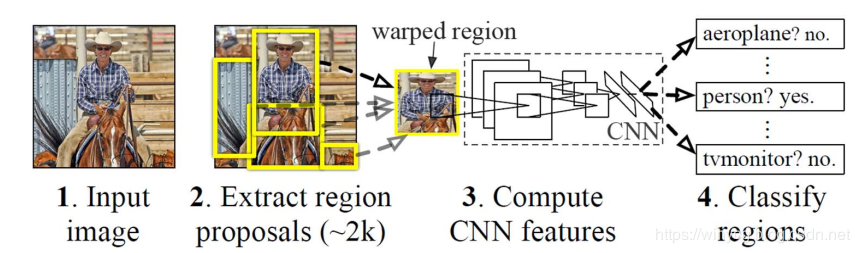
目标检测

目标检测算法一般分为两阶段检测和单阶段检测，常见的两阶段的检测方法有滑动窗口、R-CNN系列，SPPNET等，单阶段检测器有YOLO/SSD/Retinanet等

1.RCNN

因为我们把region proposal和CNNs结合起来，所以该方法被称为R-CNN：Regions with CNN features。这篇论文将CNN引入到目标检测中来，从下图的第三个步骤可以看出，他使用了CNN对大量的候选框的图片进行特征提取，之后用SVM分类器进行分类，RCNN的目标检测主要分为四个步骤：用下图可以大致表示：



首先输入一张图片，然后以selective search的方法生成候选区域，大概有两千个候选区域，生成候选区域是基于图像分割任务进行的，是无监督的提取。生成候选框的步骤：1、利用现有的分割方法将图像快速划分为许许多多的小区域 。2、然后基于相似度将相邻的相似度高的区域进行合并。3、不断合并相似的区域，直到整张图像成为一个区域。4在合并的过程中，基于所有产生的区域，都给出相应的矩形框。得到用于目标检测的候选窗口。当这些区域在分类器上呗分类的结果大于某个阈值的时候，相应的框会被标注出来。用下面这张图可以形象的说明到底是怎样生成这个候选区域的。

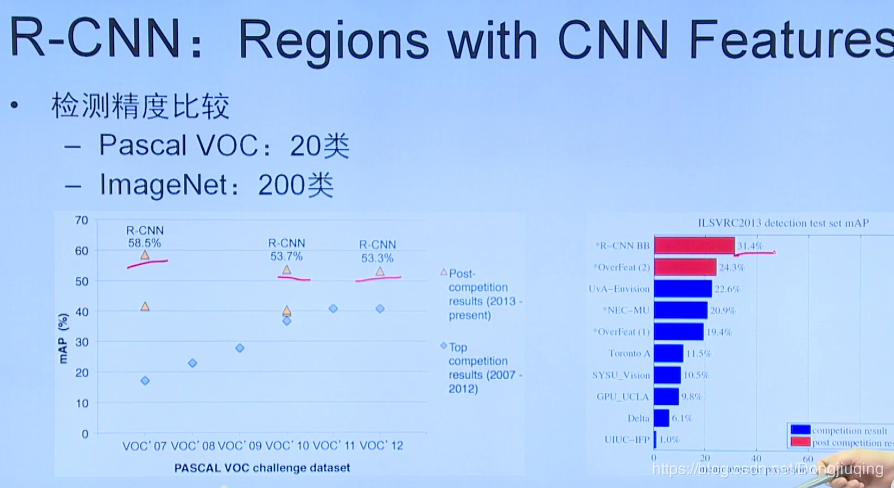
左边第一张最下面的图是最开始生成的很多很多个小块，然后基于相似度的一种算法，将相似的区域进行合并，直到整张图是一个大的区域，然后中间的图是说，在我生成这些区域的时候，同时画很多框，这些框就是我要检测的对象，在滑动窗口中，每一个窗口就是一个检测对象，同样，RCNN中的每一个候选区域，都要经过CNN进行特征提取，然后提取到底特征进一步分类。



之后的操作就是用CNN网络进行特征提取，那么对于这么多尺寸不一致的区域，该怎么处理呢？论文中给出了一种放缩的处理方式，就是不论你的候选框是大还是小，统统都放缩到227\*227，（当然论文中还提到了一些细节，比如放缩的时候，添加了一个16层的padding）然后送入CNN网络，以最后一个全连接层作为该区域的特征表示。



最后一步是对提取到的区域的特征表示，进行分类。通常使用SVM和softmax进行分类。论文中用了一种线性变换来对边框的位置进行校准。

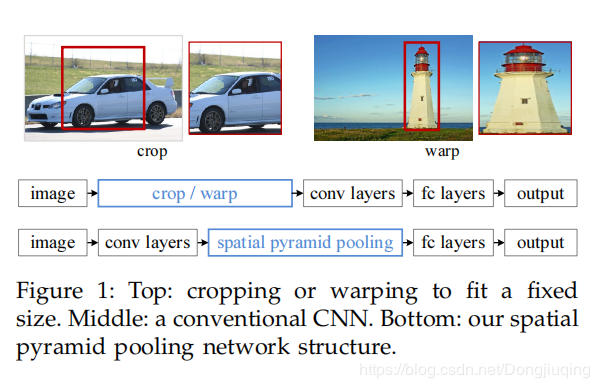


这张图显示了RCNN在两个数据集上的表现，可以看到RCNN明显优于传统算法，和别的深度学习方法。在目标检测的任务上是最好的，然而检测一张图需要花费大量的时间，需要47秒。

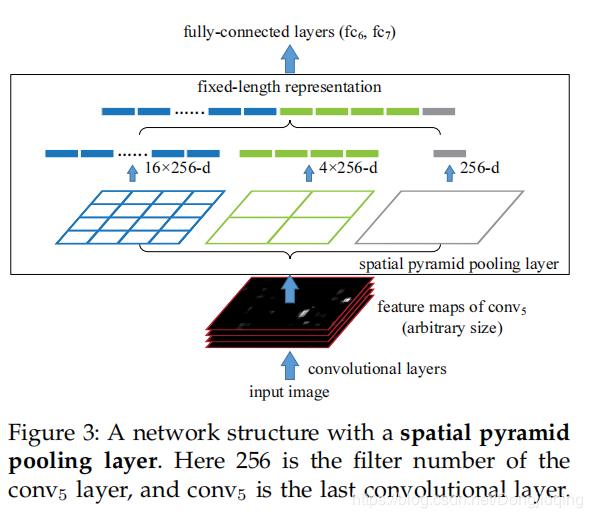
2. SPP-Net

SPP-Net（Spatial Pyramid Pooling）是何凯明2014年提出的方法，通过解决传统CNN无法处理不同尺寸输入的问题对RCNN做改进，实验结果表明SPP方法比R-CNN快了近100倍。在RCNN算法中，无论是对候选区域的放缩，还是将2000多张候选区域放入CNN进行特征提取，都需要耗费巨大的时间，这也是RCNN中主要的耗时。

RCNN的问题：1、对候选区域进行放缩或裁剪导致信息丢失或变形。既然会丢失信息，那么为什么要进行resize，调整到一个固定的尺寸呢？那是因为同样大小的图片经过卷积层之后，会产生同样大小的特征图或特征向量，然后送入全连接层，因为全连接层的输入输出大小是固定的，因此，送入卷积层的图片大小也必须是固定的。那么sppnet就想，主要的问题是由于全连接层的输入必须是固定的向量长度，而卷积层的输入是没有限制的，（因为一个大图片和一个小图片经过卷积层之后无非就是卷完的特征图一个大一个小而已）。那么在sppnet中，我对输入的图像没有尺寸限制，然后采用了一种空间金字塔池化的方式来产生特定大小的向量，然后送入全连接层。



空间金字塔池化方式：假设一个很简单的两层网络，池化的输出尺寸是未知的，全连接的输入要求为21



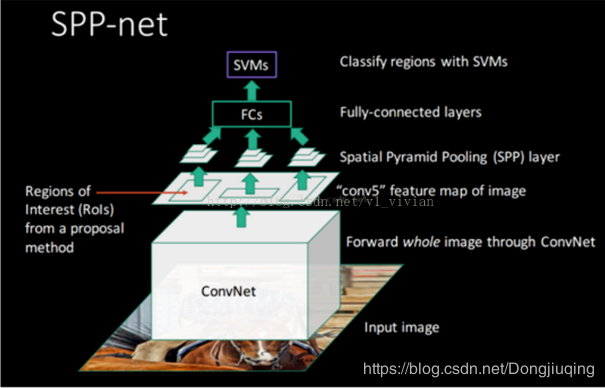
在这里SPP一共3层：第一层对特征图像做池化（最大、均值等）；第二层将特征图像划分为2\*2=4个块，分别做池化；第三层划分为4\*4=16个块，分别做池化

通过以上步骤，最后获得1+4+16=21个特征值，（最后还要乘个通道数）

通过这样的方式，CNN网络可以接受任意尺寸的特征图，而不用考虑最后生成的特征向量的长度。

SPPnet还有一处改进，大大减少了网络耗时。RCNN中，首先生成2k个候选框，然后将2k个框送入CNN中，那么这样以来，一张图片我经历了2K次CNN计算特征，这太耗时了，而且一张图就这么大，计算了两千次，那么也就是说有很多特征都是重复计算的。SPPnet中我首先将一整张图送入网络，然后计算所有的特征，同样的我也有区域候选的步骤，选完了之后，我直接将这些区域通过坐标转换的方式，直接对应到原图的特征图上，也就是说在特征图上找到这些候选区域，然后送进SPPnet进行分类，相当于我跳过了把候选框送入CNN进行特征提取的步骤，通过这样一个技巧，速度提升了一百倍。

最后用一张图来表示一下SPPnet：



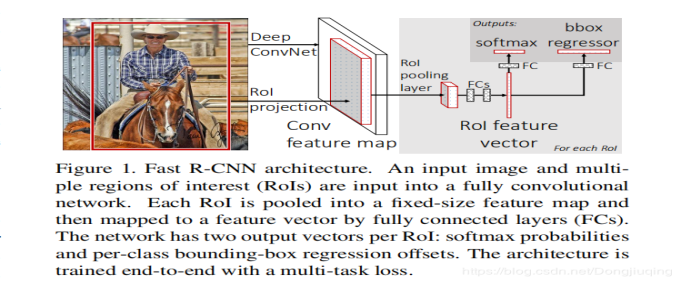
3 Fast RCNN

Fast RCNN保持了sppnet的优势，并进一步简化sppnet为单尺度，文中称作ROI polling，这个操作和空间池化是一样的，只是他只有一个划分模板，具体的关于ROI pooling可以参考这个网址：https://deepsense.ai/region-of-interest-pooling-explained/

改进：引入光滑L1 损失函数：smooth L1 LOSS

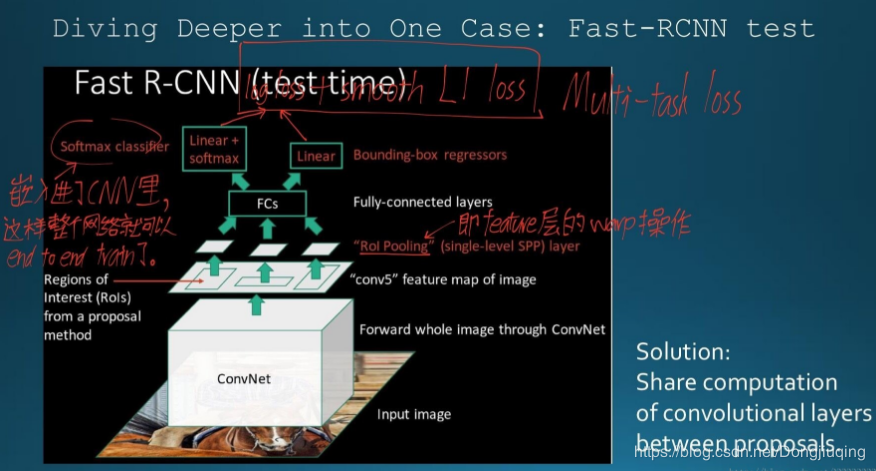
之前的损失函数通常用的是L2损失函数，而L2损失函数，通常类似二次曲线，在大的地方梯度大，而上面这条曲线则是在绝对值小于1的时候，是二次曲线，大于1的时候是一条直线，所以对于光滑L1损失函数，可以防止在特征较大的时候，梯度突然变得较大。因此具有更好的稳定性。

全连接层加速：将全连接层的权重矩阵分解为两个简单的矩阵，降低复杂度。加快计算速度。讲一个大的全连接层分解为两个小的全连接层。



a)与SPP类似，它只对整幅图像做一次CNN特征提取，在后面加了一个类似于SPP的ROI pooling layer，其实就是下采样。不过因为不是固定尺寸输入，因此每次的pooling网格大小需要动态调整。

b) 整个的训练过程是端到端的(除去region proposal提取阶段)，梯度能够通过RoI Pooling层直接传播，直接使用softmax替代SVM分类，同时利用Multi-task Loss（多任务损失函数）将边框回归和分类一起进行。



这是论文中使用的损失函数，p和u分别表示类别的概率与真实的样本，t和v表示坐标预测与真实框。

1.multi-loss traing相比单独训练classification确有提升

2.multi-scale相比single-scale精度略有提升，但带来的时间开销更大。一定程度上说明CNN结构可以内在地学习尺度不变性

3.在更多的数据(VOC)上训练后，精度是有进一步提升的

4.Softmax分类器比"one vs rest"型的SVM表现略好，引入了类间的竞争

5.更多的Proposal并不一定带来精度的提升，最开始产生了两千个框，而我通过计算IOU，排除掉大量的阈值低于某个值的框，只计算剩下的300个。大大减少了计算量

这里再说一下Fast R-CNN的主要步骤：

a) 特征提取：以整张图片为输入利用CNN得到图片的特征层；

b) region proposal：通过Selective Search等方法从原始图片提取区域候选框，并把这些候选框一一投影到最后的特征层；

c) 区域归一化：针对特征层上的每个区域候选框进行RoI Pooling操作，得到固定大小的特征表示；

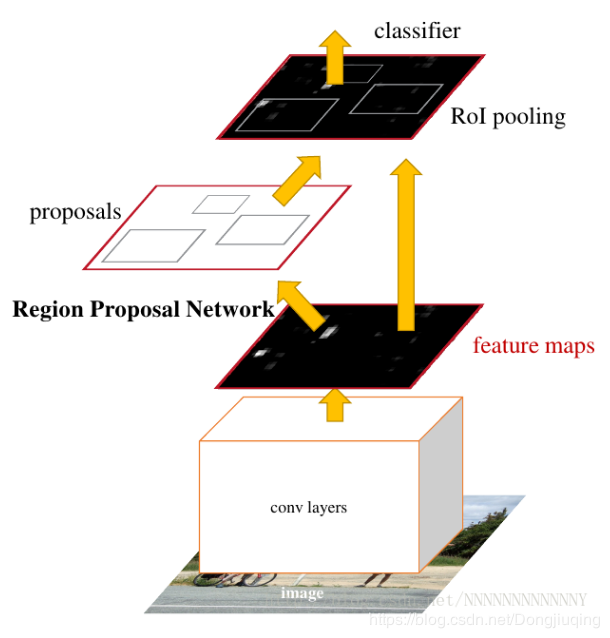
d) 分类与回归：然后再通过两个全连接层，分别用softmax做多目标分类，用回归模型进行边框位置与大小微调。

总结：Fast R-CNN确实做得很棒，其缺点在于：region proposal的提取使用selective search，目标检测时间大多消耗在这上面（提region proposal 2~3s，而提特征分类只需0.32s），无法满足实时应用。

4.Faster RCNN

上面最后一句话说出了Fast RCNN还有缺点，就是这个region proposal使用的还是selective search，而文章开头说，这个是基于图像分割任务做的，说白了还是含有传统算法的内容，于是FasterRCNN在这方面做了一些改进，使用RPN（region proposal network）来产生候选区域。提出的RPN网络取代Selective Search算法使得检测任务可以由神经网络端到端地完成。这也是FasterRCNN的核心贡献。

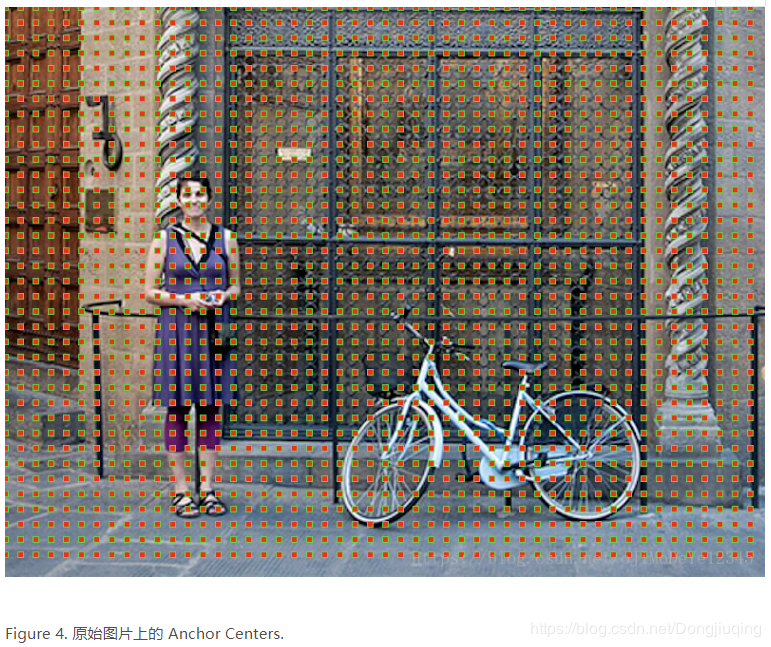
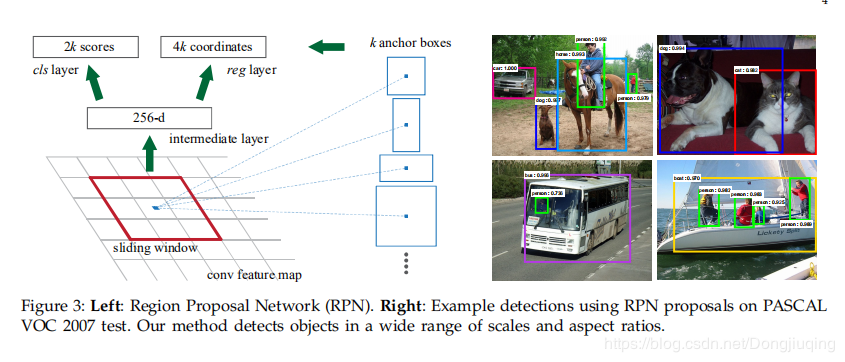
让生成候选窗口的CNN和分类的CNN 共享参数。

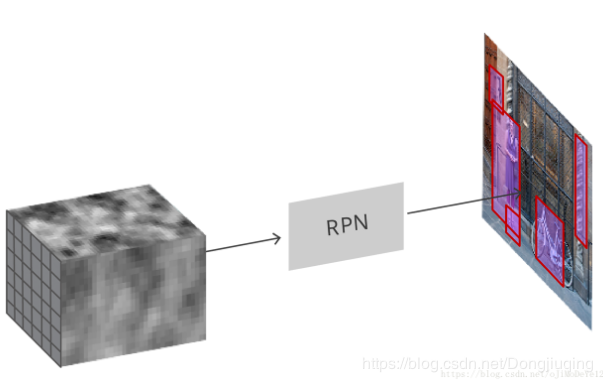


首先输入一张图片，然后通过卷积层得到最后一张特征图，用一个滑动窗口来遍历特征图上的每一个点，来生成k个候选窗口，称作anchor box。为了尽量代表所有物体，所以这些anchor box长宽比不同，尺寸大小不同。

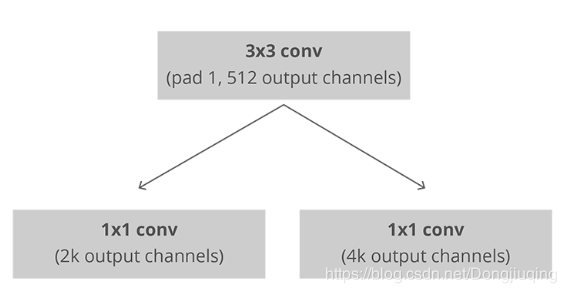
因为是对提取的卷积特征图进行处理，因此，在特征图上的每个点创建 anchors. 需要理解的是，虽然 anchors 是基于卷积特征图定义的，但最终的 anchors 是相对于原始图片的。（最后显示出来的是在原图片上的一个框，而不是框在特征图上）

由于只有卷积层和 pooling 层，特征图的维度是与原始图片的尺寸成比例关系的. 即，数学地表述，如果图片尺寸 w×h，特征图的尺寸则是 w/r×h/r. 其中，r 是下采样率(subsampling ratio)你可以理解为缩放率。 如果在卷积特征图空间位置定义 anchor，则最终的图片会是由 r 像素划分的 anchors 集. 在 VGG 中，r=16.





RPN不关注每个位置到底是什么类别，只关注这个点预测的框是物体还是背景。RPN 是全卷积(full conv) 网络，其采用基础网络输出的卷积特征图作为输入. 首先，采用 512 channel，3×3 kernel 的卷积层，然后是两个并行的 1×1 kernel 的卷积层，该卷积层的 channels 数量取决每个点的 anchors 的数量.



k是anchor的数量。

对于分类层，每个 anchor 输出两个预测值：anchor 是背景的 score 和 object 的 score.

对于回归层，也可以叫边界框调整层，每个 anchor 输出 4 个预测值：Δxcenter、Δycenter、Δwidth、Δheight即用于 anchors 来得到最终的 proposals.

根据最终的 proposal 坐标和其对应的 objectness score，即可得到良好的 objects proposals.

Faster R-CNN的主要步骤如下：

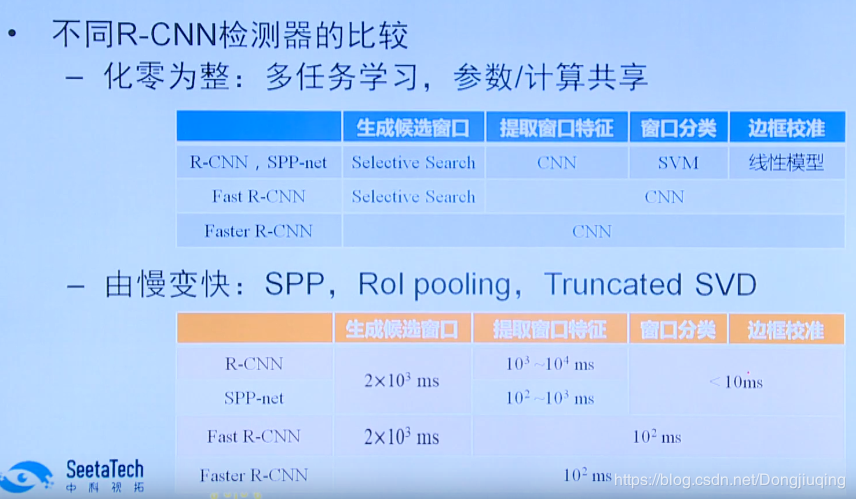
a) 特征提取：同Fast R-CNN；

b) region proposal：在最终的卷积特征层上利用k个不同的矩形框（Anchor Box）进行region proposal；即对每个Anchor Box对应的区域进行object/non-object二分类，并用k个回归模型（各自对应不同的Anchor Box）微调候选框位置与大小

c) 区域归一化：同fast R-CNN；

d) 分类与回归：进行目标分类，并做边框回归（感觉这一块再做一次边框回归是不是有点重复）。

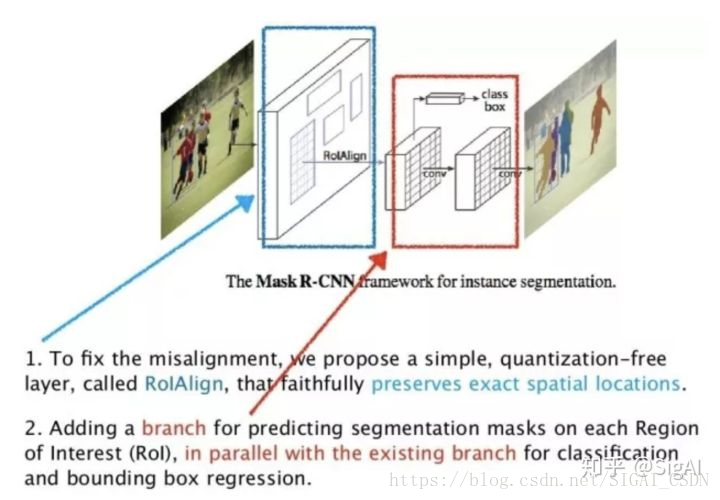
Faster R-CNN的训练：为了让RPN的网络和Fast R-CNN网络实现卷积层的权值共享，其训练方法比较复杂。



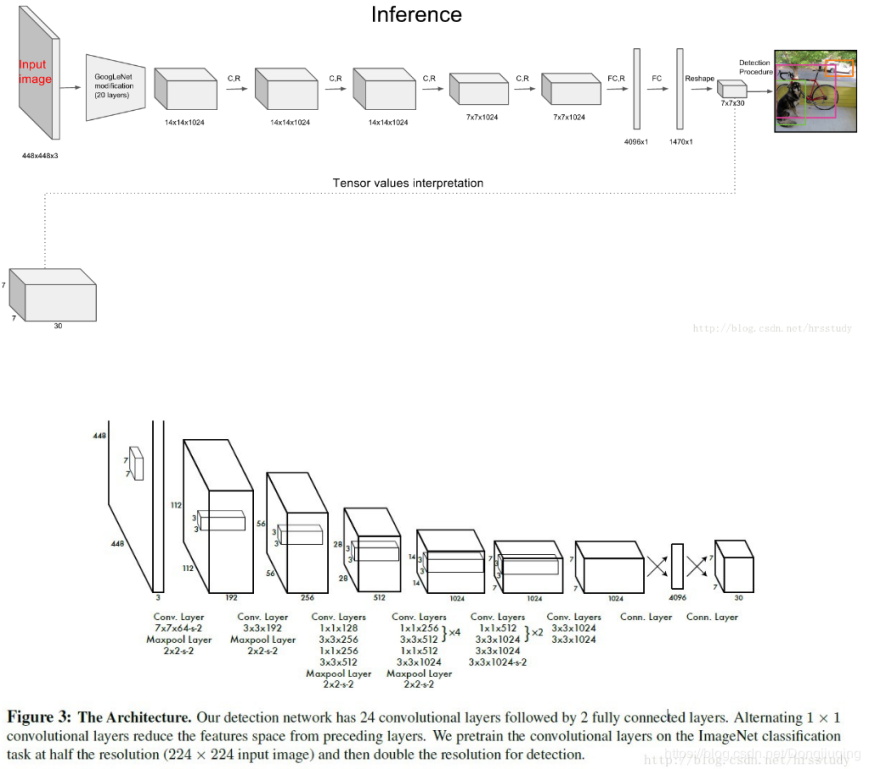
集中单阶段的检测器

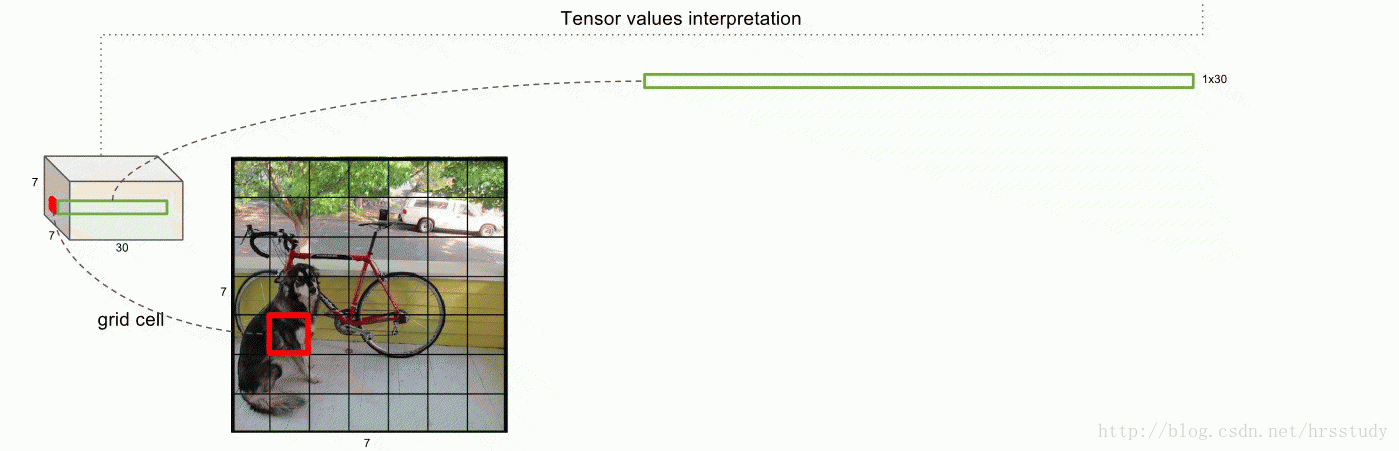
Mask R-CNN

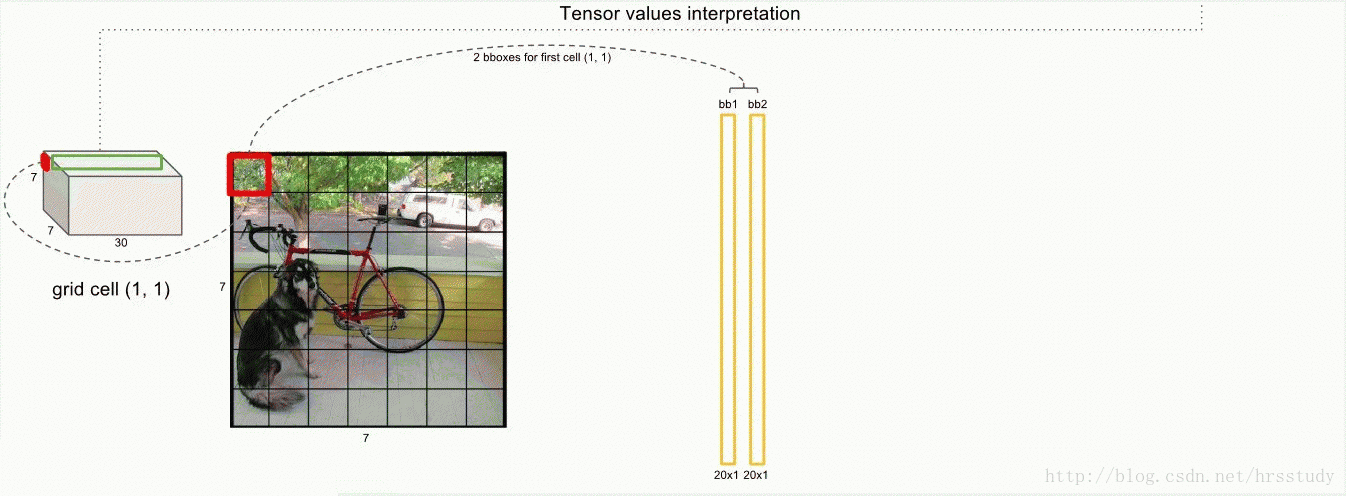
2017年Kaiming He等提出了Mask R-CNN ，并获得ICCV2017 Best Paper Award。作者指出，Faster R-CNN在做下采样和RoI Pooling时都对特征图大小做了取整操作，这种做法对于分类任务基本没有影响，但对检测任务会有一定影响，对语义分割这种像素级任务的精度影响则更为严重。为此，作者对网络中涉及特征图尺寸变化的环节都不使用取整操作，而是通过双线性差值填补非整数位置的像素。这使得下游特征图向上游映射时没有位置误差，不仅提升了目标检测效果，还使得算法能满足语义分割任务的精度要求。

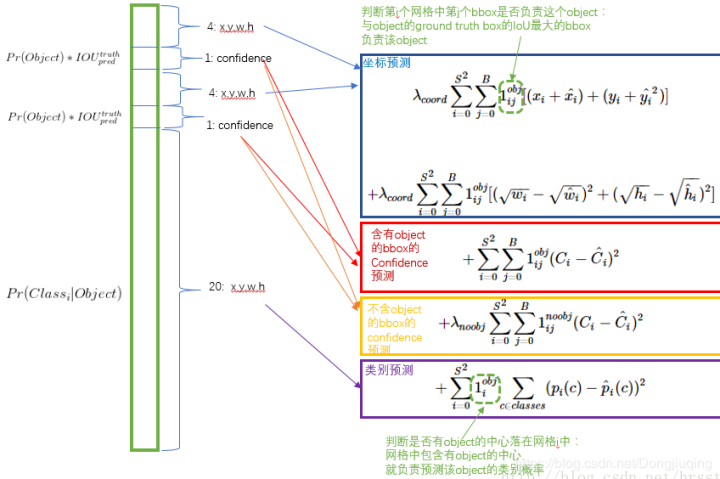


YOLO-1







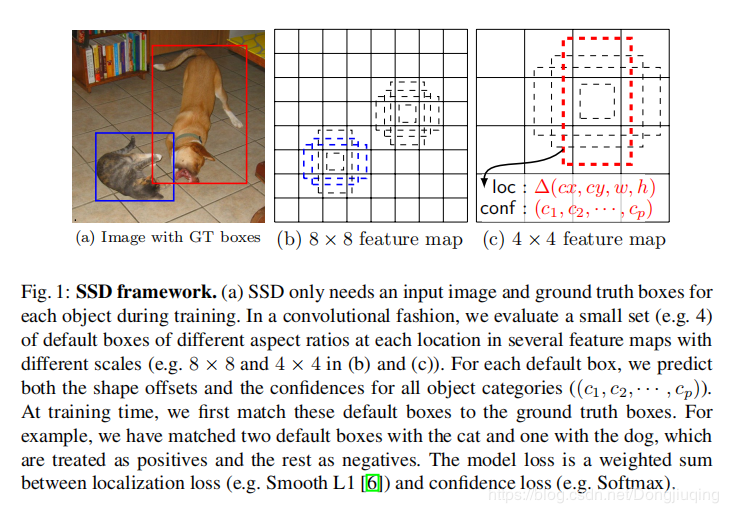


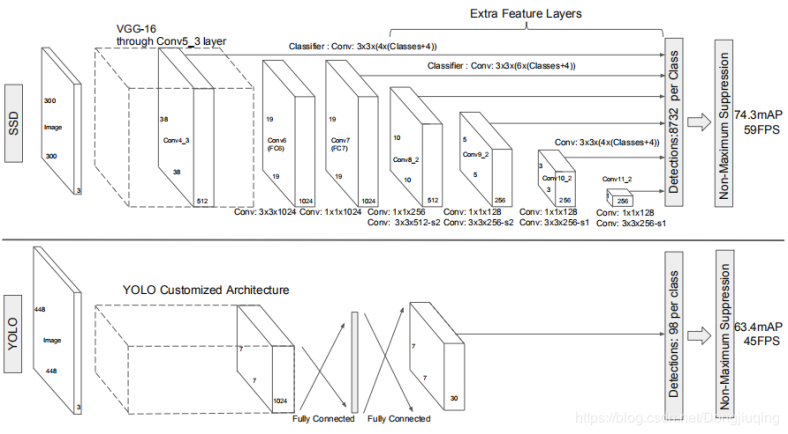
SSD

SSD对YOLO进行了改进，达到了和两阶段方法相当的精度，同时又保持了较快的运行速度。SSD也采用了网格划分的思想，和Faster RCNN不同的是它将所有的操作整合在一个卷积网络中完成。为了检测不同尺度的目标，SSD对不同卷积层的特征图像进行滑窗扫描；在前面的卷积层输出的特征图像中检测小的目标，在后面的卷积层输出的特征图像中检测大的目标。它的主要特点是：

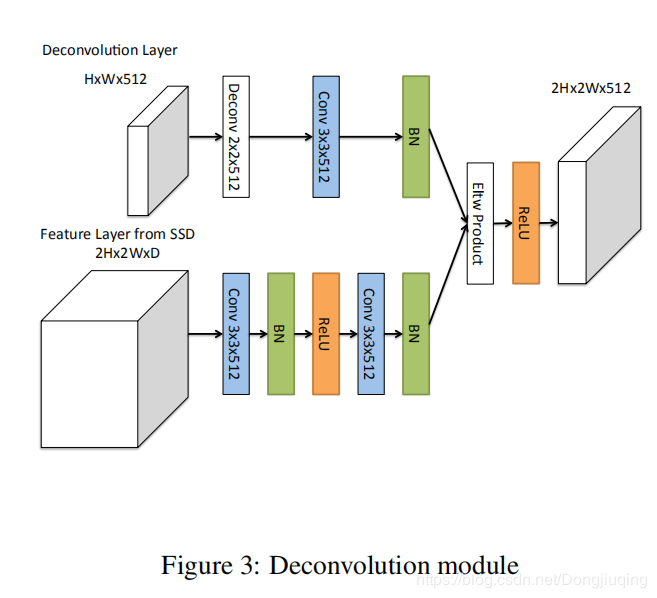
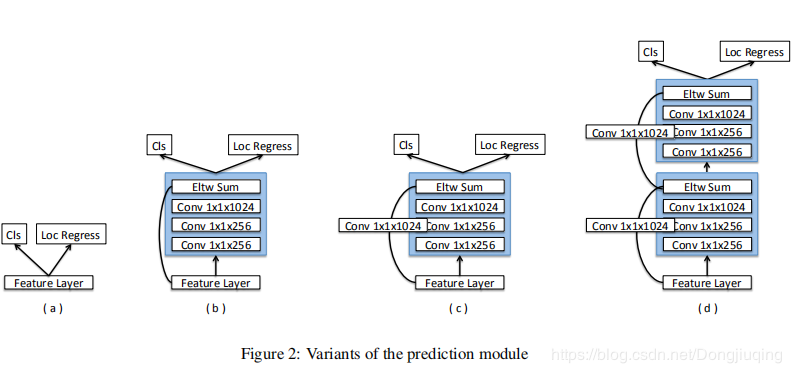
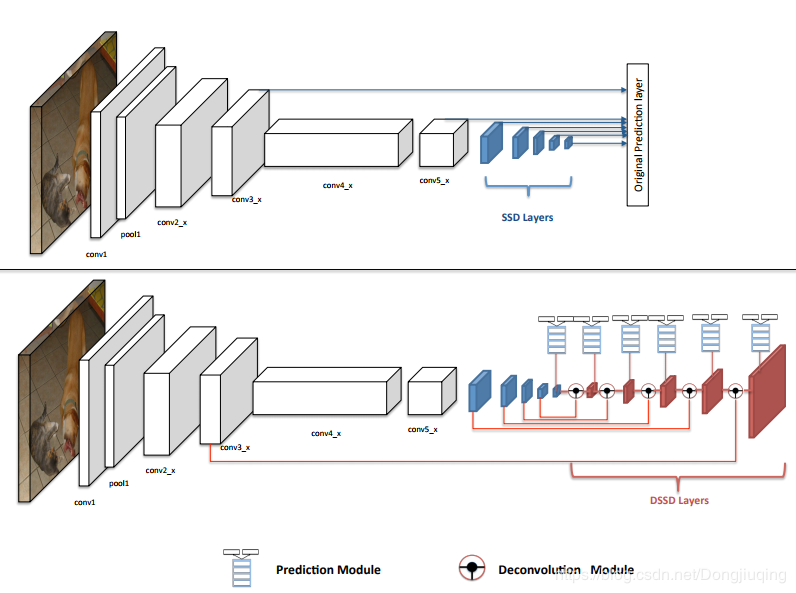
1.基于多尺度特征图像的检测：在多个尺度的卷积特征图上进行预测，以检测不同大小的目标，一定程度上提升了小目标物体的检测精度。

2.借鉴了Faster R-CNN中的Anchor boxes思想，在不同尺度的特征图上采样候选区域，一定程度上提升了检测的召回率以及小目标的检测效果。下图是SSD的原理：





DSSD:Deconvolutional Single-short Detector



ceiling函数返回大于或等于所给数字表达式的最小整数。  
floor函数返回小于或等于所给数字表达式的最大整数。  
比如  
celling(12.1) 结果为 13  
floor(12.1)结果为 12

|  |
| --- |
| from torch import nn |
|  | from utils import \* |
|  | import torch.nn.functional as F |
|  | from math import sqrt |
|  | from itertools import product as product |
|  | import torchvision |
|  |  |
|  | device = torch.device("cuda" if torch.cuda.is\_available() else "cpu") |
|  |  |
|  |  |
|  | class VGGBase(nn.Module): |
|  | """ |
|  | VGG base convolutions to produce lower-level feature maps. |
|  | """ |
|  |  |
|  | def \_\_init\_\_(self): |
|  | super(VGGBase, self).\_\_init\_\_() |
|  |  |
|  | # Standard convolutional layers in VGG16 |
|  | self.conv1\_1 = nn.Conv2d(3, 64, kernel\_size=3, padding=1) # stride = 1, by default |
|  | self.conv1\_2 = nn.Conv2d(64, 64, kernel\_size=3, padding=1) |
|  | self.pool1 = nn.MaxPool2d(kernel\_size=2, stride=2) |
|  |  |
|  | self.conv2\_1 = nn.Conv2d(64, 128, kernel\_size=3, padding=1) |
|  | self.conv2\_2 = nn.Conv2d(128, 128, kernel\_size=3, padding=1) |
|  | self.pool2 = nn.MaxPool2d(kernel\_size=2, stride=2) |
|  |  |
|  | self.conv3\_1 = nn.Conv2d(128, 256, kernel\_size=3, padding=1) |
|  | self.conv3\_2 = nn.Conv2d(256, 256, kernel\_size=3, padding=1) |
|  | self.conv3\_3 = nn.Conv2d(256, 256, kernel\_size=3, padding=1) |
|  | self.pool3 = nn.MaxPool2d(kernel\_size=2, stride=2, ceil\_mode=True) # ceiling (not floor) here for even dims |
|  |  |
|  | self.conv4\_1 = nn.Conv2d(256, 512, kernel\_size=3, padding=1) |
|  | self.conv4\_2 = nn.Conv2d(512, 512, kernel\_size=3, padding=1) |
|  | self.conv4\_3 = nn.Conv2d(512, 512, kernel\_size=3, padding=1) |
|  | self.pool4 = nn.MaxPool2d(kernel\_size=2, stride=2) |
|  |  |
|  | self.conv5\_1 = nn.Conv2d(512, 512, kernel\_size=3, padding=1) |
|  | self.conv5\_2 = nn.Conv2d(512, 512, kernel\_size=3, padding=1) |
|  | self.conv5\_3 = nn.Conv2d(512, 512, kernel\_size=3, padding=1) |
|  | self.pool5 = nn.MaxPool2d(kernel\_size=3, stride=1, padding=1) # retains size because stride is 1 (and padding) |
|  |  |
|  | # Replacements for FC6 and FC7 in VGG16 |
|  | self.conv6 = nn.Conv2d(512, 1024, kernel\_size=3, padding=6, dilation=6) # atrous convolution |
|  |  |
|  | self.conv7 = nn.Conv2d(1024, 1024, kernel\_size=1) |
|  |  |
|  | # Load pretrained layers |
|  | self.load\_pretrained\_layers() |
|  |  |
|  | def forward(self, image): |
|  | """ |
|  | Forward propagation. |
|  |  |
|  | :param image: images, a tensor of dimensions (N, 3, 300, 300) |
|  | :return: lower-level feature maps conv4\_3 and conv7 |
|  | """ |
|  | out = F.relu(self.conv1\_1(image)) # (N, 64, 300, 300) |
|  | out = F.relu(self.conv1\_2(out)) # (N, 64, 300, 300) |
|  | out = self.pool1(out) # (N, 64, 150, 150) |
|  |  |
|  | out = F.relu(self.conv2\_1(out)) # (N, 128, 150, 150) |
|  | out = F.relu(self.conv2\_2(out)) # (N, 128, 150, 150) |
|  | out = self.pool2(out) # (N, 128, 75, 75) |
|  |  |
|  | out = F.relu(self.conv3\_1(out)) # (N, 256, 75, 75) |
|  | out = F.relu(self.conv3\_2(out)) # (N, 256, 75, 75) |
|  | out = F.relu(self.conv3\_3(out)) # (N, 256, 75, 75) |
|  | out = self.pool3(out) # (N, 256, 38, 38), it would have been 37 if not for ceil\_mode = True |
|  |  |
|  | out = F.relu(self.conv4\_1(out)) # (N, 512, 38, 38) |
|  | out = F.relu(self.conv4\_2(out)) # (N, 512, 38, 38) |
|  | out = F.relu(self.conv4\_3(out)) # (N, 512, 38, 38) |
|  | conv4\_3\_feats = out # (N, 512, 38, 38) |
|  | out = self.pool4(out) # (N, 512, 19, 19) |
|  |  |
|  | out = F.relu(self.conv5\_1(out)) # (N, 512, 19, 19) |
|  | out = F.relu(self.conv5\_2(out)) # (N, 512, 19, 19) |
|  | out = F.relu(self.conv5\_3(out)) # (N, 512, 19, 19) |
|  | out = self.pool5(out) # (N, 512, 19, 19), pool5 does not reduce dimensions |
|  |  |
|  | out = F.relu(self.conv6(out)) # (N, 1024, 19, 19) |
|  |  |
|  | conv7\_feats = F.relu(self.conv7(out)) # (N, 1024, 19, 19) |
|  |  |
|  | # Lower-level feature maps |
|  | return conv4\_3\_feats, conv7\_feats |
|  |  |
|  | def load\_pretrained\_layers(self): |
|  | """ |
|  | As in the paper, we use a VGG-16 pretrained on the ImageNet task as the base network. |
|  | There's one available in PyTorch, see https://pytorch.org/docs/stable/torchvision/models.html#torchvision.models.vgg16 |
|  | We copy these parameters into our network. It's straightforward for conv1 to conv5. |
|  | However, the original VGG-16 does not contain the conv6 and con7 layers. |
|  | Therefore, we convert fc6 and fc7 into convolutional layers, and subsample by decimation. See 'decimate' in utils.py. |
|  | """ |
|  | # Current state of base |
|  | state\_dict = self.state\_dict() |
|  | param\_names = list(state\_dict.keys()) |
|  |  |
|  | # Pretrained VGG base |
|  | pretrained\_state\_dict = torchvision.models.vgg16(pretrained=True).state\_dict() |
|  | pretrained\_param\_names = list(pretrained\_state\_dict.keys()) |
|  |  |
|  | # Transfer conv. parameters from pretrained model to current model |
|  | for i, param in enumerate(param\_names[:-4]): # excluding conv6 and conv7 parameters |
|  | state\_dict[param] = pretrained\_state\_dict[pretrained\_param\_names[i]] |
|  |  |
|  | # Convert fc6, fc7 to convolutional layers, and subsample (by decimation) to sizes of conv6 and conv7 |
|  | # fc6 |
|  | conv\_fc6\_weight = pretrained\_state\_dict['classifier.0.weight'].view(4096, 512, 7, 7) # (4096, 512, 7, 7) |
|  | conv\_fc6\_bias = pretrained\_state\_dict['classifier.0.bias'] # (4096) |
|  | state\_dict['conv6.weight'] = decimate(conv\_fc6\_weight, m=[4, None, 3, 3]) # (1024, 512, 3, 3) |
|  | state\_dict['conv6.bias'] = decimate(conv\_fc6\_bias, m=[4]) # (1024) |
|  | # fc7 |
|  | conv\_fc7\_weight = pretrained\_state\_dict['classifier.3.weight'].view(4096, 4096, 1, 1) # (4096, 4096, 1, 1) |
|  | conv\_fc7\_bias = pretrained\_state\_dict['classifier.3.bias'] # (4096) |
|  | state\_dict['conv7.weight'] = decimate(conv\_fc7\_weight, m=[4, 4, None, None]) # (1024, 1024, 1, 1) |
|  | state\_dict['conv7.bias'] = decimate(conv\_fc7\_bias, m=[4]) # (1024) |
|  |  |
|  | # Note: an FC layer of size (K) operating on a flattened version (C\*H\*W) of a 2D image of size (C, H, W)... |
|  | # ...is equivalent to a convolutional layer with kernel size (H, W), input channels C, output channels K... |
|  | # ...operating on the 2D image of size (C, H, W) without padding |
|  |  |
|  | self.load\_state\_dict(state\_dict) |
|  |  |
|  | print("\nLoaded base model.\n") |
|  |  |
|  |  |
|  | class AuxiliaryConvolutions(nn.Module): |
|  | """ |
|  | Additional convolutions to produce higher-level feature maps. |
|  | """ |
|  |  |
|  | def \_\_init\_\_(self): |
|  | super(AuxiliaryConvolutions, self).\_\_init\_\_() |
|  |  |
|  | # Auxiliary/additional convolutions on top of the VGG base |
|  | self.conv8\_1 = nn.Conv2d(1024, 256, kernel\_size=1, padding=0) # stride = 1, by default |
|  | self.conv8\_2 = nn.Conv2d(256, 512, kernel\_size=3, stride=2, padding=1) # dim. reduction because stride > 1 |
|  |  |
|  | self.conv9\_1 = nn.Conv2d(512, 128, kernel\_size=1, padding=0) |
|  | self.conv9\_2 = nn.Conv2d(128, 256, kernel\_size=3, stride=2, padding=1) # dim. reduction because stride > 1 |
|  |  |
|  | self.conv10\_1 = nn.Conv2d(256, 128, kernel\_size=1, padding=0) |
|  | self.conv10\_2 = nn.Conv2d(128, 256, kernel\_size=3, padding=0) # dim. reduction because padding = 0 |
|  |  |
|  | self.conv11\_1 = nn.Conv2d(256, 128, kernel\_size=1, padding=0) |
|  | self.conv11\_2 = nn.Conv2d(128, 256, kernel\_size=3, padding=0) # dim. reduction because padding = 0 |
|  |  |
|  | # Initialize convolutions' parameters |
|  | self.init\_conv2d() |
|  |  |
|  | def init\_conv2d(self): |
|  | """ |
|  | Initialize convolution parameters. |
|  | """ |
|  | for c in self.children(): |
|  | if isinstance(c, nn.Conv2d): |
|  | nn.init.xavier\_uniform\_(c.weight) |
|  | nn.init.constant\_(c.bias, 0.) |
|  |  |
|  | def forward(self, conv7\_feats): |
|  | """ |
|  | Forward propagation. |
|  |  |
|  | :param conv7\_feats: lower-level conv7 feature map, a tensor of dimensions (N, 1024, 19, 19) |
|  | :return: higher-level feature maps conv8\_2, conv9\_2, conv10\_2, and conv11\_2 |
|  | """ |
|  | out = F.relu(self.conv8\_1(conv7\_feats)) # (N, 256, 19, 19) |
|  | out = F.relu(self.conv8\_2(out)) # (N, 512, 10, 10) |
|  | conv8\_2\_feats = out # (N, 512, 10, 10) |
|  |  |
|  | out = F.relu(self.conv9\_1(out)) # (N, 128, 10, 10) |
|  | out = F.relu(self.conv9\_2(out)) # (N, 256, 5, 5) |
|  | conv9\_2\_feats = out # (N, 256, 5, 5) |
|  |  |
|  | out = F.relu(self.conv10\_1(out)) # (N, 128, 5, 5) |
|  | out = F.relu(self.conv10\_2(out)) # (N, 256, 3, 3) |
|  | conv10\_2\_feats = out # (N, 256, 3, 3) |
|  |  |
|  | out = F.relu(self.conv11\_1(out)) # (N, 128, 3, 3) |
|  | conv11\_2\_feats = F.relu(self.conv11\_2(out)) # (N, 256, 1, 1) |
|  |  |
|  | # Higher-level feature maps |
|  | return conv8\_2\_feats, conv9\_2\_feats, conv10\_2\_feats, conv11\_2\_feats |
|  |  |
|  |  |
|  | class PredictionConvolutions(nn.Module): |
|  | """ |
|  | Convolutions to predict class scores and bounding boxes using lower and higher-level feature maps. |
|  |  |
|  | The bounding boxes (locations) are predicted as encoded offsets w.r.t each of the 8732 prior (default) boxes. |
|  | See 'cxcy\_to\_gcxgcy' in utils.py for the encoding definition. |
|  |  |
|  | The class scores represent the scores of each object class in each of the 8732 bounding boxes located. |
|  | A high score for 'background' = no object. |
|  | """ |
|  |  |
|  | def \_\_init\_\_(self, n\_classes): |
|  | """ |
|  | :param n\_classes: number of different types of objects |
|  | """ |
|  | super(PredictionConvolutions, self).\_\_init\_\_() |
|  |  |
|  | self.n\_classes = n\_classes |
|  |  |
|  | # Number of prior-boxes we are considering per position in each feature map |
|  | n\_boxes = {'conv4\_3': 4, |
|  | 'conv7': 6, |
|  | 'conv8\_2': 6, |
|  | 'conv9\_2': 6, |
|  | 'conv10\_2': 4, |
|  | 'conv11\_2': 4} |
|  | # 4 prior-boxes implies we use 4 different aspect ratios, etc. |
|  |  |
|  | # Localization prediction convolutions (predict offsets w.r.t prior-boxes) |
|  | self.loc\_conv4\_3 = nn.Conv2d(512, n\_boxes['conv4\_3'] \* 4, kernel\_size=3, padding=1) |
|  | self.loc\_conv7 = nn.Conv2d(1024, n\_boxes['conv7'] \* 4, kernel\_size=3, padding=1) |
|  | self.loc\_conv8\_2 = nn.Conv2d(512, n\_boxes['conv8\_2'] \* 4, kernel\_size=3, padding=1) |
|  | self.loc\_conv9\_2 = nn.Conv2d(256, n\_boxes['conv9\_2'] \* 4, kernel\_size=3, padding=1) |
|  | self.loc\_conv10\_2 = nn.Conv2d(256, n\_boxes['conv10\_2'] \* 4, kernel\_size=3, padding=1) |
|  | self.loc\_conv11\_2 = nn.Conv2d(256, n\_boxes['conv11\_2'] \* 4, kernel\_size=3, padding=1) |
|  |  |
|  | # Class prediction convolutions (predict classes in localization boxes) |
|  | self.cl\_conv4\_3 = nn.Conv2d(512, n\_boxes['conv4\_3'] \* n\_classes, kernel\_size=3, padding=1) |
|  | self.cl\_conv7 = nn.Conv2d(1024, n\_boxes['conv7'] \* n\_classes, kernel\_size=3, padding=1) |
|  | self.cl\_conv8\_2 = nn.Conv2d(512, n\_boxes['conv8\_2'] \* n\_classes, kernel\_size=3, padding=1) |
|  | self.cl\_conv9\_2 = nn.Conv2d(256, n\_boxes['conv9\_2'] \* n\_classes, kernel\_size=3, padding=1) |
|  | self.cl\_conv10\_2 = nn.Conv2d(256, n\_boxes['conv10\_2'] \* n\_classes, kernel\_size=3, padding=1) |
|  | self.cl\_conv11\_2 = nn.Conv2d(256, n\_boxes['conv11\_2'] \* n\_classes, kernel\_size=3, padding=1) |
|  |  |
|  | # Initialize convolutions' parameters |
|  | self.init\_conv2d() |
|  |  |
|  | def init\_conv2d(self): |
|  | """ |
|  | Initialize convolution parameters. |
|  | """ |
|  | for c in self.children(): |
|  | if isinstance(c, nn.Conv2d): |
|  | nn.init.xavier\_uniform\_(c.weight) |
|  | nn.init.constant\_(c.bias, 0.) |
|  |  |
|  | def forward(self, conv4\_3\_feats, conv7\_feats, conv8\_2\_feats, conv9\_2\_feats, conv10\_2\_feats, conv11\_2\_feats): |
|  | """ |
|  | Forward propagation. |
|  |  |
|  | :param conv4\_3\_feats: conv4\_3 feature map, a tensor of dimensions (N, 512, 38, 38) |
|  | :param conv7\_feats: conv7 feature map, a tensor of dimensions (N, 1024, 19, 19) |
|  | :param conv8\_2\_feats: conv8\_2 feature map, a tensor of dimensions (N, 512, 10, 10) |
|  | :param conv9\_2\_feats: conv9\_2 feature map, a tensor of dimensions (N, 256, 5, 5) |
|  | :param conv10\_2\_feats: conv10\_2 feature map, a tensor of dimensions (N, 256, 3, 3) |
|  | :param conv11\_2\_feats: conv11\_2 feature map, a tensor of dimensions (N, 256, 1, 1) |
|  | :return: 8732 locations and class scores (i.e. w.r.t each prior box) for each image |
|  | """ |
|  | batch\_size = conv4\_3\_feats.size(0) |
|  |  |
|  | # Predict localization boxes' bounds (as offsets w.r.t prior-boxes) |
|  | l\_conv4\_3 = self.loc\_conv4\_3(conv4\_3\_feats) # (N, 16, 38, 38) |
|  | l\_conv4\_3 = l\_conv4\_3.permute(0, 2, 3, |
|  | 1).contiguous() # (N, 38, 38, 16), to match prior-box order (after .view()) |
|  | # (.contiguous() ensures it is stored in a contiguous chunk of memory, needed for .view() below) |
|  | l\_conv4\_3 = l\_conv4\_3.view(batch\_size, -1, 4) # (N, 5776, 4), there are a total 5776 boxes on this feature map |
|  |  |
|  | l\_conv7 = self.loc\_conv7(conv7\_feats) # (N, 24, 19, 19) |
|  | l\_conv7 = l\_conv7.permute(0, 2, 3, 1).contiguous() # (N, 19, 19, 24) |
|  | l\_conv7 = l\_conv7.view(batch\_size, -1, 4) # (N, 2166, 4), there are a total 2116 boxes on this feature map |
|  |  |
|  | l\_conv8\_2 = self.loc\_conv8\_2(conv8\_2\_feats) # (N, 24, 10, 10) |
|  | l\_conv8\_2 = l\_conv8\_2.permute(0, 2, 3, 1).contiguous() # (N, 10, 10, 24) |
|  | l\_conv8\_2 = l\_conv8\_2.view(batch\_size, -1, 4) # (N, 600, 4) |
|  |  |
|  | l\_conv9\_2 = self.loc\_conv9\_2(conv9\_2\_feats) # (N, 24, 5, 5) |
|  | l\_conv9\_2 = l\_conv9\_2.permute(0, 2, 3, 1).contiguous() # (N, 5, 5, 24) |
|  | l\_conv9\_2 = l\_conv9\_2.view(batch\_size, -1, 4) # (N, 150, 4) |
|  |  |
|  | l\_conv10\_2 = self.loc\_conv10\_2(conv10\_2\_feats) # (N, 16, 3, 3) |
|  | l\_conv10\_2 = l\_conv10\_2.permute(0, 2, 3, 1).contiguous() # (N, 3, 3, 16) |
|  | l\_conv10\_2 = l\_conv10\_2.view(batch\_size, -1, 4) # (N, 36, 4) |
|  |  |
|  | l\_conv11\_2 = self.loc\_conv11\_2(conv11\_2\_feats) # (N, 16, 1, 1) |
|  | l\_conv11\_2 = l\_conv11\_2.permute(0, 2, 3, 1).contiguous() # (N, 1, 1, 16) |
|  | l\_conv11\_2 = l\_conv11\_2.view(batch\_size, -1, 4) # (N, 4, 4) |
|  |  |
|  | # Predict classes in localization boxes |
|  | c\_conv4\_3 = self.cl\_conv4\_3(conv4\_3\_feats) # (N, 4 \* n\_classes, 38, 38) |
|  | c\_conv4\_3 = c\_conv4\_3.permute(0, 2, 3, |
|  | 1).contiguous() # (N, 38, 38, 4 \* n\_classes), to match prior-box order (after .view()) |
|  | c\_conv4\_3 = c\_conv4\_3.view(batch\_size, -1, |
|  | self.n\_classes) # (N, 5776, n\_classes), there are a total 5776 boxes on this feature map |
|  |  |
|  | c\_conv7 = self.cl\_conv7(conv7\_feats) # (N, 6 \* n\_classes, 19, 19) |
|  | c\_conv7 = c\_conv7.permute(0, 2, 3, 1).contiguous() # (N, 19, 19, 6 \* n\_classes) |
|  | c\_conv7 = c\_conv7.view(batch\_size, -1, |
|  | self.n\_classes) # (N, 2166, n\_classes), there are a total 2116 boxes on this feature map |
|  |  |
|  | c\_conv8\_2 = self.cl\_conv8\_2(conv8\_2\_feats) # (N, 6 \* n\_classes, 10, 10) |
|  | c\_conv8\_2 = c\_conv8\_2.permute(0, 2, 3, 1).contiguous() # (N, 10, 10, 6 \* n\_classes) |
|  | c\_conv8\_2 = c\_conv8\_2.view(batch\_size, -1, self.n\_classes) # (N, 600, n\_classes) |
|  |  |
|  | c\_conv9\_2 = self.cl\_conv9\_2(conv9\_2\_feats) # (N, 6 \* n\_classes, 5, 5) |
|  | c\_conv9\_2 = c\_conv9\_2.permute(0, 2, 3, 1).contiguous() # (N, 5, 5, 6 \* n\_classes) |
|  | c\_conv9\_2 = c\_conv9\_2.view(batch\_size, -1, self.n\_classes) # (N, 150, n\_classes) |
|  |  |
|  | c\_conv10\_2 = self.cl\_conv10\_2(conv10\_2\_feats) # (N, 4 \* n\_classes, 3, 3) |
|  | c\_conv10\_2 = c\_conv10\_2.permute(0, 2, 3, 1).contiguous() # (N, 3, 3, 4 \* n\_classes) |
|  | c\_conv10\_2 = c\_conv10\_2.view(batch\_size, -1, self.n\_classes) # (N, 36, n\_classes) |
|  |  |
|  | c\_conv11\_2 = self.cl\_conv11\_2(conv11\_2\_feats) # (N, 4 \* n\_classes, 1, 1) |
|  | c\_conv11\_2 = c\_conv11\_2.permute(0, 2, 3, 1).contiguous() # (N, 1, 1, 4 \* n\_classes) |
|  | c\_conv11\_2 = c\_conv11\_2.view(batch\_size, -1, self.n\_classes) # (N, 4, n\_classes) |
|  |  |
|  | # A total of 8732 boxes |
|  | # Concatenate in this specific order (i.e. must match the order of the prior-boxes) |
|  | locs = torch.cat([l\_conv4\_3, l\_conv7, l\_conv8\_2, l\_conv9\_2, l\_conv10\_2, l\_conv11\_2], dim=1) # (N, 8732, 4) |
|  | classes\_scores = torch.cat([c\_conv4\_3, c\_conv7, c\_conv8\_2, c\_conv9\_2, c\_conv10\_2, c\_conv11\_2], |
|  | dim=1) # (N, 8732, n\_classes) |
|  |  |
|  | return locs, classes\_scores |
|  |  |
|  |  |
|  | class SSD300(nn.Module): |
|  | """ |
|  | The SSD300 network - encapsulates the base VGG network, auxiliary, and prediction convolutions. |
|  | """ |
|  |  |
|  | def \_\_init\_\_(self, n\_classes): |
|  | super(SSD300, self).\_\_init\_\_() |
|  |  |
|  | self.n\_classes = n\_classes |
|  |  |
|  | self.base = VGGBase() |
|  | self.aux\_convs = AuxiliaryConvolutions() |
|  | self.pred\_convs = PredictionConvolutions(n\_classes) |
|  |  |
|  | # Since lower level features (conv4\_3\_feats) have considerably larger scales, we take the L2 norm and rescale |
|  | # Rescale factor is initially set at 20, but is learned for each channel during back-prop |
|  | self.rescale\_factors = nn.Parameter(torch.FloatTensor(1, 512, 1, 1)) # there are 512 channels in conv4\_3\_feats |
|  | nn.init.constant\_(self.rescale\_factors, 20) |
|  |  |
|  | # Prior boxes |
|  | self.priors\_cxcy = self.create\_prior\_boxes() |
|  |  |
|  | def forward(self, image): |
|  | """ |
|  | Forward propagation. |
|  |  |
|  | :param image: images, a tensor of dimensions (N, 3, 300, 300) |
|  | :return: 8732 locations and class scores (i.e. w.r.t each prior box) for each image |
|  | """ |
|  | # Run VGG base network convolutions (lower level feature map generators) |
|  | conv4\_3\_feats, conv7\_feats = self.base(image) # (N, 512, 38, 38), (N, 1024, 19, 19) |
|  |  |
|  | # Rescale conv4\_3 after L2 norm |
|  | norm = conv4\_3\_feats.pow(2).sum(dim=1, keepdim=True).sqrt() # (N, 1, 38, 38) |
|  | conv4\_3\_feats = conv4\_3\_feats / norm # (N, 512, 38, 38) |
|  | conv4\_3\_feats = conv4\_3\_feats \* self.rescale\_factors # (N, 512, 38, 38) |
|  | # (PyTorch autobroadcasts singleton dimensions during arithmetic) |
|  |  |
|  | # Run auxiliary convolutions (higher level feature map generators) |
|  | conv8\_2\_feats, conv9\_2\_feats, conv10\_2\_feats, conv11\_2\_feats = \ |
|  | self.aux\_convs(conv7\_feats) # (N, 512, 10, 10), (N, 256, 5, 5), (N, 256, 3, 3), (N, 256, 1, 1) |
|  |  |
|  | # Run prediction convolutions (predict offsets w.r.t prior-boxes and classes in each resulting localization box) |
|  | locs, classes\_scores = self.pred\_convs(conv4\_3\_feats, conv7\_feats, conv8\_2\_feats, conv9\_2\_feats, conv10\_2\_feats, |
|  | conv11\_2\_feats) # (N, 8732, 4), (N, 8732, n\_classes) |
|  |  |
|  | return locs, classes\_scores |
|  |  |
|  | def create\_prior\_boxes(self): |
|  | """ |
|  | Create the 8732 prior (default) boxes for the SSD300, as defined in the paper. |
|  |  |
|  | :return: prior boxes in center-size coordinates, a tensor of dimensions (8732, 4) |
|  | """ |
|  | fmap\_dims = {'conv4\_3': 38, |
|  | 'conv7': 19, |
|  | 'conv8\_2': 10, |
|  | 'conv9\_2': 5, |
|  | 'conv10\_2': 3, |
|  | 'conv11\_2': 1} |
|  |  |
|  | obj\_scales = {'conv4\_3': 0.1, |
|  | 'conv7': 0.2, |
|  | 'conv8\_2': 0.375, |
|  | 'conv9\_2': 0.55, |
|  | 'conv10\_2': 0.725, |
|  | 'conv11\_2': 0.9} |
|  |  |
|  | aspect\_ratios = {'conv4\_3': [1., 2., 0.5], |
|  | 'conv7': [1., 2., 3., 0.5, .333], |
|  | 'conv8\_2': [1., 2., 3., 0.5, .333], |
|  | 'conv9\_2': [1., 2., 3., 0.5, .333], |
|  | 'conv10\_2': [1., 2., 0.5], |
|  | 'conv11\_2': [1., 2., 0.5]} |
|  |  |
|  | fmaps = list(fmap\_dims.keys()) |
|  |  |
|  | prior\_boxes = [] |
|  |  |
|  | for k, fmap in enumerate(fmaps): |
|  | for i in range(fmap\_dims[fmap]): |
|  | for j in range(fmap\_dims[fmap]): |
|  | cx = (j + 0.5) / fmap\_dims[fmap] |
|  | cy = (i + 0.5) / fmap\_dims[fmap] |
|  |  |
|  | for ratio in aspect\_ratios[fmap]: |
|  | prior\_boxes.append([cx, cy, obj\_scales[fmap] \* sqrt(ratio), obj\_scales[fmap] / sqrt(ratio)]) |
|  |  |
|  | # For an aspect ratio of 1, use an additional prior whose scale is the geometric mean of the |
|  | # scale of the current feature map and the scale of the next feature map |
|  | if ratio == 1.: |
|  | try: |
|  | additional\_scale = sqrt(obj\_scales[fmap] \* obj\_scales[fmaps[k + 1]]) |
|  | # For the last feature map, there is no "next" feature map |
|  | except IndexError: |
|  | additional\_scale = 1. |
|  | prior\_boxes.append([cx, cy, additional\_scale, additional\_scale]) |
|  |  |
|  | prior\_boxes = torch.FloatTensor(prior\_boxes).to(device) # (8732, 4) |
|  | prior\_boxes.clamp\_(0, 1) # (8732, 4) |
|  |  |
|  | return prior\_boxes |
|  |  |
|  | def detect\_objects(self, predicted\_locs, predicted\_scores, min\_score, max\_overlap, top\_k): |
|  | """ |
|  | Decipher the 8732 locations and class scores (output of ths SSD300) to detect objects. |
|  |  |
|  | For each class, perform Non-Maximum Suppression (NMS) on boxes that are above a minimum threshold. |
|  |  |
|  | :param predicted\_locs: predicted locations/boxes w.r.t the 8732 prior boxes, a tensor of dimensions (N, 8732, 4) |
|  | :param predicted\_scores: class scores for each of the encoded locations/boxes, a tensor of dimensions (N, 8732, n\_classes) |
|  | :param min\_score: minimum threshold for a box to be considered a match for a certain class |
|  | :param max\_overlap: maximum overlap two boxes can have so that the one with the lower score is not suppressed via NMS |
|  | :param top\_k: if there are a lot of resulting detection across all classes, keep only the top 'k' |
|  | :return: detections (boxes, labels, and scores), lists of length batch\_size |
|  | """ |
|  | batch\_size = predicted\_locs.size(0) |
|  | n\_priors = self.priors\_cxcy.size(0) |
|  | predicted\_scores = F.softmax(predicted\_scores, dim=2) # (N, 8732, n\_classes) |
|  |  |
|  | # Lists to store final predicted boxes, labels, and scores for all images |
|  | all\_images\_boxes = list() |
|  | all\_images\_labels = list() |
|  | all\_images\_scores = list() |
|  |  |
|  | assert n\_priors == predicted\_locs.size(1) == predicted\_scores.size(1) |
|  |  |
|  | for i in range(batch\_size): |
|  | # Decode object coordinates from the form we regressed predicted boxes to |
|  | decoded\_locs = cxcy\_to\_xy( |
|  | gcxgcy\_to\_cxcy(predicted\_locs[i], self.priors\_cxcy)) # (8732, 4), these are fractional pt. coordinates |
|  |  |
|  | # Lists to store boxes and scores for this image |
|  | image\_boxes = list() |
|  | image\_labels = list() |
|  | image\_scores = list() |
|  |  |
|  | max\_scores, best\_label = predicted\_scores[i].max(dim=1) # (8732) |
|  |  |
|  | # Check for each class |
|  | for c in range(1, self.n\_classes): |
|  | # Keep only predicted boxes and scores where scores for this class are above the minimum score |
|  | class\_scores = predicted\_scores[i][:, c] # (8732) |
|  | score\_above\_min\_score = class\_scores > min\_score # torch.uint8 (byte) tensor, for indexing |
|  | n\_above\_min\_score = score\_above\_min\_score.sum().item() |
|  | if n\_above\_min\_score == 0: |
|  | continue |
|  | class\_scores = class\_scores[score\_above\_min\_score] # (n\_qualified), n\_min\_score <= 8732 |
|  | class\_decoded\_locs = decoded\_locs[score\_above\_min\_score] # (n\_qualified, 4) |
|  |  |
|  | # Sort predicted boxes and scores by scores |
|  | class\_scores, sort\_ind = class\_scores.sort(dim=0, descending=True) # (n\_qualified), (n\_min\_score) |
|  | class\_decoded\_locs = class\_decoded\_locs[sort\_ind] # (n\_min\_score, 4) |
|  |  |
|  | # Find the overlap between predicted boxes |
|  | overlap = find\_jaccard\_overlap(class\_decoded\_locs, class\_decoded\_locs) # (n\_qualified, n\_min\_score) |
|  |  |
|  | # Non-Maximum Suppression (NMS) |
|  |  |
|  | # A torch.uint8 (byte) tensor to keep track of which predicted boxes to suppress |
|  | # 1 implies suppress, 0 implies don't suppress |
|  | suppress = torch.zeros((n\_above\_min\_score), dtype=torch.uint8).to(device) # (n\_qualified) |
|  |  |
|  | # Consider each box in order of decreasing scores |
|  | for box in range(class\_decoded\_locs.size(0)): |
|  | # If this box is already marked for suppression |
|  | if suppress[box] == 1: |
|  | continue |
|  |  |
|  | # Suppress boxes whose overlaps (with this box) are greater than maximum overlap |
|  | # Find such boxes and update suppress indices |
|  | suppress = torch.max(suppress, overlap[box] > max\_overlap) |
|  | # The max operation retains previously suppressed boxes, like an 'OR' operation |
|  |  |
|  | # Don't suppress this box, even though it has an overlap of 1 with itself |
|  | suppress[box] = 0 |
|  |  |
|  | # Store only unsuppressed boxes for this class |
|  | image\_boxes.append(class\_decoded\_locs[1 - suppress]) |
|  | image\_labels.append(torch.LongTensor((1 - suppress).sum().item() \* [c]).to(device)) |
|  | image\_scores.append(class\_scores[1 - suppress]) |
|  |  |
|  | # If no object in any class is found, store a placeholder for 'background' |
|  | if len(image\_boxes) == 0: |
|  | image\_boxes.append(torch.FloatTensor([[0., 0., 1., 1.]]).to(device)) |
|  | image\_labels.append(torch.LongTensor([0]).to(device)) |
|  | image\_scores.append(torch.FloatTensor([0.]).to(device)) |
|  |  |
|  | # Concatenate into single tensors |
|  | image\_boxes = torch.cat(image\_boxes, dim=0) # (n\_objects, 4) |
|  | image\_labels = torch.cat(image\_labels, dim=0) # (n\_objects) |
|  | image\_scores = torch.cat(image\_scores, dim=0) # (n\_objects) |
|  | n\_objects = image\_scores.size(0) |
|  |  |
|  | # Keep only the top k objects |
|  | if n\_objects > top\_k: |
|  | image\_scores, sort\_ind = image\_scores.sort(dim=0, descending=True) |
|  | image\_scores = image\_scores[:top\_k] # (top\_k) |
|  | image\_boxes = image\_boxes[sort\_ind][:top\_k] # (top\_k, 4) |
|  | image\_labels = image\_labels[sort\_ind][:top\_k] # (top\_k) |
|  |  |
|  | # Append to lists that store predicted boxes and scores for all images |
|  | all\_images\_boxes.append(image\_boxes) |
|  | all\_images\_labels.append(image\_labels) |
|  | all\_images\_scores.append(image\_scores) |
|  |  |
|  | return all\_images\_boxes, all\_images\_labels, all\_images\_scores # lists of length batch\_size |
|  |  |
|  |  |
|  | class MultiBoxLoss(nn.Module): |
|  | """ |
|  | The MultiBox loss, a loss function for object detection. |
|  |  |
|  | This is a combination of: |
|  | (1) a localization loss for the predicted locations of the boxes, and |
|  | (2) a confidence loss for the predicted class scores. |
|  | """ |
|  |  |
|  | def \_\_init\_\_(self, priors\_cxcy, threshold=0.5, neg\_pos\_ratio=3, alpha=1.): |
|  | super(MultiBoxLoss, self).\_\_init\_\_() |
|  | self.priors\_cxcy = priors\_cxcy |
|  | self.priors\_xy = cxcy\_to\_xy(priors\_cxcy) |
|  | self.threshold = threshold |
|  | self.neg\_pos\_ratio = neg\_pos\_ratio |
|  | self.alpha = alpha |
|  |  |
|  | self.smooth\_l1 = nn.L1Loss() |
|  | self.cross\_entropy = nn.CrossEntropyLoss(reduce=False) |
|  |  |
|  | def forward(self, predicted\_locs, predicted\_scores, boxes, labels): |
|  | """ |
|  | Forward propagation. |
|  |  |
|  | :param predicted\_locs: predicted locations/boxes w.r.t the 8732 prior boxes, a tensor of dimensions (N, 8732, 4) |
|  | :param predicted\_scores: class scores for each of the encoded locations/boxes, a tensor of dimensions (N, 8732, n\_classes) |
|  | :param boxes: true object bounding boxes in boundary coordinates, a list of N tensors |
|  | :param labels: true object labels, a list of N tensors |
|  | :return: multibox loss, a scalar |
|  | """ |
|  | batch\_size = predicted\_locs.size(0) |
|  | n\_priors = self.priors\_cxcy.size(0) |
|  | n\_classes = predicted\_scores.size(2) |
|  |  |
|  | assert n\_priors == predicted\_locs.size(1) == predicted\_scores.size(1) |
|  |  |
|  | true\_locs = torch.zeros((batch\_size, n\_priors, 4), dtype=torch.float).to(device) # (N, 8732, 4) |
|  | true\_classes = torch.zeros((batch\_size, n\_priors), dtype=torch.long).to(device) # (N, 8732) |
|  |  |
|  | # For each image |
|  | for i in range(batch\_size): |
|  | n\_objects = boxes[i].size(0) |
|  |  |
|  | overlap = find\_jaccard\_overlap(boxes[i], |
|  | self.priors\_xy) # (n\_objects, 8732) |
|  |  |
|  | # For each prior, find the object that has the maximum overlap |
|  | overlap\_for\_each\_prior, object\_for\_each\_prior = overlap.max(dim=0) # (8732) |
|  |  |
|  | # We don't want a situation where an object is not represented in our positive (non-background) priors - |
|  | # 1. An object might not be the best object for all priors, and is therefore not in object\_for\_each\_prior. |
|  | # 2. All priors with the object may be assigned as background based on the threshold (0.5). |
|  |  |
|  | # To remedy this - |
|  | # First, find the prior that has the maximum overlap for each object. |
|  | \_, prior\_for\_each\_object = overlap.max(dim=1) # (N\_o) |
|  |  |
|  | # Then, assign each object to the corresponding maximum-overlap-prior. (This fixes 1.) |
|  | object\_for\_each\_prior[prior\_for\_each\_object] = torch.LongTensor(range(n\_objects)).to(device) |
|  |  |
|  | # To ensure these priors qualify, artificially give them an overlap of greater than 0.5. (This fixes 2.) |
|  | overlap\_for\_each\_prior[prior\_for\_each\_object] = 1. |
|  |  |
|  | # Labels for each prior |
|  | label\_for\_each\_prior = labels[i][object\_for\_each\_prior] # (8732) |
|  | # Set priors whose overlaps with objects are less than the threshold to be background (no object) |
|  | label\_for\_each\_prior[overlap\_for\_each\_prior < self.threshold] = 0 # (8732) |
|  |  |
|  | # Store |
|  | true\_classes[i] = label\_for\_each\_prior |
|  |  |
|  | # Encode center-size object coordinates into the form we regressed predicted boxes to |
|  | true\_locs[i] = cxcy\_to\_gcxgcy(xy\_to\_cxcy(boxes[i][object\_for\_each\_prior]), self.priors\_cxcy) # (8732, 4) |
|  |  |
|  | # Identify priors that are positive (object/non-background) |
|  | positive\_priors = true\_classes != 0 # (N, 8732) |
|  |  |
|  | # LOCALIZATION LOSS |
|  |  |
|  | # Localization loss is computed only over positive (non-background) priors |
|  | loc\_loss = self.smooth\_l1(predicted\_locs[positive\_priors], true\_locs[positive\_priors]) # (), scalar |
|  |  |
|  | # Note: indexing with a torch.uint8 (byte) tensor flattens the tensor when indexing is across multiple dimensions (N & 8732) |
|  | # So, if predicted\_locs has the shape (N, 8732, 4), predicted\_locs[positive\_priors] will have (total positives, 4) |
|  |  |
|  | # CONFIDENCE LOSS |
|  |  |
|  | # Confidence loss is computed over positive priors and the most difficult (hardest) negative priors in each image |
|  | # That is, FOR EACH IMAGE, |
|  | # we will take the hardest (neg\_pos\_ratio \* n\_positives) negative priors, i.e where there is maximum loss |
|  | # This is called Hard Negative Mining - it concentrates on hardest negatives in each image, and also minimizes pos/neg imbalance |
|  |  |
|  | # Number of positive and hard-negative priors per image |
|  | n\_positives = positive\_priors.sum(dim=1) # (N) |
|  | n\_hard\_negatives = self.neg\_pos\_ratio \* n\_positives # (N) |
|  |  |
|  | # First, find the loss for all priors |
|  | conf\_loss\_all = self.cross\_entropy(predicted\_scores.view(-1, n\_classes), true\_classes.view(-1)) # (N \* 8732) |
|  | conf\_loss\_all = conf\_loss\_all.view(batch\_size, n\_priors) # (N, 8732) |
|  |  |
|  | # We already know which priors are positive |
|  | conf\_loss\_pos = conf\_loss\_all[positive\_priors] # (sum(n\_positives)) |
|  |  |
|  | # Next, find which priors are hard-negative |
|  | # To do this, sort ONLY negative priors in each image in order of decreasing loss and take top n\_hard\_negatives |
|  | conf\_loss\_neg = conf\_loss\_all.clone() # (N, 8732) |
|  | conf\_loss\_neg[positive\_priors] = 0. # (N, 8732), positive priors are ignored (never in top n\_hard\_negatives) |
|  | conf\_loss\_neg, \_ = conf\_loss\_neg.sort(dim=1, descending=True) # (N, 8732), sorted by decreasing hardness |
|  | hardness\_ranks = torch.LongTensor(range(n\_priors)).unsqueeze(0).expand\_as(conf\_loss\_neg).to(device) # (N, 8732) |
|  | hard\_negatives = hardness\_ranks < n\_hard\_negatives.unsqueeze(1) # (N, 8732) |
|  | conf\_loss\_hard\_neg = conf\_loss\_neg[hard\_negatives] # (sum(n\_hard\_negatives)) |
|  |  |
|  | # As in the paper, averaged over positive priors only, although computed over both positive and hard-negative priors |
|  | conf\_loss = (conf\_loss\_hard\_neg.sum() + conf\_loss\_pos.sum()) / n\_positives.sum().float() # (), scalar |
|  |  |
|  | # TOTAL LOSS |
|  |  |
|  | return conf\_loss + self.alpha \* loc\_loss |