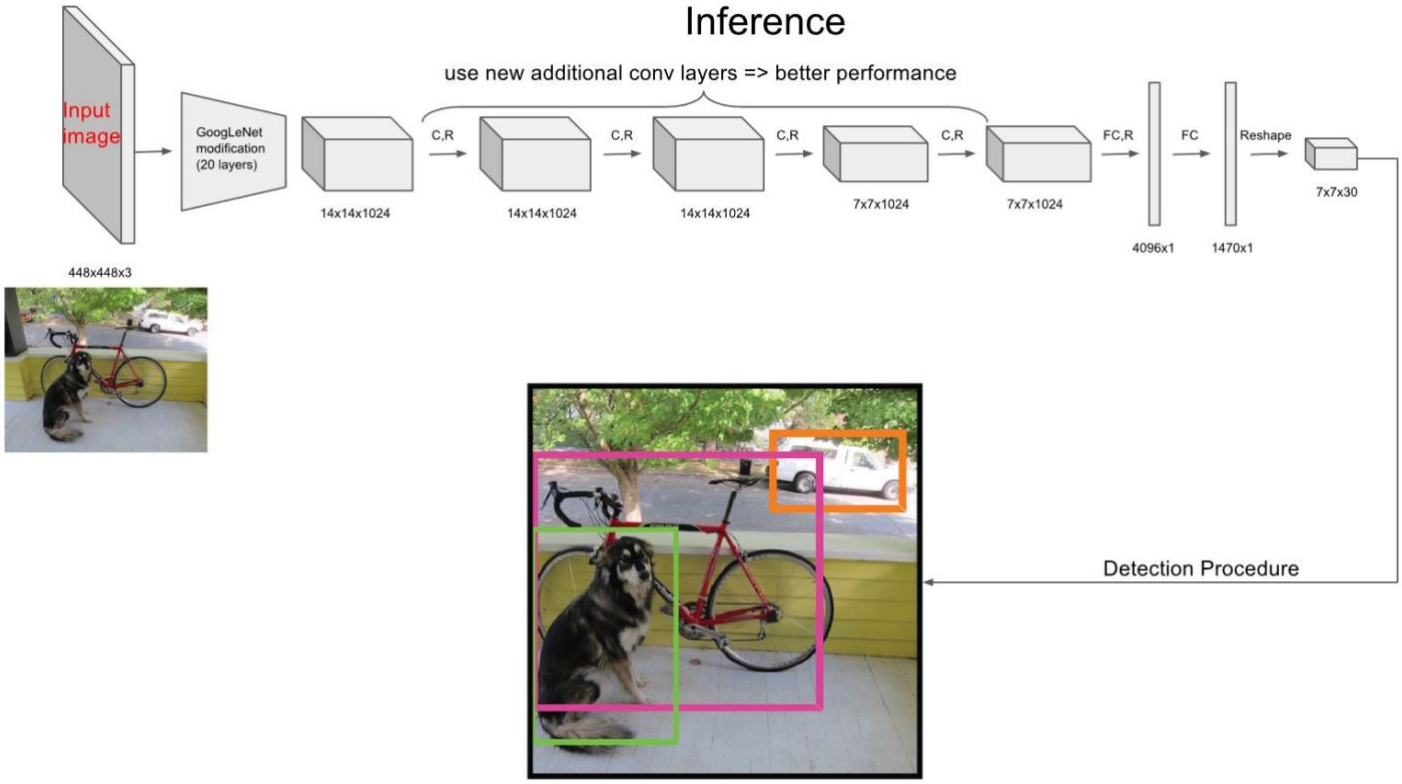
知乎 首发于 机器学习算法工程师



目标检测|YOLO原理与实现

小小将
为人民日益增长的美好生活需要而读书
关注他

码字不易，欢迎给个赞！

欢迎交流与转载，文章会同步发布在公众号：机器学习算法全栈工程师(Jeemy110)

最新的YOLOv2和YOLOv3:

小白将：目标检测|YOLOv2原理与实现(附YOLOv3)
zhuanlan.zhihu.com

等，见图1所示。其中目标检测是一件比较实际的且具有挑战性的计算机视觉任务，其可以看成图像分类与定位的结合，给定一张图片，目标检测系统要能够识别出图片的目标并给出其位置，由于图片中目标数是不定的，且要给出目标的精确位置，目标检测相比分类任务更复杂。目标检测的一个实际应用场景就是无人驾驶，如果能够在无人车上装载一个有效的目标检测系统，那么无人车将和人一样有了眼睛，可以快速地检测出前面的行人与车辆，从而作出实时决策。

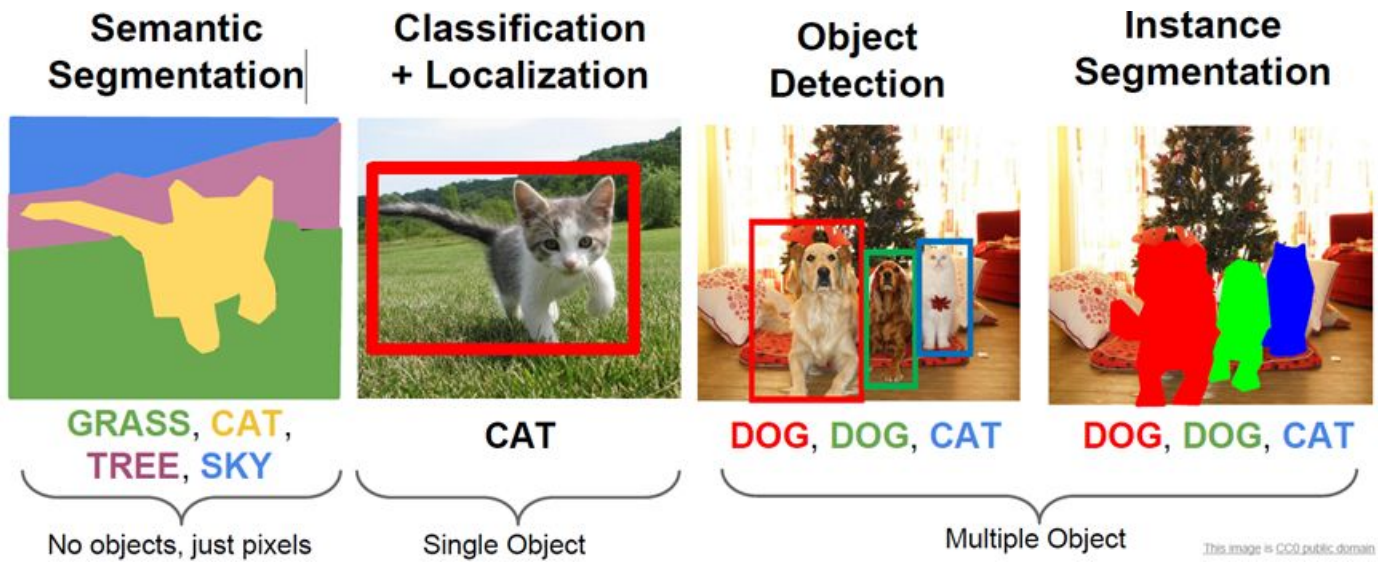


图1 计算机视觉任务（来源: cs231n）

近几年来，目标检测算法取得了很大的突破。比较流行的算法可以分为两类，一类是基于Region Proposal的R-CNN系算法（R-CNN，Fast R-CNN, Faster R-CNN），它们是two-stage的，需要先使用启发式方法（selective search）或者CNN网络（RPN）产生Region Proposal，然后再在Region Proposal上做分类与回归。而另一类是Yolo，SSD这类one-stage算法，其仅仅使用一个CNN网络直接预测不同目标的类别与位置。第一类方法是准确度高一些，但是速度慢，但是第二类算法是速度快，但是准确性要低一些。这可以在图2中看到。本文介绍的是Yolo算法，其全称是You Only Look Once: Unified, Real-Time Object Detection，其实个人觉得这个题目取得非常好，基本上把Yolo算法的特点概括全了：You Only Look Once说的是只需要一次CNN运算，Unified指的是这是一个统一的框架，提供end-to-end的预测，而Real-Time体现是Yolo算法速度快。这里我们谈的是Yolo-v1版本算法，其性能是差于后来的SSD算法的，但是Yolo后来也继续进行改进，产生了Yolo9000算法。本文主要讲述Yolo-v1算法的原理，特别是算法的训练与预测中详细细节，最后将给出如何使用TensorFlow实现Yolo算法。

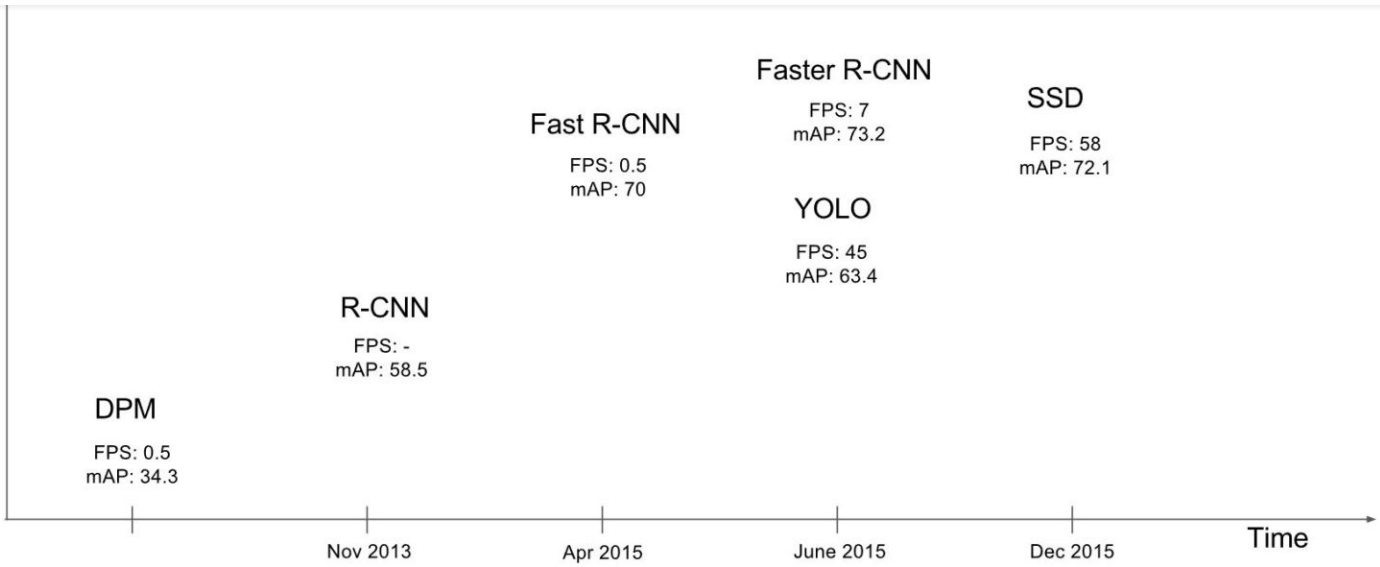


图2 目标检测算法进展与对比

滑动窗口与CNN

在介绍Yolo算法之前，首先先介绍一下滑动窗口技术，这对我们理解Yolo算法是有帮助的。采用滑动窗口的目标检测算法思路非常简单，它将检测问题转化为了图像分类问题。其基本原理就是采用不同大小和比例（宽高比）的窗口在整张图片上以一定的步长进行滑动，然后对这些窗口对应的区域做图像分类，这样就可以实现对整张图片的检测了，如下图3所示，如DPM就是采用这种思路。但是这个方法有致命的缺点，就是你并不知道要检测的目标大小是什么规模，所以你要设置不同大小和比例的窗口去滑动，而且还要选取合适的步长。但是这样会产生很多的子区域，并且都要经过分类器去做预测，这需要很大的计算量，所以你的分类器不能太复杂，因为要保证速度。解决思路之一就是减少要分类的子区域，这就是R-CNN的一个改进策略，其采用了selective search方法来找到最有可能包含目标的子区域（Region Proposal），其实可以看成采用启发式方法过滤掉很多子区域，这会提升效率。

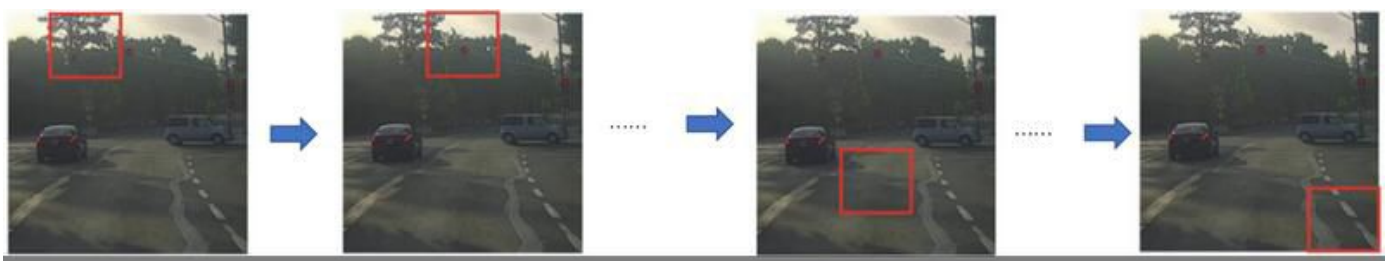


图3 采用滑动窗口进行目标检测（来源：deeplearning.ai）

如果你使用的是CNN分类器，那么滑动窗口是非常耗时的。但是结合卷积运算的特点，我们可以使用CNN实现更高效的滑动窗口方法。这里要介绍的是一种全卷积的方法，简单来说就是网络中用卷积层代替了全连接层，如图4所示。输入图片大小是16x16，经过一系列卷积操作，提取了2x2

积操作的特性，就是图片的空间位置信息的不变性，尽管卷积过程中图片大小减少，但是位置对应关系还是保存的。说点题外话，这个思路也被R-CNN借鉴，从而诞生了Fast R-cNN算法。

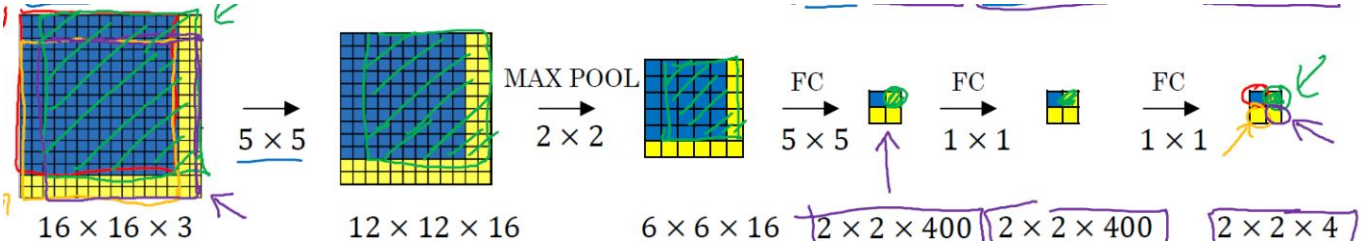


图4 滑动窗口的CNN实现（来源：deeplearning.ai）

上面尽管可以减少滑动窗口的计算量，但是只是针对一个固定大小与步长的窗口，这是远远不够的。Yolo算法很好的解决了这个问题，它不再是窗口滑动了，而是直接将原始图片分割成互不重合的小方块，然后通过卷积最后生产这样大小的特征图，基于上面的分析，可以认为特征图的每个元素也是对应原始图片的一个小方块，然后用每个元素来可以预测那些中心点在该小方格内的目标，这就是Yolo算法的朴素思想。下面将详细介绍Yolo算法的设计理念。

设计理念

整体来看，Yolo算法采用一个单独的CNN模型实现end-to-end的目标检测，整个系统如图5所示：首先将输入图片resize到448x448，然后送入CNN网络，最后处理网络预测结果得到检测的目标。相比R-CNN算法，其是一个统一的框架，其速度更快，而且Yolo的训练过程也是end-to-end的。

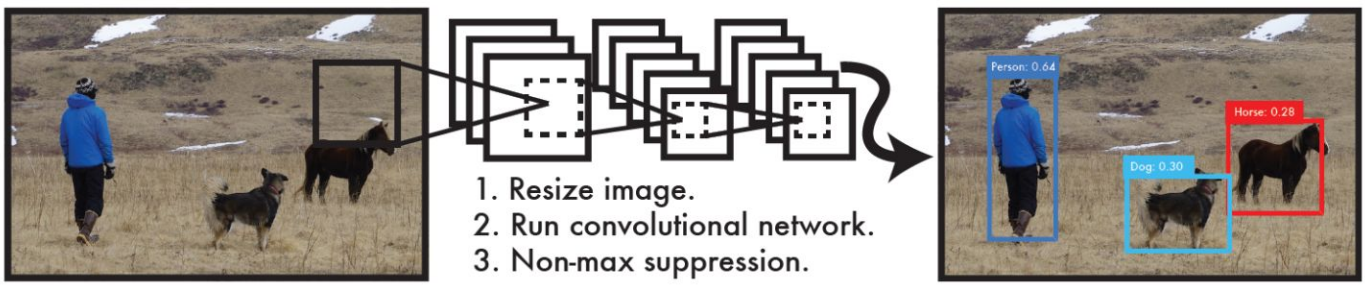


图5 Yolo检测系统

具体来说，Yolo的CNN网络将输入的图片分割成 $S \times S$ 网格，然后每个单元格负责去检测那些中心点落在该格子内的目标，如图6所示，可以看到狗这个目标的中心落在左下角一个单元格内，那么该单元格负责预测这个狗。每个单元格会预测 B 个边界框（bounding box）以及边界框的置信度（confidence score）。所谓置信度其实包含两个方面，一是这个边界框含有目标的可能性大小，二是这个边界框的准确度。前者记为 $Pr(object)$，当该边界框是背景时（即不包含目标），此时 $Pr(object)=0$。而当该边界框包含目标时，$Pr(object)=1$。边界框的准确度可以用预测框与实际框（ground truth）的IOU（intersection over union，交并比）来表征，记为 $\mathrm{IOU}_{pred}^{truth}$

并且单位是相对于单元格大小的，单元格的坐标定义如图6所示。而边界框的 w 和 h 预测值是相对于整个图片的宽与高的比例，这样理论上4个元素的大小应该在 $[0,1]$ 范围。这样，每个边界框的预测值实际上包含5个元素： (x,y,w,h,c) ，其中前4个表征边界框的大小与位置，而最后一个值是置信度。

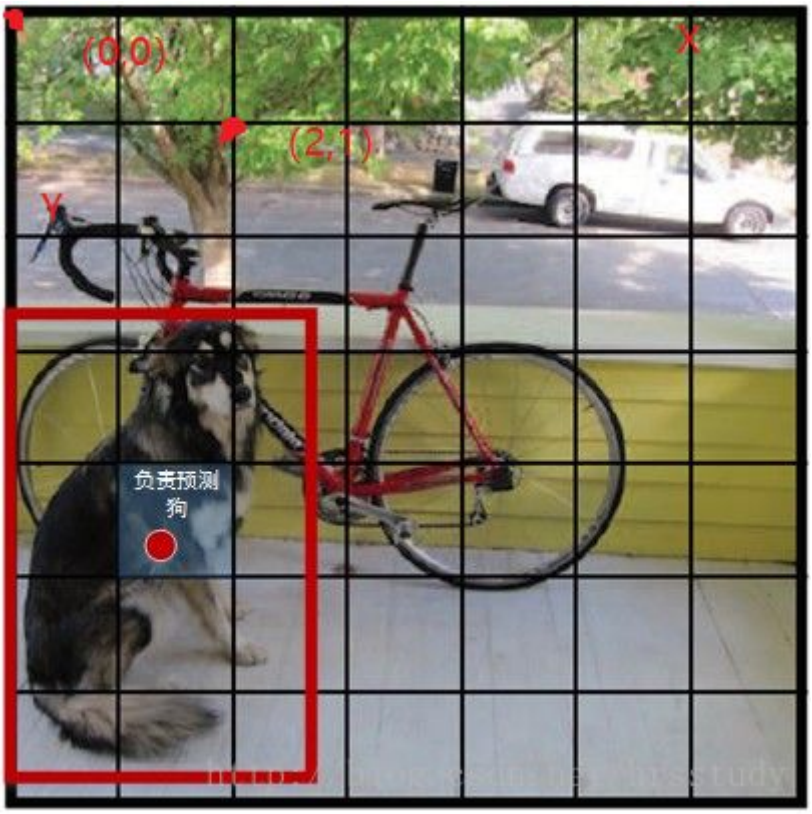


图6 网格划分

还有分类问题，对于每一个单元格其还要给出预测出 C 个类别概率值，其表征的是由该单元格负责预测的边界框其目标属于各个类别的概率。但是这些概率值其实是在各个边界框置信度下的条件概率，即 $Pr(class_i|object)$ 。值得注意的是，不管一个单元格预测多少个边界框，其只预测一组类别概率值，这是Yolo算法的一个缺点，在后来的改进版本中，Yolo9000是把类别概率预测值与边界框是绑定在一起的。同时，我们可以计算出各个边界框类别置信度（class-specific confidence scores）： $Pr(class_i|object)*Pr(object)*\text{IOU}_{pred}^{truth}=Pr(class_i)*\text{IOU}_{pred}^{truth}$ 。

边界框类别置信度表征的是该边界框中目标属于各个类别的可能性大小以及边界框匹配目标的好坏。后面会说，一般会根据类别置信度来过滤网络的预测框。

总结一下，每个单元格需要预测 $(B*5+C)$ 个值。如果将输入图片划分为 $S\times S$ 网格，那么最终预测值为 $S\times S\times(B*5+C)$ 大小的张量。整个模型的预测值结构如下图所示。对于PASCAL VOC数据，其共有20个类别，如果使用 $S=7,B=2$ ，那么最终的预测结果就是 $7\times 7\times 30$ 大小的张量。在下面的网络结构中我们会详细讲述每个单元格的预测值的分布位置。

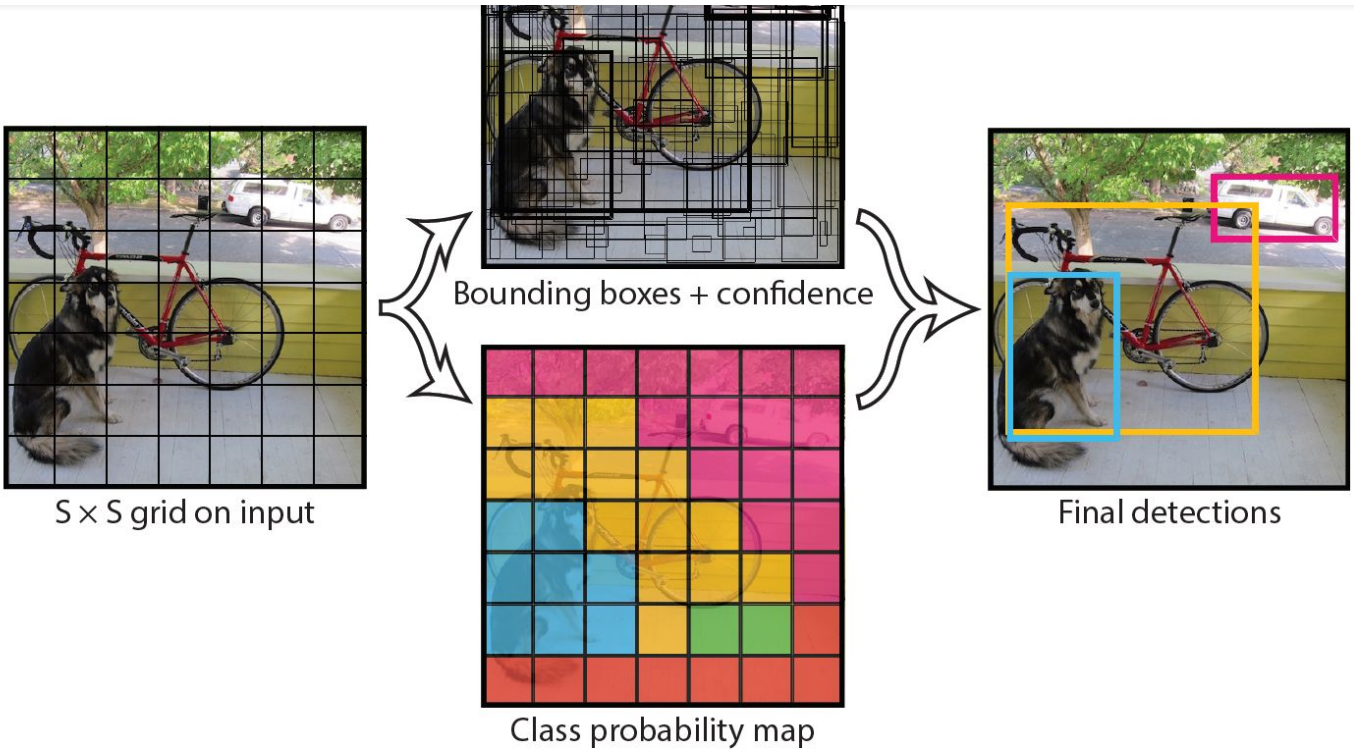


图7 模型预测值结构

网络设计

Yolo采用卷积网络来提取特征，然后使用全连接层来得到预测值。网络结构参考GooLeNet模型，包含24个卷积层和2个全连接层，如图8所示。对于卷积层，主要使用1x1卷积来做channle reduction，然后紧跟3x3卷积。对于卷积层和全连接层，采用Leaky ReLU激活函数：$max(x, 0.1x)$ 。但是最后一层却采用线性激活函数。

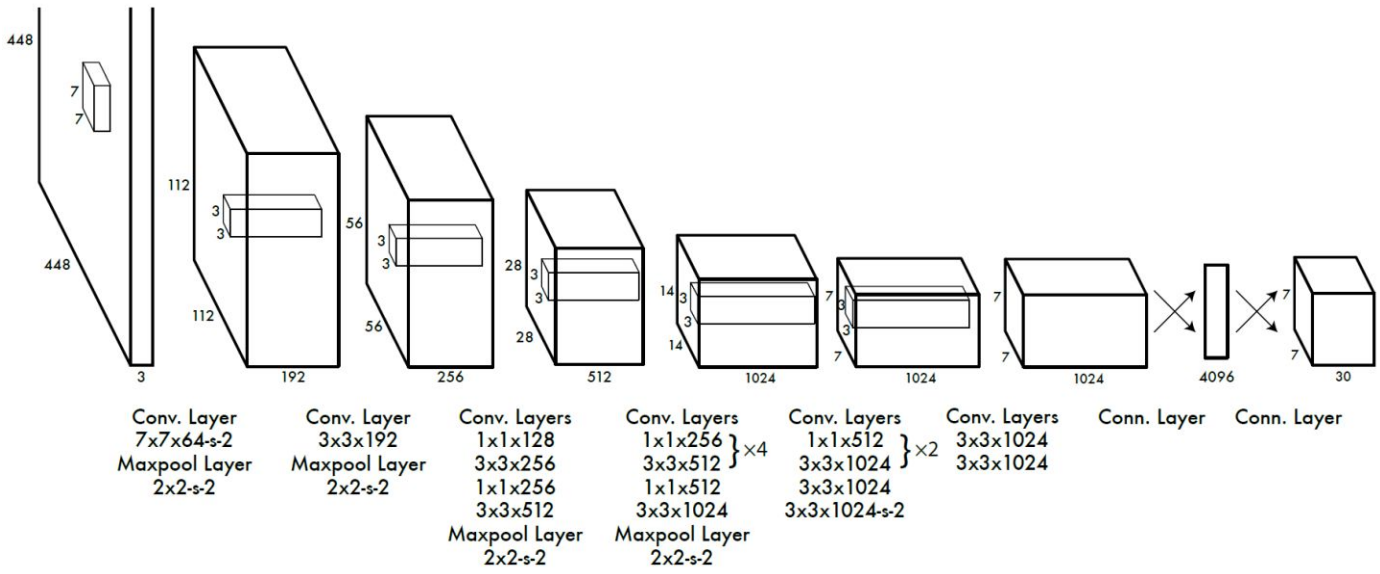


图8 网络结构

信度，两者相乘可以得到类别置信度，最后8个元素是边界框的 (x, y, w, h) 。大家可能会感到奇怪，对于边界框为什么把置信度 c 和 (x, y, w, h) 都分开排列，而不是按照 (x, y, w, h, c) 这样排列，其实纯粹是为了计算方便，因为实际上这30个元素都是对应一个单元格，其排列是可以任意的。但是分离排布，可以方便地提取每一个部分。这里来解释一下，首先网络的预测值是一个二维张量 P ，其shape为 $[batch, 7 \times 7 \times 30]$ 。采用切片，那么 $P_{[:,0:7*7*20]}$ 就是类别概率部分，而 $P_{[:,7*7*20:7*7*(20+2)]}$ 是置信度部分，最后剩余部分 $P_{[:,7*7*(20+2):]}$ 是边界框的预测结果。这样，提取每个部分是非常方便的，这会方面后面的训练及预测时的计算。

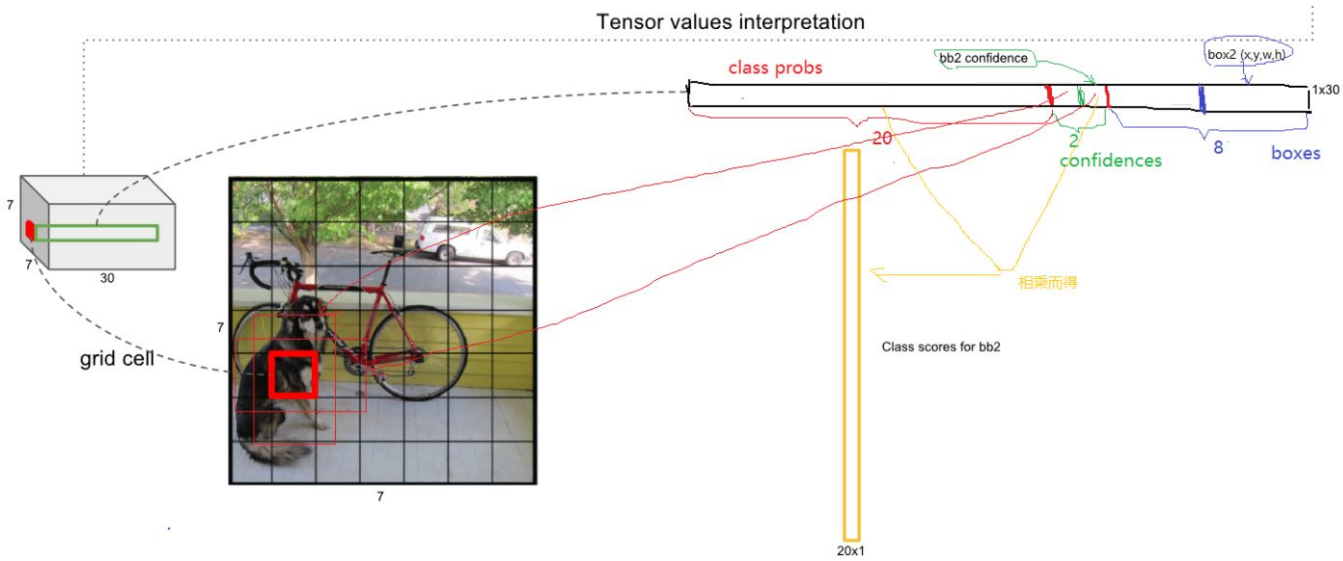


图9 预测张量的解析

网络训练

在训练之前，先在ImageNet上进行了预训练，其预训练的分类模型采用图8中前20个卷积层，然后添加一个average-pool层和全连接层。预训练之后，在预训练得到的20层卷积层之上加上随机初始化的4个卷积层和2个全连接层。由于检测任务一般需要更高清的图片，所以将网络的输入从224x224增加到了448x448。整个网络的流程如下图所示：

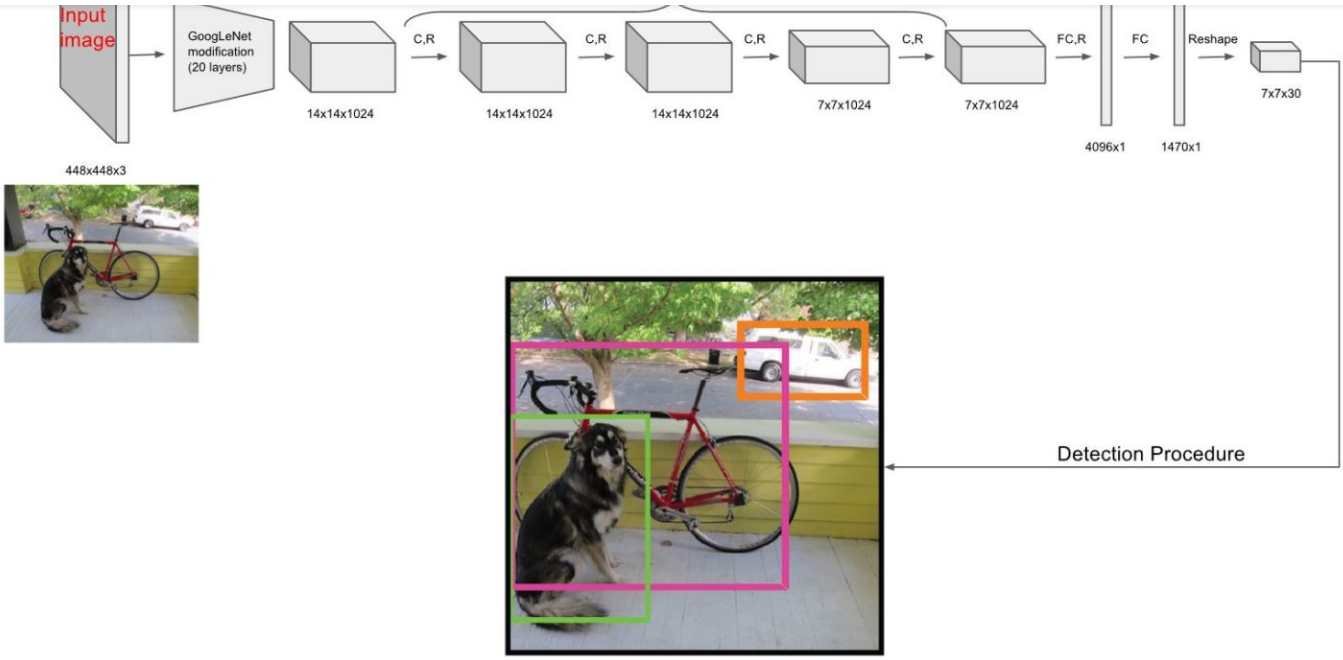


图10 Yolo网络流程

下面是训练损失函数的分析，Yolo算法将目标检测看成回归问题，所以采用的是均方差损失函数。但是对不同的部分采用了不同的权重值。首先区分定位误差和分类误差。对于定位误差，即边界框坐标预测误差，采用较大的权重 $\lambda_{coord}=5$ 。然后其区分不包含目标的边界框与含有目标的边界框的置信度，对于前者，采用较小的权重值 $\lambda_{noobj}=0.5$ 。其它权重值均设为1。然后采用均方误差，其同等对待大小不同的边界框，但是实际上较小的边界框的坐标误差应该要比较大的边界框要更敏感。为了保证这一点，将网络的边界框的宽与高预测改为对其平方根的预测，即预测值变为 $(x,y,\sqrt{w},\sqrt{h})$ 。

另外一点时，由于每个单元格预测多个边界框。但是其对应类别只有一个。那么在训练时，如果该单元格内确实存在目标，那么只选择与ground truth的IOU最大的那个边界框来负责预测该目标，而其它边界框认为不存在目标。这样设置的一个结果将会使一个单元格对应的边界框更加专业化，其可以分别适用不同大小，不同高宽比的目标，从而提升模型性能。大家可能会想如果一个单元格内存在多个目标怎么办，其实这时候Yolo算法就只能选择其中一个来训练，这也是Yolo算法的缺点之一。要注意的一点时，对于不存在对应目标的边界框，其误差项就是只有置信度，坐标项误差是没法计算的。而只有当一个单元格内确实存在目标时，才计算分类误差项，否则该项也是无法计算的。

综上讨论，最终的损失函数计算如下：

$$\lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \right]$$

$$+ \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[\left(\sqrt{w_i} - \sqrt{\hat{w}_i} \right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h}_i} \right)^2 \right]$$

$$+ \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left(C_i - \hat{C}_i \right)^2$$

$$+ \lambda_{\text{noobj}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{noobj}} \left(C_i - \hat{C}_i \right)^2$$

$$+ \sum_{i=0}^{S^2} \mathbb{1}_{i}^{\text{obj}} \sum_{c \in \text{classes}} \left(p_i(c) - \hat{p}_i(c) \right)^2$$

其中第一项是边界框中心坐标的误差项，$\mathbb{1}_{ij}^{obj}$ 指的是第 i 个单元格存在目标，且该单元格中的第 j 个边界框负责预测该目标。第二项是边界框的高与宽的误差项。第三项是包含目标的边界框的置信度误差项。第四项是不包含目标的边界框的置信度误差项。而最后一项是包含目标的单元格的分类误差项，$\mathbb{1}_{i}^{obj}$ 指的是第 i 个单元格存在目标。这里特别说一下置信度的target值 C_i ，如果是不存在目标，此时由于 $Pr(object)=0$，那么 $C_i=0$ 。如果存在目标，$Pr(object)=1$ ，此时需要确定 $\text{IOU}_{pred}^{truth}$ ，当然你希望最好的话，可以将IOU取1，这样 $C_i=1$ ，但是在YOLO实现中，使用了一个控制参数rescore（默认为1），当其为1时，IOU不是设置为1，而就是计算truth和pred之间的真实IOU。不过很多复现YOLO的项目还是取 $C_i=1$ ，这个差异应该不会太影响结果吧。

网络预测

在说明Yolo算法的预测过程之前，这里先介绍一下非极大值抑制算法（non maximum suppression, NMS），这个算法不单单是针对Yolo算法的，而是所有的检测算法中都会用到。NMS算法主要解决的是一个目标被多次检测的问题，如图11中人脸检测，可以看到人脸被多次检测，但是其实我们希望最后仅仅输出其中一个最好的预测框，比如对于美女，只想要红色那个检测结果。那么可以采用NMS算法来实现这样的效果：首先从所有的检测框中找到置信度最大的那个框，然后挨个计算其与剩余框的IOU，如果其值大于一定阈值（重合度过高），那么就将该框剔除；然后对剩余的检测框重复上述过程，直到处理完所有的检测框。Yolo预测过程也需要用到NMS算法

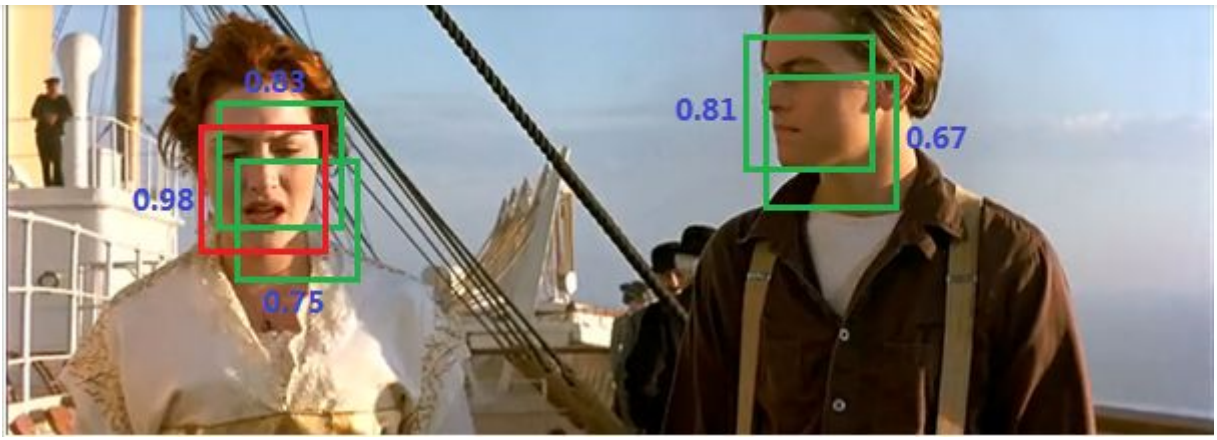


图11 NMS应用在人脸检测

下面就来分析Yolo的预测过程，这里我们不考虑batch，认为只是预测一张输入图片。根据前面的分析，最终的网络输出是 $7\times7\times30$ ，但是我们可以将其分割成三个部分：类别概率部分为 $[7,7,20]$ ，置信度部分为 $[7,7,2]$ ，而边界框部分为 $[7,7,2,4]$ （对于这部分不要忘记根据原始图片计算出其真实值）。然后将前两项相乘（矩阵 $[7,7,20]$ 乘以 $[7,7,2]$ 可以各补一个维度来完成 $[7,7,1,20]\times[7,7,2,1]$ ）可以得到类别置信度值为 $[7,7,2,20]$ ，这里总共预测了 $7*7*2=98$ 个边界框。

所有的准备数据已经得到了，那么我们先说第一种策略来得到检测框的结果，我认为这是最正常与自然的处理。首先，对于每个预测框根据类别置信度选取置信度最大的那个类别作为其预测标签，经过这层处理我们得到各个预测框的预测类别及对应的置信度值，其大小都是 $[7,7,2]$ 。一般情况下，会设置置信度阈值，就是将置信度小于该阈值的box过滤掉，所以经过这层处理，剩余的是置信度比较高的预测框。最后再对这些预测框使用NMS算法，最后留下来的就是检测结果。一个值得注意的点是NMS是对所有预测框一视同仁，还是区分每个类别，分别使用NMS。Ng在deeplearning.ai中讲应该区分每个类别分别使用NMS，但是看了很多实现，其实还是同等对待所有的框，我觉得可能是不同类别的目标出现在相同位置这种概率很低吧。

上面的预测方法应该非常简单明了，但是对于Yolo算法，其却采用了另外一个不同的处理思路（至少从C源码看是这样的），其区别就是先使用NMS，然后再确定各个box的类别。其基本过程如图12所示。对于98个boxes，首先将小于置信度阈值的值归0，然后分类别地对置信度值采用NMS，这里NMS处理结果不是剔除，而是将其置信度值归为0。最后才是确定各个box的类别，当其置信度值不为0时才做出检测结果输出。这个策略不是很直接，但是貌似Yolo源码就是这样做的。Yolo论文里面说NMS算法对Yolo的性能是影响很大的，所以可能这种策略对Yolo更好。但是我测试了普通的图片检测，两种策略结果是一样的。

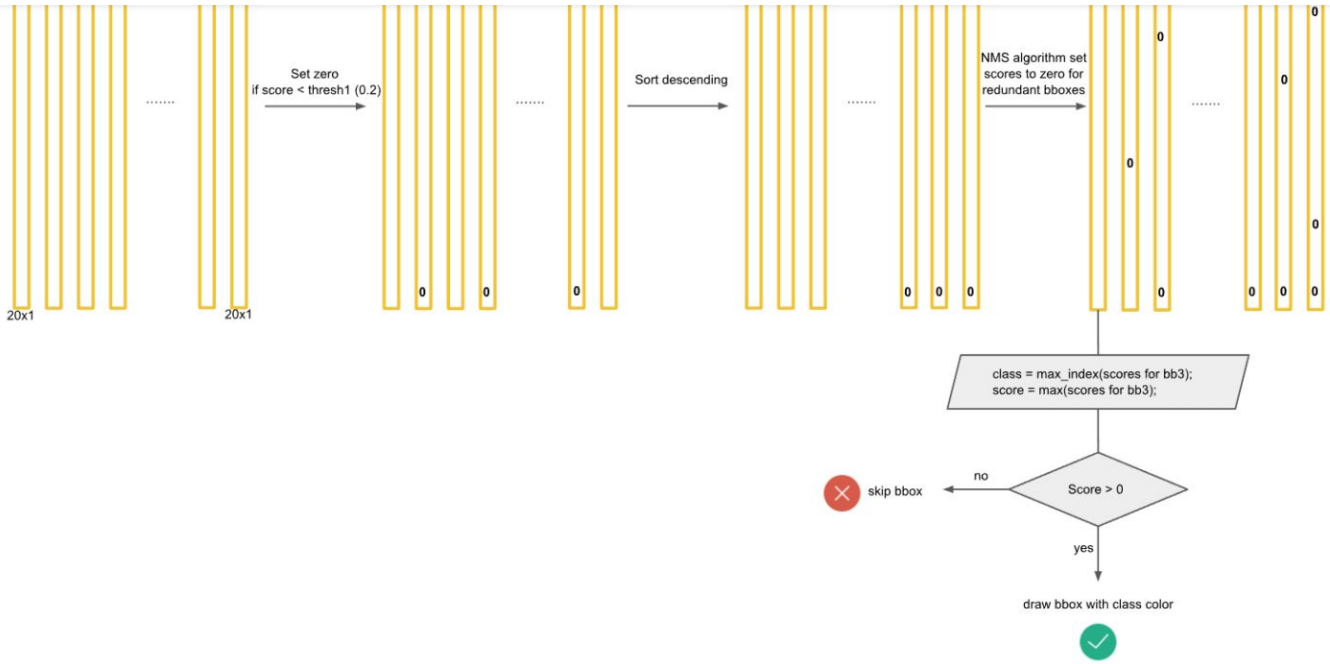


图12 Yolo的预测处理流程

算法性能分析

这里看一下Yolo算法在PASCAL VOC 2007数据集上的性能，这里Yolo与其它检测算法做了对比，包括DPM，R-CNN，Fast R-CNN以及Faster R-CNN。其对比结果如表1所示。与实时性检测方法DPM对比，可以看到Yolo算法可以在较高的mAP上达到较快的检测速度，其中Fast Yolo算法比快速DPM还快，而且mAP是远高于DPM。但是相比Faster R-CNN，Yolo的mAP稍低，但是速度更快。所以。Yolo算法算是在速度与准确度上做了折中。

100Hz DPM [31]	2007	16.0	100
30Hz DPM [31]	2007	26.1	30
Fast YOLO	2007+2012	52.7	**155**
YOLO	2007+2012	**63.4**	45
Less Than Real-Time			
Fastest DPM [38]	2007	30.4	15
R-CNN Minus R [20]	2007	53.5	6
Fast R-CNN [14]	2007+2012	70.0	0.5
Faster R-CNN VGG-16[28]	2007+2012	73.2	7
Faster R-CNN ZF [28]	2007+2012	62.1	18
YOLO VGG-16	2007+2012	66.4	21

表1 Yolo在PASCAL VOC 2007上与其他算法的对比

为了进一步分析Yolo算法，文章还做了误差分析，将预测结果按照分类与定位准确性分成以下5类：

- Correct：类别正确，IOU>0.5；（准确度）
- Localization：类别正确，0.1 < IOU<0.5（定位不准）；
- Similar：类别相似，IOU>0.1；
- Other：类别错误，IOU>0.1；
- Background：对任何目标其IOU<0.1。（误把背景当物体）

Yolo与Fast R-CNN的误差对比分析如下图所示：

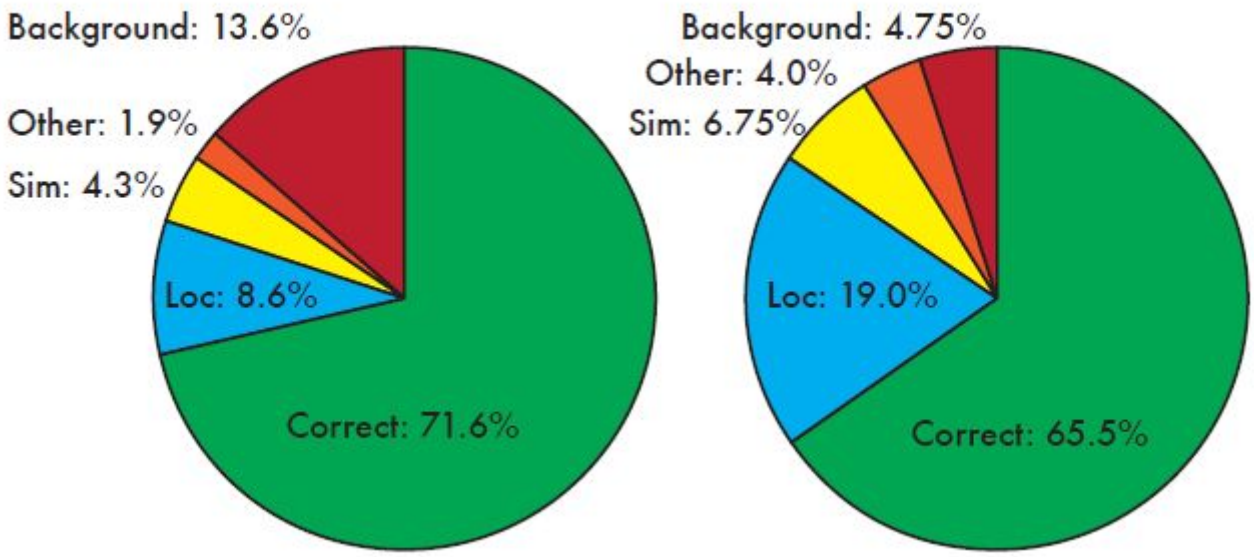


图13 Yolo与Fast R-CNN的误差对比分析

可以看到，Yolo的Correct的是低于Fast R-CNN。另外Yolo的Localization误差偏高，即定位不是很准确。但是Yolo的Background误差很低，说明其对背景的误判率较低。Yolo的那篇文章中还有更多性能对比，感兴趣可以看看。

现在来总结一下Yolo的优缺点。首先是优点，Yolo采用一个CNN网络来实现检测，是单管道策略，其训练与预测都是end-to-end，所以Yolo算法比较简洁且速度快。第二点由于Yolo是对整张图片做卷积，所以其在检测目标有更大的视野，它不容易对背景误判。其实我觉得全连接层也是对这个有贡献的，因为全连接起到了attention的作用。另外，Yolo的泛化能力强，在做迁移时，模型鲁棒性高。

最后不得不谈一下Yolo的缺点，首先Yolo各个单元格仅仅预测两个边界框，而且属于一个类别。对于小物体，Yolo的表现会不如人意。这方面的改进可以看SSD，其采用多尺度单元格。也可以看Faster R-CNN，其采用了anchor boxes。Yolo对于在物体的宽高比方面泛化率低，就是无法定位不寻常比例的物体。当然Yolo的定位不准确也是很大的问题。

算法的TF实现

Yolo的源码是用C实现的，但是好在Github上有很多开源的TF复现。这里我们参考gliese581gg的YOLO_tensorflow的实现来分析Yolo的Inference实现细节。我们的代码将构建一个end-to-end的Yolo的预测模型，利用的已经训练好的权重文件，你将可以用自然的图片去测试检测效果。

首先，我们定义Yolo的模型参数：

```
        self.verbose = verbose
        # detection params
        self.S = 7  # cell size
        self.B = 2  # boxes_per_cell
        self.classes = ["aeroplane", "bicycle", "bird", "boat", "bottle",
                        "bus", "car", "cat", "chair", "cow", "diningtable",
                        "dog", "horse", "motorbike", "person", "pottedplant",
                        "sheep", "sofa", "train","tvmonitor"]
        self.C = len(self.classes) # number of classes
        # offset for box center (top left point of each cell)
        self.x_offset = np.transpose(np.reshape(np.array([np.arange(self.S)]*se
                                                [self.B, self.S, self.S]), [1, 2,
        self.y_offset = np.transpose(self.x_offset, [1, 0, 2])

        self.threshold = 0.2  # confidence scores threhold
        self.iou_threshold = 0.4
        #  the maximum number of boxes to be selected by non max suppression
        self.max_output_size = 10

        self.sess = tf.Session()
        self._build_net()
        self._build_detector()
        self._load_weights(weights_file)
```

然后是我们模型的主体网络部分，这个网络将输出[batch,7*7*30]的张量:

```
def _build_net(self):
        """build the network"""
        if self.verbose:
            print("Start to build the network ...")
        self.images = tf.placeholder(tf.float32, [None, 448, 448, 3])
        net = self._conv_layer(self.images, 1, 64, 7, 2)
        net = self._maxpool_layer(net, 1, 2, 2)
        net = self._conv_layer(net, 2, 192, 3, 1)
        net = self._maxpool_layer(net, 2, 2, 2)
        net = self._conv_layer(net, 3, 128, 1, 1)
        net = self._conv_layer(net, 4, 256, 3, 1)
        net = self._conv_layer(net, 5, 256, 1, 1)
        net = self._conv_layer(net, 6, 512, 3, 1)
        net = self._maxpool_layer(net, 6, 2, 2)
```

```
        net = self._conv_layer(net, 11, 256, 1, 1)
        net = self._conv_layer(net, 12, 512, 3, 1)
        net = self._conv_layer(net, 13, 256, 1, 1)
        net = self._conv_layer(net, 14, 512, 3, 1)
        net = self._conv_layer(net, 15, 512, 1, 1)
        net = self._conv_layer(net, 16, 1024, 3, 1)
        net = self._maxpool_layer(net, 16, 2, 2)
        net = self._conv_layer(net, 17, 512, 1, 1)
        net = self._conv_layer(net, 18, 1024, 3, 1)
        net = self._conv_layer(net, 19, 512, 1, 1)
        net = self._conv_layer(net, 20, 1024, 3, 1)
        net = self._conv_layer(net, 21, 1024, 3, 1)
        net = self._conv_layer(net, 22, 1024, 3, 2)
        net = self._conv_layer(net, 23, 1024, 3, 1)
        net = self._conv_layer(net, 24, 1024, 3, 1)
        net = self._flatten(net)
        net = self._fc_layer(net, 25, 512, activation=leak_relu)
        net = self._fc_layer(net, 26, 4096, activation=leak_relu)
        net = self._fc_layer(net, 27, self.S*self.S*(self.C+5*self.B))
        self.predicts = net
```

接下来，我们要去解析网络的预测结果，这里采用了第一种预测策略，即判断预测框类别，再NMS，多亏了TF提供了NMS的函数tf.image.non_max_suppression，其实实现起来很简单，所有的细节前面已经交代了：

```
def _build_detector(self):
        """Interpret the net output and get the predicted boxes"""
        # the width and height of orignal image
        self.width = tf.placeholder(tf.float32, name="img_w")
        self.height = tf.placeholder(tf.float32, name="img_h")
        # get class prob, confidence, boxes from net output
        idx1 = self.S * self.S * self.C
        idx2 = idx1 + self.S * self.S * self.B
        # class prediction
        class_probs = tf.reshape(self.predicts[0, :idx1], [self.S, self.S, sel
        # confidence
        confs = tf.reshape(self.predicts[0, idx1:idx2], [self.S, self.S, self.B
        # boxes -> (x, y, w, h)
        boxes = tf.reshape(self.predicts[0, idx2:], [self.S, self.S, self.B, 4]
```

```
                (boxes[:, :, :, 1] + tf.constant(self.y_offset, dtype
                tf.square(boxes[:, :, :, 2]) * self.width,
                tf.square(boxes[:, :, :, 3]) * self.height], axis=3)

        # class-specific confidence scores [S, S, B, C]
        scores = tf.expand_dims(confs, -1) * tf.expand_dims(class_probs, 2)

        scores = tf.reshape(scores, [-1, self.C])  # [S*S*B, C]
        boxes = tf.reshape(boxes, [-1, 4])  # [S*S*B, 4]

        # find each box class, only select the max score
        box_classes = tf.argmax(scores, axis=1)
        box_class_scores = tf.reduce_max(scores, axis=1)

        # filter the boxes by the score threshold
        filter_mask = box_class_scores >= self.threshold
        scores = tf.boolean_mask(box_class_scores, filter_mask)
        boxes = tf.boolean_mask(boxes, filter_mask)
        box_classes = tf.boolean_mask(box_classes, filter_mask)

        # non max suppression (do not distinguish different classes)
        # ref: https://tensorflow.google.cn/api_docs/python/tf/image/non_max_su
        # box (x, y, w, h) -> box (x1, y1, x2, y2)
        _boxes = tf.stack([boxes[:, 0] - 0.5 * boxes[:, 2], boxes[:, 1] - 0.5 *
                           boxes[:, 0] + 0.5 * boxes[:, 2], boxes[:, 1] + 0.5 *
        nms_indices = tf.image.non_max_suppression(_boxes, scores,
                                                   self.max_output_size, self.
        self.scores = tf.gather(scores, nms_indices)
        self.boxes = tf.gather(boxes, nms_indices)
        self.box_classes = tf.gather(box_classes, nms_indices)
```

其他的就比较容易了，详细代码附在xiaohu2015/DeepLearning_tutorials上了，欢迎给点个赞，权重文件在这里下载。

最后就是愉快地测试你自己的图片了：

知乎 首发于 机器学习算法工程师

当然，如果你对训练过程感兴趣，你可以参考这里的实现thtrieu/darkflow，如果你看懂了预测过程的代码，这里也会很容易阅读。

小结

这篇长文详细介绍了Yolo算法的原理及实现，当然Yolo-v1还是有很多问题的，所以后续可以读读Yolo9000算法，看看其如何改进的。Ng说Yolo的paper是比较难读的，其实是很多实现细节，如果不看代码是很难理解的。所以，文章中如果有错误也可能是难免的，欢迎交流指正。

参考文献

1. You Only Look Once: Unified, Real-Time Object Detection.
2. Yolo官网.
3. Yolo的TF实现.
4. YOLO: You only look once (How it works).（注：很多实现细节，需要墙）
5. Ng的deeplearning.ai课程.

码字不易，欢迎给个赞！

欢迎交流与转载，文章会同步发布在公众号：机器学习算法全栈工程师(Jeemy110)

编辑于 2018-06-15