Yolov3 Note

Yiwei Ren

last changed on 2023 年 2 月 22 日

1 总览

1.1 特征提取网络

使用 Darknet 网络架构, 全卷积网络并带有 res 残差层. 分别对于三个不同的网格大小 (即感受野大小) 进行分别的三次特征提取, 同时对于特征提取后的特征图进行分部分次叠加, 形成三个用不同大小的网格形成的网格图下的特征图.

1.2 detect 过程

1.2.1 对特征图解码

生成的三个特征图每一个的 shape 是

$$[bs, Num\_single\_anchors * (5 + Num\_class), num\_gridx, num\_gridy]$$

其中 Num_single_anchors 是单个网格点需要生成的锚框数量 Num_class 是需要分类的类别数量, 5 是预测锚框的相对于所在网格中心的归一化的 xywh 信息以及含有物体置信度, num_gridx 和 num_gridy 是生成网格数量. 对于单个特征图下的特征图, 需要对其进行变换, 变成相对于整个图片的 xywh 信息并再次归一化, 并且改变张量维度, 将之变成

$$[bs, num\_all\_anchors, 5 + num_class]$$

也就是 batch size 下的所有锚框的数量 (每个点生成的锚框数目乘点的数量), 每一个锚框都有类别概率, xywh 和含有物体置信度的若干信息.

1.2.2 锚框筛选

对于解码后的张量, 需要对所有锚框预测结果进行筛选, 去掉低于含有物体置信度阈值的锚框. 对于保留下来的锚框, 还需要进行进一步的整理, 对于留下的含有物体的锚框, 选择其 class 概率最高的种类作为预测的种类. 最后对于预测的同一个种类下的锚框们, 还需要进行非极大抑制, 即高于一定 iou 阈值的两个锚框, 保留其拥有该类别下拥有最大预测概率的锚框.

1.2.3 绘图

对于筛选后的锚框, 需要对其进行逆向归一化, 将之在图片上真实的坐标反解出来. 随后就可对其进行检测框的绘制.

1.3 train 过程

1.3.1 加载数据集

对于需要训练的每一张图片, 将图片中的每一个检测框单独作为一个样本进行训练, 载入时, 最好将之坐标归一化

1.3.2 建立损失函数与训练

先进行解码, 解码到

$$[bs, Num\_single\_anchors, num\_gridx, num\_gridy, 7]$$

其中 7 = xywh+have_conf+class_conf+class_pred. 随后将样本所在网格计算出, 并将该网格下的锚框们视作正样本, 计算损失. 对于其他所有的网格, 视作负样本, 计算负样本损失.

随后即可开始训练和验证过程

2 特征提取网络搭建

根据下图搭建特征提取网络.

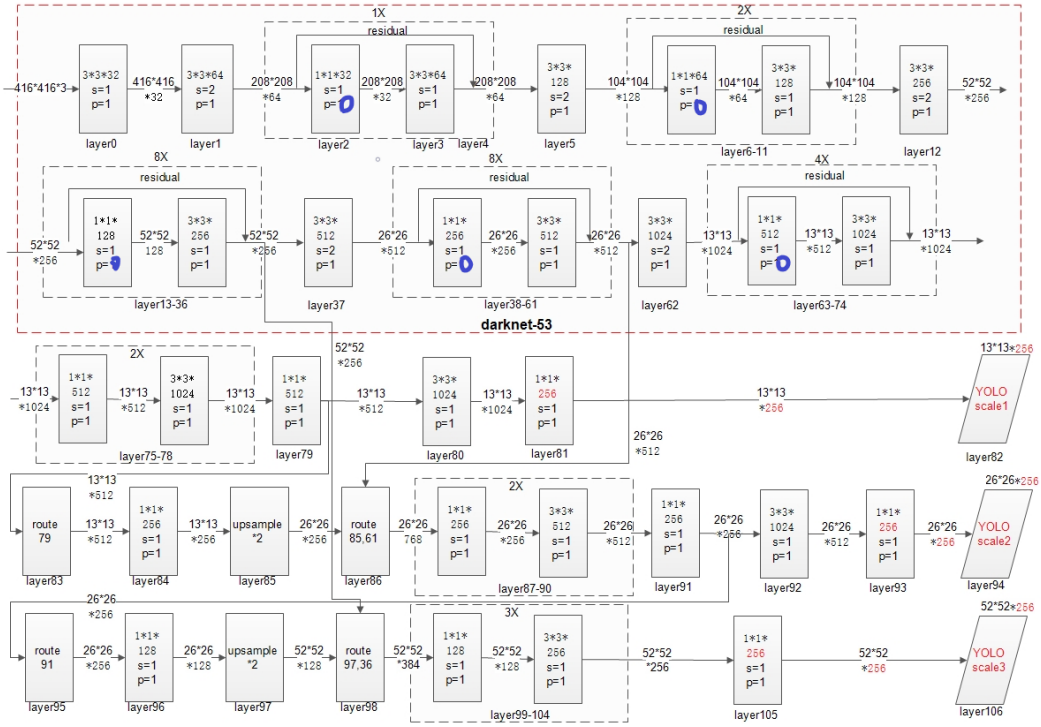

图 1: yolov3 网络结构

需要注意的是在每一个卷积后, 都有 BatchNormal 层和 RELU 层进行处理.(且其中的 padding 都应该为 0)

但是在得到了三个 scale 窗口后, 并没有结束, 还需要对得到的特征图进行通道变换, 将其 256 的通道改变为 Num_single_anchors*(Num_class+5) 的通道数.

3 预测解码

首先针对整个数据集进行锚框大小聚类. 每一种网格数目聚类 Num_single_anchors 个, 并将每一个网格数目下所得的大小存为一个 list.

以下的操作均为在指定网格数目下的操作. 聚类得出的大小是 (聚类 w, 聚类 h).

随后处理特征图. 每一个特征提取出来的图像 shape 为

$$[bs, Num\_single\_anchors * (Num\_class + 5), num\_gridx, num\_gridy]$$

先调整尺寸, 将之变成下列 shape

$$[bs, Num\_single\_anchors, num\_gridx, num\_gridy, 5 + Num\_class]$$

并对该张量进行处理, 将张量最后一个维度中的 xy, 含有物体置信度, 和整个 class 置信度进行 sigmoid 归一化. 随后计算缩放比例, 即特征图上的一个像素点相当于原图上多少 w 和 h, 并计算 list 中锚框大小对应到特征图中的相对大小.

再计算锚框的调整. 上述张量中的 xy 均是预测的锚框的中心点相对于锚框所在网格的左上角点的偏移量的归一化形式, wh 为锚框的相对于聚类所得大小的偏移量. 故由此可以计算出锚框真正的坐标位置和大小. 先生成一个以 num_gridx 为 linespace 终点的序列, 并将之重复 num_gridy 次形成二维数组, 并将之扩展为 shape

$$[bs, Num\_single\_anchors, num\_gridx, num\_gridy]$$

随后和 x 相加即可得到 x 的网格真实位置. 同理可得到 y 的. 而对于 wh 聚类偏移量, 需要对于单个的 w 或者 h 进行扩展, 依旧是上述的 shape, 但是其是和 exp(特征图的 w 或 h) 进行相加.

最后将之归一化. 对于得到的锚框关于网格真实的位置和大小, 对其除以网格的数量, 即可得到归一化下的结果.

4 锚框筛选

对于上述得到的锚框, 还需要筛选. 对于上述得到的张量的类别维度进行 argmax 处理, 得到该锚框的预测类别. 其中同一预测类别下的锚框可能有重叠, 此时需要对其进行进一步的筛选, 选择除去可能存在的重叠非常多的同类的锚框. 此时使用 nms 非极大抑制方法, 对于同一个类别下的锚框集合计算 iou, 将大于一定 iou 阈值的同类锚框们只保留预测概率最大的锚框, 去掉其他的锚框.

随后进行锚框的反归一化, 将他们归一化尺寸返回到原图尺寸, 并进行绘制.

5 数据集构建与加载

对于 coco 数据集, torchvision 自带的需要变换使用, 比较麻烦, 于是就重写了一个.

下载下来的 annotations 文件中的 instance.json 文件描述了图片中目标检测和图像分割的信息, 使用 pycocotools.coco 中的 COCO 类自动获得 json 文件中的 index, 先要用整个 COCO 类进行实例化载入内存后得到实例 coco, 实例 coco 是对 json 文件的解析. coco.imgs 存储一个大字典, 其中对于每一张图片都进行了 id 编号, 每一个图片 id 下的都是一个小字典, 其中包含了图片的宽高, 名字等信息. coco.imgToAnns 存储一个图片 id 大字典, 其中对应每个 id 下的都是一个字典 list, list 中的每一个字典都存放了一个图像分割信息以及对应的 bbox 坐标信息以及对应的类别号. 需要注意的是这里的 bbox 坐标是 [minx, miny, w, h] 形式的, 是左上角点的 xy 而不是中心点的 xy.

对该 coco 进行解析, 先将每一张图片对应的 bboxes 提取出来, 存一个字典, 将图片 id 作为 key, 将该图片内的 bboxes 存为一个二维数组 [[minx, miny, w, h, class]...], 并在存储之前完成坐标和大小的归一化.

重新排列样本. 遍历每一个图片 id 下的所有 bbox, 将单个 bbox 作为样本, 做两个字典, 分别用于存储 bbox 单独样本编号下的图片路径和 bbox 信息. (id 号是 int 用于 ___getitem___ 中得到顺序号). 最后重写 torch.utils.data.Dataset 类

6 损失函数的建立

首先通过对数据集上的图片进行聚类算法计算出划分成不同网格数下的锚框大小. 每一个网格数量下聚类三个不同大小的锚框. 并将每一对 hw 存为一个元组. 再将这三个元组存为一个 list.

对于从特征提取网络出来的三种不同网格数下的聚类锚框大小 list, 从中每次取出一个 list 计算该网格下的损失.

首先是对预测的锚框进行正负样本分类. 如果真实的框的中心在某个网格中, 那么该网格的三个锚框就负责预测该真实的框, 即这三个锚框为正样本. 而其他网格中的三个锚框为负样本. 对于正样本, 计算三大类损失.

1. 坐标损失. 计算该真实框和三个预测锚框的 ciou 并取 1-ciou 作为损失值

2. 分类损失. 用 BCElog 计算分类损失.

3. 含有物体置信度损失. 再利用 BCElog 计算置信度损失.

对于负样本, 只需要计算含有物体置信度损失. 需要注意的是, 负样本数量远多于正样本, 因此需要在负样本的损失函数前乘小权重.

最终将这些损失全部加和, 即得到损失.

7 PyTorch 细节

- Tensor 类数据结构. Tensor 类由头部信息和 Storage 实际存储组成. Storage 在内存中以一维数组形式存放而 Tensor 是外部封装的形状步长数据类型等信息. 切片 [:n] 操作, view 操作, squeeze, expend 等操作均属于浅复制操作不会改变 Storage. 所以这些切片等在进行修改时会将原来的母 Tensor 一并修改.

- 原地运算和异地运算. 原地运算均是函数后带有一个下划线 _ 的方法. 这样的函数会在原本的 Tensor 上进行运算并且不会引入新的内存使用. 但是这样有可能会覆盖梯度的取值, 需要减少使用.

- 深复制方法和浅复制方法. py 中 = 属于非典型浅复制. 对于数组 = 浅复制 (如 b = a), 则 ab 共享同一段内存但是如果整体改变 a 的取值 (如 a = [new1, new2, new3]) 则 a 会指向新的内存. 而 b 仍然指向原来 a 所在的内存位置, 故 ab 值不同. 而若是一个个改变其中的数值 (如 a[0] = new1, a[1] = new2, a[2] = new3) 则将会同时影响 b 的取值, 此时没有改变 a 指向的内存, 而是直接改变了内存中存下的数值. 因此此时 b 也是同步修改.

  而 torch 中, Tensor(), tensor() 是深复制, 在内存中新开辟一个 Storage 区域存储, 与原来的值不干扰. from_numpy(), as_tensor() 是浅复制, 在内存中公用一个 Storage.

  detach() 和 clone(). detach() 是浅复制, 即新的变量和原来的变量公用一个 Storage, 但是梯度从此处被截断. 新的变量的梯度不会流入到原来的变量上去. clone() 是深复制, 即新的变量重新指定一个 Storage. 但是新的变量的梯度会流入到原来的变量上去. 使用 clone().detach() 叠加即可得到一个新的 Storage 下的并且不将梯度流回到原变量的 Tensor.

  contiguous(). 在 transpose, permute 操作后, 改变了张量的形状, 但是在其 Storage 内部仍是以原来的数组顺序存放的 (改变了步长). 若此时再用 view 等方法就会报错, 所以需要用 contiguous 方法使 Storage 适应新形状下的一维内存, 以使 view 能正确运行. (详见知乎链接), 而使用 reshape 替代上述的操作会使用新的内存空间即默认 contiguous 化.