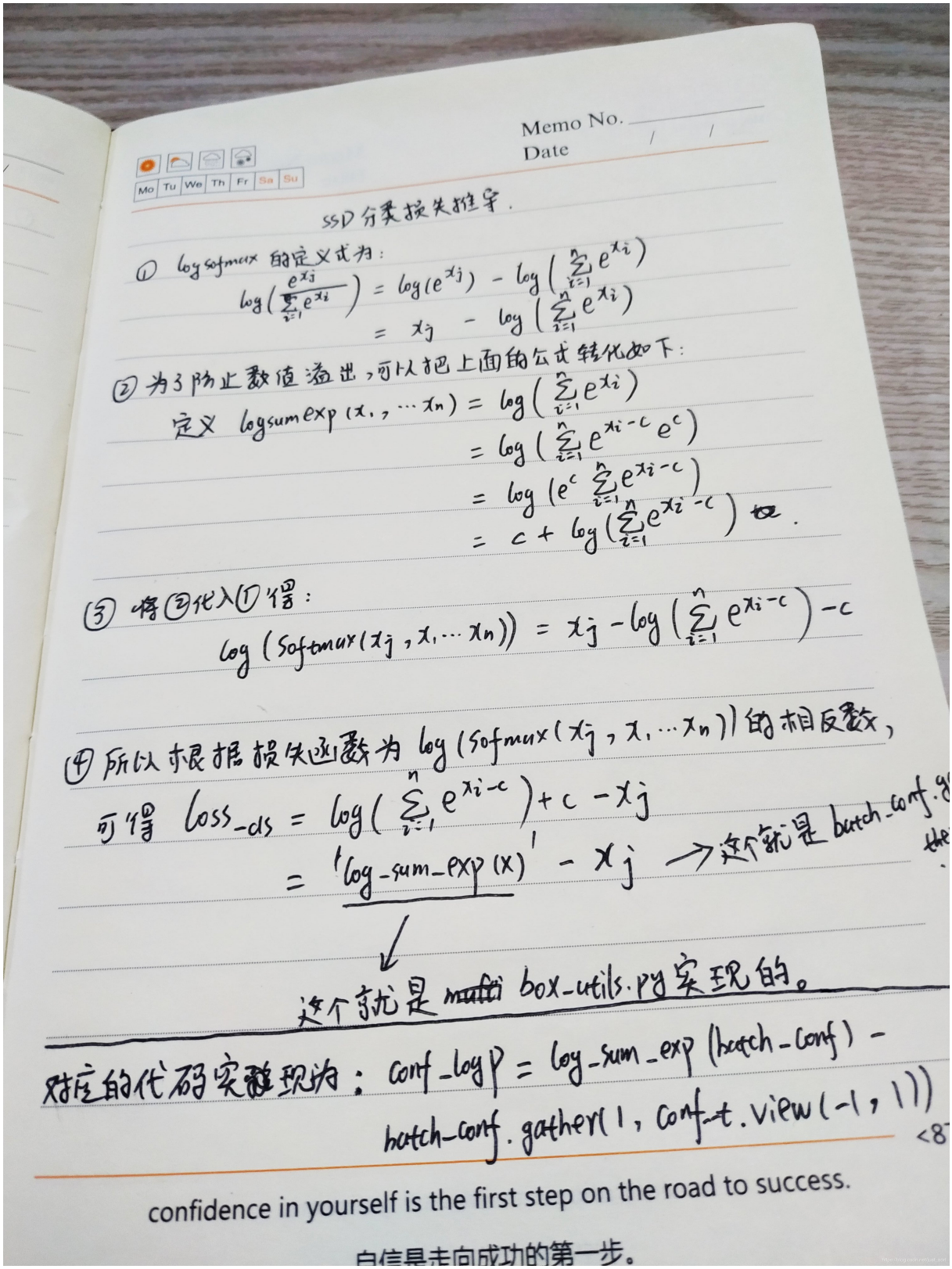
- Reshape 所有 batch 中的 conf，即代码中的 `batch_conf = conf_data.view(-1, self.num_classes)`，方便后续排序。
- 置信度误差越大，实际上就是预测背景的置信度越小。
- 把所有 conf 进行 `logsoftmax` 处理 (均为负值)，预测的置信度越小，则 `logsoftmax` 越小，取绝对值，则 `|logsoftmax|` 越大，降序排列`-logsoftmax`，取前 `top_k` 的负样本。其中，log_sum_exp 函数的代码如下 (在 `box_utils.py` 实现)：

```
def log_sum_exp(x):
    x_max = x.detach().max()
    return torch.log(torch.sum(torch.exp(x-x_max), 1, keepdim=True))+x_max
```

分类损失 `conf_logP` 函数如下：

```
conf_logP = log_sum_exp(batch_conf) - batch_conf.gather(1, conf_t.view(-1,
↪  1))
```

这样计算的原因主要是为了增强 `logsoftmax` 损失的数值稳定性。放一张我的手推图：


Memo No.
Date
Mo Tu We Th Fr Sa Su
SSD 分类损失推导.
① logSoftmax 的定义式为:
$\log\left(\frac{e^{x_j}}{\sum_{i=1}^{n} e^{x_i}}\right) = \log(e^{x_j}) - \log\left(\sum_{i=1}^{n} e^{x_i}\right)$
$= x_j - \log\left(\sum_{i=1}^{n} e^{x_i}\right)$
② 为了防止数值溢出,可以把上面的公式转化如下:
定义 $\text{logsumexp}(x_1, \cdots x_n) = \log\left(\sum_{i=1}^{n} e^{x_i}\right)$
$= \log\left(\sum_{i=1}^{n} e^{x_i - c} e^{c}\right)$
$= \log\left(e^{c} \sum_{i=1}^{n} e^{x_i - c}\right)$
$= c + \log\left(\sum_{i=1}^{n} e^{x_i - c}\right)$.
③ 将②代入①得:
$\log(\text{Softmax}(x_j, x_1, \cdots x_n)) = x_j - \log\left(\sum_{i=1}^{n} e^{x_i - c}\right) - c$
④ 所以根据损失函数为 $\log(\text{Softmax}(x_j, x_1, \cdots x_n))$ 的相反数,
可得 $\text{loss\_cls} = \log\left(\sum_{i=1}^{n} e^{x_i - c}\right) + c - x_j$
$= \text{log\_sum\_exp}(x) - x_j$ → 这个就是 batch_conf.
这个就是 box_utils.py 实现的。
对应的代码实现为: conf_logP = log_sum_exp(batch_conf) -
batch_conf.gather(1, conf_t.view(-1, 1))
confidence in yourself is the first step on the road to success.
自信是走向成功的第一步。

损失函数完整代码实现，来自 layers/modules/multibox_loss.py：

```
class MultiBoxLoss(nn.Module):
    """SSD Weighted Loss Function
    Compute Targets:
        1) Produce Confidence Target Indices by matching  ground truth boxes
           with (default) 'priorboxes' that have jaccard index > threshold
 ↪  parameter
           (default threshold: 0.5).
        2) Produce localization target by 'encoding' variance into offsets
 ↪  of ground
           truth boxes and their matched  'priorboxes'.
        3) Hard negative mining to filter the excessive number of negative
 ↪  examples
           that comes with using a large number of default bounding boxes.
           (default negative:positive ratio 3:1)
    Objective Loss:
        L(x,c,l,g) = (Lconf(x, c) + αLloc(x,l,g)) / N
        Where, Lconf is the CrossEntropy Loss and Lloc is the SmoothL1 Loss
        weighted by α which is set to 1 by cross val.
        Args:
            c: class confidences,
            l: predicted boxes,
            g: ground truth boxes
            N: number of matched default boxes
        See: https://arxiv.org/pdf/1512.02325.pdf for more details.
    """

    def __init__(self, num_classes, overlap_thresh, prior_for_matching,
                 bkg_label, neg_mining, neg_pos, neg_overlap, encode_target,
                 use_gpu=True):
        super(MultiBoxLoss, self).__init__()
        self.use_gpu = use_gpu
        self.num_classes = num_classes
        self.threshold = overlap_thresh
        self.background_label = bkg_label
        self.encode_target = encode_target
        self.use_prior_for_matching = prior_for_matching
        self.do_neg_mining = neg_mining
        self.negpos_ratio = neg_pos
        self.neg_overlap = neg_overlap
        self.variance = cfg['variance']
```

```
    def forward(self, predictions, targets):
        """Multibox Loss
        Args:
            predictions (tuple): A tuple containing loc preds, conf preds,
            and prior boxes from SSD net.
                conf shape: torch.size(batch_size,num_priors,num_classes)
                loc shape: torch.size(batch_size,num_priors,4)
                priors shape: torch.size(num_priors,4)
            targets (tensor): Ground truth boxes and labels for a batch,
                shape: [batch_size,num_objs,5] (last idx is the label).
        """
        loc_data, conf_data, priors = predictions
        num = loc_data.size(0)# batch_size
        priors = priors[:loc_data.size(1), :]
        num_priors = (priors.size(0)) # 先验框个数
        num_classes = self.num_classes # 类别数

        # match priors (default boxes) and ground truth boxes
        # 获取匹配每个 prior box 的 ground truth
        # 创建 loc_t 和 conf_t 保存真实 box 的位置和类别
        loc_t = torch.Tensor(num, num_priors, 4)
        conf_t = torch.LongTensor(num, num_priors)
        for idx in range(num):
            truths = targets[idx][:, :-1].data #ground truth box 信息
            labels = targets[idx][:, -1].data # ground truth conf 信息
            defaults = priors.data # priors 的 box 信息
            # 匹配 ground truth
            match(self.threshold, truths, defaults, self.variance, labels,
                  loc_t, conf_t, idx)
        if self.use_gpu:
            loc_t = loc_t.cuda()
            conf_t = conf_t.cuda()
        # wrap targets
        loc_t = Variable(loc_t, requires_grad=False)
        conf_t = Variable(conf_t, requires_grad=False)
        # 匹配中所有的正样本 mask,shape[b,M]
        pos = conf_t > 0
        num_pos = pos.sum(dim=1, keepdim=True)
        # Localization Loss, 使用 Smooth L1
        # shape[b,M]-->shape[b,M,4]
        pos_idx = pos.unsqueeze(pos.dim()).expand_as(loc_data)
        loc_p = loc_data[pos_idx].view(-1, 4) # 预测的正样本 box 信息
        loc_t = loc_t[pos_idx].view(-1, 4) # 真实的正样本 box 信息
```

```
        loss_l = F.smooth_l1_loss(loc_p, loc_t, size_average=False) #Smooth
↪  L1 损失

        '''
        Target:
            下面进行 hard negative mining
        过程:
            1、 针对所有 batch 的 conf，按照置信度误差 (预测背景的置信度越小，误差越大)
↪  进行降序排列;
            2、 负样本的 label 全是背景，那么利用 log softmax 计算出 logP,
               logP 越大，则背景概率越低, 误差越大;
            3、 选取误差交大的 top_k 作为负样本，保证正负样本比例接近 1:3;
        '''
        # Compute max conf across batch for hard negative mining
        # shape[b*M,num_classes]
        batch_conf = conf_data.view(-1, self.num_classes)
        # 使用 logsoftmax，计算置信度,shape[b*M, 1]
        loss_c = log_sum_exp(batch_conf) - batch_conf.gather(1,
↪  conf_t.view(-1, 1))

        # Hard Negative Mining
        loss_c[pos] = 0  # 把正样本排除，剩下的就全是负样本，可以进行抽样
        loss_c = loss_c.view(num, -1)# shape[b, M]
        # 两次 sort 排序，能够得到每个元素在降序排列中的位置 idx_rank
        _, loss_idx = loss_c.sort(1, descending=True)
        _, idx_rank = loss_idx.sort(1)
         # 抽取负样本
        # 每个 batch 中正样本的数目，shape[b,1]
        num_pos = pos.long().sum(1, keepdim=True)
        num_neg = torch.clamp(self.negpos_ratio*num_pos, max=pos.size(1)-1)
        # 抽取前 top_k 个负样本，shape[b, M]
        neg = idx_rank < num_neg.expand_as(idx_rank)

        # Confidence Loss Including Positive and Negative Examples
        # shape[b,M] --> shape[b,M,num_classes]
        pos_idx = pos.unsqueeze(2).expand_as(conf_data)
        neg_idx = neg.unsqueeze(2).expand_as(conf_data)
        # 提取出所有筛选好的正负样本 (预测的和真实的)
        conf_p = conf_data[(pos_idx+neg_idx).gt(0)].view(-1,
↪  self.num_classes)
        targets_weighted = conf_t[(pos+neg).gt(0)]
        # 计算 conf 交叉熵
```

```
        loss_c = F.cross_entropy(conf_p, targets_weighted,
↪   size_average=False)

        # Sum of losses: L(x,c,l,g) = (Lconf(x, c) + αLloc(x,l,g)) / N
        # 正样本个数
        N = num_pos.data.sum()
        loss_l /= N
        loss_c /= N
        return loss_l, loss_c
```

5.4.2 先验框匹配策略

上面的代码中还有一个地方没讲到，就是 match 函数。这是 SSD 算法的先验框匹配函数。在训练时首先需要确定训练图片中的 ground truth 是由哪一个先验框来匹配，与之匹配的先验框所对应的边界框将负责预测它。SSD 的先验框和 ground truth 匹配原则主要有 2 点。第一点是对于图片中的每个 ground truth，找到和它 IOU 最大的先验框，该先验框与其匹配，这样可以保证每个 ground truth 一定与某个 prior 匹配。第二点是对于剩余的未匹配的先验框，若某个 ground truth 和它的 IOU 大于某个阈值 (一般设为 0.5)，那么改 prior 和这个 ground truth，剩下没有匹配上的先验框都是负样本（如果多个 ground truth 和某一个先验框的 IOU 均大于阈值，那么 prior 只与 IOU 最大的那个进行匹配）。代码实现如下，来自 `layers/box_utils.py`：

```
def match(threshold, truths, priors, variances, labels, loc_t, conf_t, idx):
    """ 把和每个 prior box 有最大的 IOU 的 ground truth box 进行匹配，
    同时，编码包围框，返回匹配的索引，对应的置信度和位置
    Args:
        threshold: IOU 阈值，小于阈值设为背景
        truths: ground truth boxes, shape[N,4]
        priors: 先验框， shape[M,4]
        variances: prior 的方差, list(float)
        labels: 图片的所有类别，shape[num_obj]
        loc_t: 用于填充 encoded loc 目标张量
        conf_t: 用于填充 encoded conf 目标张量
        idx: 现在的 batch index
        The matched indices corresponding to 1)location and 2)confidence
↪   preds.
    """
    # jaccard index
    # 计算 IOU
    overlaps = jaccard(
        truths,
        point_form(priors)
```

```
    )
    # (Bipartite Matching)
    # [1,num_objects] 和每个 ground truth box 交集最大的 prior box
    best_prior_overlap, best_prior_idx = overlaps.max(1, keepdim=True)
    # [1,num_priors] 和每个 prior box 交集最大的 ground truth box
    best_truth_overlap, best_truth_idx = overlaps.max(0, keepdim=True)
    best_truth_idx.squeeze_(0) #M
    best_truth_overlap.squeeze_(0) #M
    best_prior_idx.squeeze_(1) #N
    best_prior_overlap.squeeze_(1) #N
    # 保证每个 ground truth box 与某一个 prior box 匹配，固定值为 2 > threshold
   best_truth_overlap.index_fill_(0, best_prior_idx, 2)  # ensure best prior
    # TODO refactor: index  best_prior_idx with long tensor
    # ensure every gt matches with its prior of max overlap
    # 保证每一个 ground truth 匹配它的都是具有最大 IOU 的 prior
    # 根据 best_prior_dix 锁定 best_truth_idx 里面的最大 IOU prior
    for j in range(best_prior_idx.size(0)):
        best_truth_idx[best_prior_idx[j]] = j
    matches = truths[best_truth_idx]          # 提取出所有匹配的 ground truth
↪   box, Shape: [M,4]
    conf = labels[best_truth_idx] + 1         # 提取出所有 GT 框的类别,
↪   Shape:[M]
    # 把 iou < threshold 的框类别设置为 bg, 即为 0
    conf[best_truth_overlap < threshold] = 0  # label as background
    # 编码包围框
    loc = encode(matches, priors, variances)
    # 保存匹配好的 loc 和 conf 到 loc_t 和 conf_t 中
    loc_t[idx] = loc    # [num_priors,4] encoded offsets to learn
    conf_t[idx] = conf  # [num_priors] top class label for each prior
```

5.4.3 位置坐标转换

我们看到上面出现了一个 point_form 函数，这是什么意思呢？这是因为目标框有 2 种表示方式：

- $(x_{min}, y_{min}, x_{max}, y_{max})$
- (x, y, w, h) 这部分的代码在 layers/box_utils.py 下：

```
def point_form(boxes):
    """ Convert prior_boxes to (xmin, ymin, xmax, ymax)
   把 prior_box (cx, cy, w, h) 转化为 (xmin, ymin, xmax, ymax)
    """
    return torch.cat((boxes[:, :2] - boxes[:, 2:]/2,     # xmin, ymin
                      boxes[:, :2] + boxes[:, 2:]/2), 1)  # xmax, ymax
```

```
def center_size(boxes):
    """ Convert prior_boxes to (cx, cy, w, h)
    把 prior_box (xmin, ymin, xmax, ymax) 转化为 (cx, cy, w, h)
    """
    return torch.cat((boxes[:, 2:] + boxes[:, :2])/2,  # cx, cy
                     boxes[:, 2:] - boxes[:, :2], 1) # w, h
```

5.4.4 IOU 计算

这部分比较简单，对于两个 Box 来讲，首先计算两个 box 左上角点坐标的最大值和右下角坐标的最小值，然后计算交集面积，最后把交集面积除以对应的并集面积。代码仍在 layers/box_utils.py：

```
def intersect(box_a, box_b):
    """ We resize both tensors to [A,B,2] without new malloc:
    [A,2] -> [A,1,2] -> [A,B,2]
    [B,2] -> [1,B,2] -> [A,B,2]
    Then we compute the area of intersect between box_a and box_b.
    Args:
      box_a: (tensor) bounding boxes, Shape: [A,4].
      box_b: (tensor) bounding boxes, Shape: [B,4].
    Return:
      (tensor) intersection area, Shape: [A,B].
    """
    A = box_a.size(0)
    B = box_b.size(0)
     # 右下角，选出最小值
    max_xy = torch.min(box_a[:, 2:].unsqueeze(1).expand(A, B, 2),
                       box_b[:, 2:].unsqueeze(0).expand(A, B, 2))
    # 左上角，选出最大值
    min_xy = torch.max(box_a[:, :2].unsqueeze(1).expand(A, B, 2),
                       box_b[:, :2].unsqueeze(0).expand(A, B, 2))
    # 负数用 0 截断，为 0 代表交集为 0
    inter = torch.clamp((max_xy - min_xy), min=0)
    return inter[:, :, 0] * inter[:, :, 1]


def jaccard(box_a, box_b):
    """Compute the jaccard overlap of two sets of boxes.  The jaccard overlap
    is simply the intersection over union of two boxes.  Here we operate on
```

```
    ground truth boxes and default boxes.
    E.g.:
        A ∩ B / A ⊠ B = A ∩ B / (area(A) + area(B) - A ∩ B)
    Args:
        box_a: (tensor) Ground truth bounding boxes, Shape: [num_objects,4]
        box_b: (tensor) Prior boxes from priorbox layers, Shape:
↪   [num_priors,4]
    Return:
        jaccard overlap: (tensor) Shape: [box_a.size(0), box_b.size(0)]
    """
    inter = intersect(box_a, box_b)# A∩B
     # box_a 和 box_b 的面积
    area_a = ((box_a[:, 2]-box_a[:, 0]) *
              (box_a[:, 3]-box_a[:, 1])).unsqueeze(1).expand_as(inter)  #
↪   [A,B]#(N,)
    area_b = ((box_b[:, 2]-box_b[:, 0]) *
              (box_b[:, 3]-box_b[:, 1])).unsqueeze(0).expand_as(inter)  #
↪   [A,B]#(M,)
    union = area_a + area_b - inter
    return inter / union  # [A,B]
```

5.5 L2 标准化

VGG16 的 conv4_3 特征图的大小为 38×38，网络层靠前，方差比较大，需要加一个 L2 标准化，以保证和后面的检测层差异不是很大。L2 标准化的公式如下：$\hat{x} = \frac{x}{||x||^2}$，其中 $x = (x_1 ... x_d)||x||_2 = (\sum_{i=1}^{d} |x_i|^2)^{1/2}$。同时，这里还要注意的是如果简单的对一个 layer 的输入进行 L2 标准化就会改变该层的规模，并且会减慢学习速度，因此这里引入了一个缩放系数 γ_i，对于每一个通道 l2 标准化后的结果为：$y_i = \gamma_i \hat{x_i}$，通常 $scale$ 的值设 10 或者 20，效果比较好。代码来自 layers/modules/l2norm.py。

```
class L2Norm(nn.Module):
    '''
    conv4_3 特征图大小 38x38，网络层靠前，norm 较大，需要加一个 L2 Normalization, 以保
↪   证和后面的检测层差异不是很大，具体可以参考： ParseNet。这个前面的推文里面有讲。
    '''
    def __init__(self, n_channels, scale):
        super(L2Norm, self).__init__()
        self.n_channels = n_channels
        self.gamma = scale or None
        self.eps = 1e-10
        # 将一个不可训练的类型 Tensor 转换成可以训练的类型 parameter
        self.weight = nn.Parameter(torch.Tensor(self.n_channels))
        self.reset_parameters()
```

```
    # 初始化参数
    def reset_parameters(self):
        nn.init.constant_(self.weight, self.gamma)

    def forward(self, x):
        # 计算 x 的 2 范数
        norm = x.pow(2).sum(dim=1, keepdim=True).sqrt() # shape[b,1,38,38]
        x = x / norm   # shape[b,512,38,38]

        # 扩展 self.weight 的维度为 shape[1,512,1,1]，然后参考公式计算
        out = self.weight[None,...,None,None] * x
        return out
```

5.6 位置信息编解码

上面提到了计算坐标损失的时候，坐标是 encoding 之后的，这是怎么回事呢？根据论文的描述，预测框和 ground truth 边界框存在一个转换关系，先定义一些变量：

- 先验框位置：$d = (d^{cx}, d^{cy}, d^w, d^h)$
- ground truth 框位置：$g = (g^{cx}, g^{cy}, g^w, g^h)$
- variance 是先验框的坐标方差。

然后编码的过程可以表示为:$\hat{g_j^{cx}} = (g_j^{cx} - d_i^{cx})/d_i^w/varicance[0]$ $\hat{g_j^{cy}} = (g_j^{cy} - d_i^{cy})/d_i^h/varicance[1]$ $\hat{g_j^w} = log(\frac{g_j^w}{d_i^w})/variance[2]$ $\hat{g_j^h} = log(\frac{g_j^h}{d_i^h})/variance[3]$

解码的过程可以表示为：$g_{predict}^{cx} = d^w * (variance[0] * l^{cx}) + d^{cx}$ $g_{predict}^{cy} = d^h * (variance[1] * l^{cy}) + d^{cy}$ $g_{predict}^w = d^w exp(vairance[2] * l^w)$ $g_{predict}^h = d^h exp(vairance[3] * l^h)$

这部分对应的代码在 layers/box_utils.py 里面：

```
def encode(matched, priors, variances):
    """Encode the variances from the priorbox layers into the ground truth
     ↪ boxes
    we have matched (based on jaccard overlap) with the prior boxes.
    Args:
        matched: (tensor) Coords of ground truth for each prior in point-form
            Shape: [num_priors, 4].
        priors: (tensor) Prior boxes in center-offset form
            Shape: [num_priors,4].
        variances: (list[float]) Variances of priorboxes
    Return:
        encoded boxes (tensor), Shape: [num_priors, 4]
```

```
    """

    # dist b/t match center and prior's center
    g_cxcy = (matched[:, :2] + matched[:, 2:])/2 - priors[:, :2]
    # encode variance
    g_cxcy /= (variances[0] * priors[:, 2:])
    # match wh / prior wh
    g_wh = (matched[:, 2:] - matched[:, :2]) / priors[:, 2:]
    g_wh = torch.log(g_wh) / variances[1]
    # return target for smooth_l1_loss
    return torch.cat([g_cxcy, g_wh], 1)  # [num_priors,4]


# Adapted from https://github.com/Hakuyume/chainer-ssd
def decode(loc, priors, variances):
    """Decode locations from predictions using priors to undo
    the encoding we did for offset regression at train time.
    Args:
        loc (tensor): location predictions for loc layers,
            Shape: [num_priors,4]
        priors (tensor): Prior boxes in center-offset form.
            Shape: [num_priors,4].
        variances: (list[float]) Variances of priorboxes
    Return:
        decoded bounding box predictions
    """

    boxes = torch.cat((
        priors[:, :2] + loc[:, :2] * variances[0] * priors[:, 2:],
        priors[:, 2:] * torch.exp(loc[:, 2:] * variances[1])), 1)
    boxes[:, :2] -= boxes[:, 2:] / 2
    boxes[:, 2:] += boxes[:, :2]
return boxes
```

5.7 后处理 NMS

这部分我在推文讲过原理了，这里不再赘述了。这里 IOU 阈值取了 0.5。不了解原理可以去看一下我的那篇推文，也给了源码讲解，目标检测中的 NMS。这部分的代码也在 `layers/box_utils.py` 里面。这里就不再拿代码来赘述了。

6. 检测函数

模型在测试的时候，需要把 loc 和 conf 输入到 detect 函数进行 nms，然后给出结果。这部分的代码在 `layers/functions/detection.py` 里面，如下：

```
class Detect(Function):
    """At test time, Detect is the final layer of SSD.  Decode location
     ↪ preds,
    apply non-maximum suppression to location predictions based on conf
    scores and threshold to a top_k number of output predictions for both
    confidence score and locations.
    """
    def __init__(self, num_classes, bkg_label, top_k, conf_thresh,
     ↪ nms_thresh):
        self.num_classes = num_classes
        self.background_label = bkg_label
        self.top_k = top_k
        # Parameters used in nms.
        self.nms_thresh = nms_thresh
        if nms_thresh <= 0:
            raise ValueError('nms_threshold must be non negative.')
        self.conf_thresh = conf_thresh
        self.variance = cfg['variance']

    def forward(self, loc_data, conf_data, prior_data):
        """
        Args:
            loc_data: 预测出的 loc 张量，shape[b,M,4], eg:[b, 8732, 4]
            conf_data: 预测出的置信度，shape[b,M,num_classes], eg:[b, 8732, 21]
            prior_data: 先验框，shape[M,4], eg:[8732, 4]
        """
        num = loc_data.size(0)  # batch size
        num_priors = prior_data.size(0)
        output = torch.zeros(num, self.num_classes, self.top_k, 5)# 初始化输出
        conf_preds = conf_data.view(num, num_priors,
                                    self.num_classes).transpose(2, 1)

        # 解码 loc 的信息，变为正常的 bboxes
        for i in range(num):
            # 解码 loc
            decoded_boxes = decode(loc_data[i], prior_data, self.variance)
            # 拷贝每个 batch 内的 conf，用于 nms
            conf_scores = conf_preds[i].clone()
```

```
            # 遍历每一个类别
            for cl in range(1, self.num_classes):
                # 筛选掉 conf < conf_thresh 的 conf
                c_mask = conf_scores[cl].gt(self.conf_thresh)
                scores = conf_scores[cl][c_mask]
                # 如果都被筛掉了，则跳入下一类
                if scores.size(0) == 0:
                    continue
                # 筛选掉 conf < conf_thresh 的框
                l_mask = c_mask.unsqueeze(1).expand_as(decoded_boxes)
                boxes = decoded_boxes[l_mask].view(-1, 4)
                # idx of highest scoring and non-overlapping boxes per class
                # nms
                ids, count = nms(boxes, scores, self.nms_thresh, self.top_k)
                # nms 后得到的输出拼接
                output[i, cl, :count] = \
                    torch.cat((scores[ids[:count]].unsqueeze(1),
                               boxes[ids[:count]]), 1)
        flt = output.contiguous().view(num, -1, 5)
        _, idx = flt[:, :, 0].sort(1, descending=True)
        _, rank = idx.sort(1)
        flt[(rank < self.top_k).unsqueeze(-1).expand_as(flt)].fill_(0)
    return output
```

7. 后记

讲解到这里这个框架的核心代码也就讲解完了，训练起来一个模型是非常简单的，但是深入到代码理解以及做调优又是另外一回事了，希望这个代码解析电子书可以对你起到一点帮助作用。