基于 Yolo 框架复现目标检测算法

张阳泽雨，余梦颖，邬昕昱
华中科技大学，武汉，中国 430070

摘要：通过计算机视觉进行目标检测能够很大程度上减少对人力资本的消耗，具有重要的现实意义，因此越来越多的对于物体的分类和目标检测需求出现，而目标检测相关的技术也不断涌现出来，被应用在各种场合，如机器人导航、无人驾驶、工业检测等等。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。我们主要研究 yolo 算法的原理，其使用在本数据集时的效果并且横向比较 yolo 与其他 CNN 算法的优劣，最后使用 PyTorch 从 0 实现一个 mini yolo。

关键字：目标检测；YOLO

1 提出方法

1.1 卷积神经网络 CNN

CNN 主要用来识别位移、缩放及其他形式扭曲不变性的二维图形。

1.1.1 CNN 的神经网络层

卷积层：卷积神经网络的核心所在。通过实现局部感知和权值共享等系列的设计理念，可以对高维输入数据实施降维处理，以及实现自动提取原始数据的核心特征。

激活层：其作用是将前一层的线性输出，通过非线性激活函数处理，从而可模拟任意函数，进而增强网络的表征能力。在深度学习领域，ReLU 是目前使用较多的激活函数，原因是它收敛更快，且不会产生梯度消失问题。

池化层：也称亚采样层。它利用局部相关性，使采样较少数据规模的同时保留了有用信息。巧妙的采样还具备局部线性转换不变性，从而增强卷积神经网络的泛化处理能力。

全连接层：这个网络层相当于传统的多层感知机（Multi-Layer Perceptron，简称 MLP）。卷积-激活-池化是一个基本的处理栈，通过多个前栈处理之后，待处理的数据特性已有了显著变化：一方面，输入数据的维度已下降到可用“全连接”网络来处理了；另一方面，此时全连接层的输入数据已不再是混杂着全部传递，而是经过反复提纯过的结果，因此最后输出的结果要可控得高。

1.1.2 CNN 相比其他神经网络的优势

传统的 BP 神经网络与 DNN 深度神经网络不同层神经元的连接方式是“全连接”，也就是这一次层的一个神经元的输入，会接受上一次每一个神经元的输出，这种方式即为“全连接神经网络”。这样的连接方式有一个的缺点——因为权值与偏置等参数量大，导致训练收敛十分缓慢。特别是对于图像这样的训练数据，高达数以百万的像素，理论上虽然可以收敛，但可能得等上很久才有结果，并且它的泛化性也会变差。而 CNN 有如下三个优点：

1. 局部感受广

一张图像实际上并不需要让每个神经元都接受整个图片的信息，而是让不同区域的神经元对应一整张图片的不同局部，最后只要再把局部信息整合到一起就可以了。这样就相当于在神经元最初的输入层实现了一次降维。

2. 卷积层的权值共享

卷积神经网络的最重要之处，就是解决了全连接神经网络权值参数太多，而卷积神经网络的卷积层，不同神经元的权值是共享的，这使得整个神经网络的参数大大减小，提高了整个网络的训练性能。

3. 池化层的使用

池化层可以看作是对图像的一次“有损压缩”，在实际的训练中，我们并不需要对图像中的每一个细节都进行特征提取和训练，而池化的作用就是更进一步的信息抽象和特征提取，也就减小了数据的处理量。

1.2 Yolo

Yolo 算法采用一个单独的 CNN 模型实现 end-to-end 的目标检测，它的检测网络包括 24 个卷积层和 2 个全连接层，其中，卷积层用来提取图像特征，全连接层用来预测图像位置和类别概率值。

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哪些中心点落在该格子内的目标，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）。置信度包含两个方面，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$\mathrm{P_r(object)}$，当该边界框是背景时（即不包含目标），此时$\mathrm{P_r(object)} = 0$。而当该边界框包含目标时，$\mathrm{P_r(object)} = 1$。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$\mathrm{IOU}_{\mathrm{pred}}^{\mathrm{truth}}$。因此置信度可以定义为$\mathrm{P_r(object)}*\ \mathrm{IOU}_{\mathrm{pred}}^{\mathrm{truth}}$。很多人可能将 Yolo 的置信度看成边界框是否含有目标的概率，但是其实它是两个因子的乘积，预测框的准确度也反映在里面。边界框的大小与位

置可以用 4 个值来表征：(x, y, w, h)，其中(x, y)是边界框的中心坐标，而 w 和 h 是边界框的宽与高。同时，中心坐标的预测值(x, y)是相对于每个单元格左上角坐标点的偏移值，并且单位是相对于单元格大小的。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，因此理论上 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。

而对于分类问题，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的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$P_r(class_i|object)$。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）：

$$P_r(class_i|object) * P_r(object) * IOU_{pred}^{truth} = P_r(class_i) * IOU_{pred}^{truth}。$$

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。一般会根据类别置信度来过滤网络的预测框。

1.2.1 Yolo 网络设计

Yolo 网络从初试卷积层中提取图像特征，在全连接层预测输出概率和坐标。Yolo 由 24 个卷积层和 2 个全连接层组成，其中卷积层的卷积核主要有两种，一种是 3*3，另外一种是 1*1。最后一个全连接层即 Yolo 网络的输出，长度为 S*S*(B*5+C)=7*7*30，网络的架构如图所示。

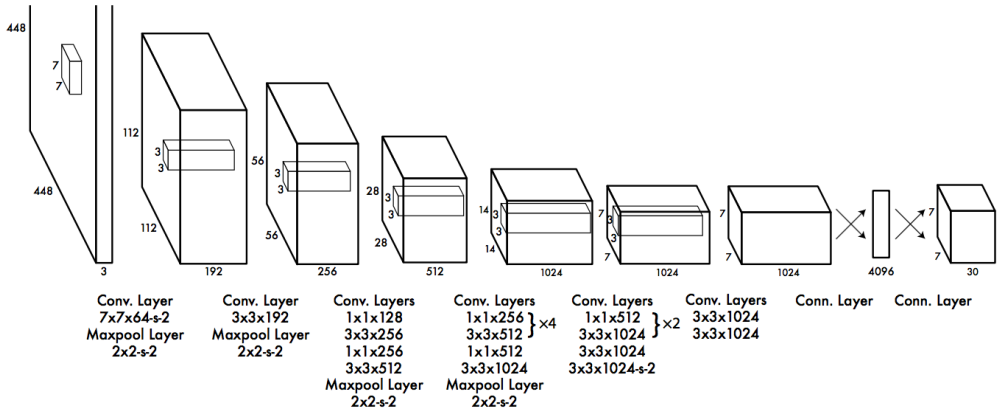

图 1. Yolo 网络架构

1.2.2 Yolo 和其他算法的对比

从 Yolo 算法在 PASCAL VOC 2007 数据集上与其他算法的性能对比，可以看出 Yolo 算法可以在较高的 mAP 上达到较快的检测速度，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。

表 1. Yolo 在 PASCAL VOC 2007 上与其他算法的对比

Real-Time Detectors	Train	mAP	FPS
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

Yolo 与 Fast R-CNN 的误差对比分析如下图所示，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。

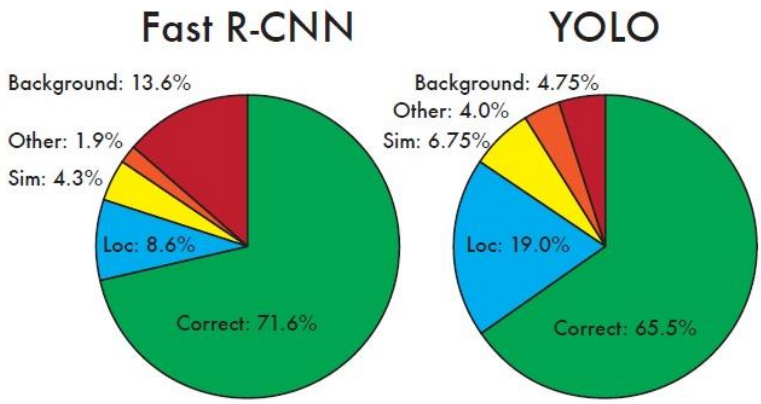

图 2. Yolo 与 Fast R-CNN 的误差对比分析

1.2.3 Yolo 的优缺点

综上，Yolo 具有如下优点：

1. 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。

2. YOLO 采用全图信息来进行预测。与滑动窗口方法和 region proposal-based 方法不同，YOLO 在训练和预测过程中可以利用全图信息。Fast R-CNN 检测方法会错误的将背景中的斑块检测为目标，原因在于 Fast R-CNN 在检测中无法看到全局图像。相对于 Fast R-CNN，YOLO 背景预测错误率低一半。
3. Yolo 的泛化能力强，在做迁移时，模型鲁棒性高。

而相比 RCNN 系列物体检测方法，Yolo 具有以下缺点：

1. 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。
2. Yolo 识别物体位置精准性差。
3. 召回率低。

2 实验过程与结果

2.1 利用 yolo_v3 官方框架训练模型并测试准确率

2.1.1 数据集

我们所使用的数据集来自于老师提供的 ImageNet VID 的一部分，包括了 bird, car, dog, lizard, turtle 五种分类，每个分类分别有 180 张分辨率为 128*128 的图片，其中 150 张是训练集，30 张作为测试集。

由于我们使用 darknet 框架进行训练，而 darknet 提供的训练测试模型的接口以及一些脚本适用于 VOC 数据集，因此我们将已有的标定数据整理为所需的格式。第一步，我们将所有图片放置在 JPEGImages 文件夹下，重命名为 VOC 要求的格式，其中共有 900 张图片——前 750 张为训练图片，后 150 张为测试图片。第二步，利用现有的 txt 的标注信息（xmin, xmax, ymin, ymax）使用脚本生成对应的 labels 下的 txt 文件，格式为<object-class> <center-x> <center-y> <width 百分比> <height 百分比>；最后，利用脚本生成 train.txt 和 test.txt 文件，用于训练和测试，这两个 txt 文件均用来保存图片的路径，train 用来保存用于训练的图片的路径。

2.1.2 使用数据集训练获得新的权重文件

2.1.2.1 修改相关配置文件以适应自己的数据集

修改 cfg/voc.data 中的类别为我们提供的类别及顺序，根据类别数量修改 cfg/yolov3-voc.cfg 中的 filter 数目，修改 data/voc.names 中训练集和测试集的路径。

2.1.2.2 从一个初始权重开始训练

在神经网络的训练之前我们一般会给定一个初始值，如一个确定的初始值或是由参数随机

生成的初始值，这里我们使用 darknet53.conv.74 权重文件。

2.1.3 网络架构的理解

在 yolov3.cfg 文件中编写了网络架构的配置。

```
[convolutional]
batch_normalize=1
filters=32
size=3
stride=1
pad=1
activation=leaky
```

图 3. Convolutional 层参数配置

[convolutiontal]代码行以下为卷积层的参数；

```
[shortcut]
from=-3
activation=linear
```

图 4. Shortcut 层参数配置

[shortcut]代码行以下为跳过连接的参数，跳过连接与残差网络中使用的结构相似，参数 from 为-3，表示捷径层的输出会通过将之前层的和之前第三个层的输出的特征图与模块的输入相加而得出；

```
[upsample]
stride=2
```

图 5. Upsample 层参数配置

[upsample]代码行以下为上采样的参数，通过参数 stride 在前面层级中双线性上采样特征图；

```
[route]
layers = -1, 61
```

图 6. Route 层参数配置

[route]代码行以下为路由层的参数，它的参数 layers 有一个或两个值，当只有一个值的时候，它输出这一层通过该值索引的特征图，比如将 layers 设置为-3，层级将输出路由层之前第三个层的特征图，当层级有两个值时，他将返回有这两个值索引的拼接特征图，比如我们将 layers 设置为-1 和 64，该层级将输出从前第 1 层到第 64 层的特征图，并将它们按深度拼接；

```
[yolo]
mask = 3,4,5
anchors = 10,13,  16,30,  33,23,  30,61,  62,45,  59,119,  116,90,  156,198,  373,326
classes=5
num=9
jitter=.3
ignore_thresh = .5
truth_thresh = 1
random=0
```

图 7. Yolo 层参数配置

[yolo]代码行以下为 yolo 层的参数，yolo 层级对应于上文所描述的检测层级，参数 anchors 定义了九组锚点，它们是只由 mask 标签使用的属性所索引的锚点。Mask 的值为 3，4，5，表示了第三个、第四个和第五个使用的锚点，而掩码表示检测层中的每一个单元预测三个框。

```
[net]
# Testing
# batch=1
# subdivisions=1
# Training
batch=32
subdivisions=8
width=416
height=416
channels=3
momentum=0.9
decay=0.0005
angle=0
saturation = 1.5
exposure = 1.5
hue=.1

learning_rate=0.001
burn_in=1000
max_batches = 10
policy=steps
steps=40000,45000
scales=.1,.1
```

图 8. Net 相关参数

[net]代码行以下的参数是用来描述网络输入和训练参数的相关信息，它为我们提供了网络输入大小等信息，可以用于调整前向传播中的锚点。

这里根据数据集的实际情况、将所有 yolo 层的 classes 参数改为 5、将所有 convolutional 层的参数改为 30(30=(classesNum+5)*3)。在 net 层、由于我们数据集不大且考虑到时间因素，我们将 batch 参数设置为 32、subdivision 设置为 8、max_batches 设置为 5000 次；使用此参数大约仅需要训练 2 小时即可。此外、考虑到显存占用问题、我们还将 yolo 层的 random 参数设置为 0、这是因为我们的数据集中物体属于较大物体、也没有必要使用多尺度训练。

在实际训练的过程中，我们还遇到了显存不够的问题。先前用 CPU 模式训练时、由于速度过慢、导致生成的 weight 文件不尽人意。当时的测试图片如下：

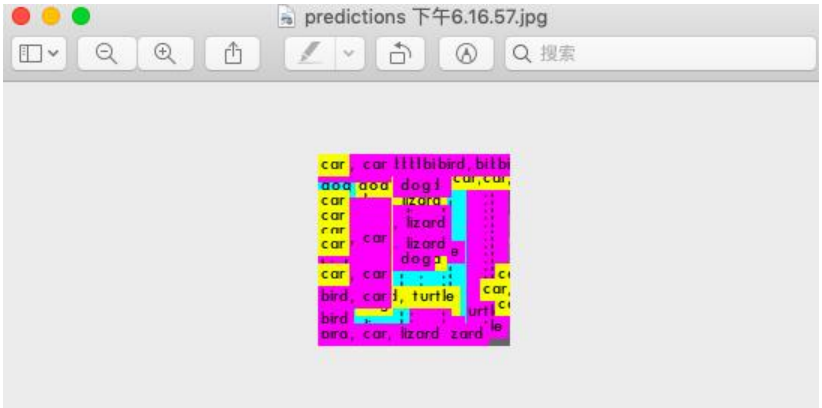

图 9. 失败的测试图片

由于迭代次数过少、训练的图片数量十分有限、故预测的结果也和真实情况有天壤之别。不得已之下我们只好采用 GPU 的来训练、修改 makefile 文件之后重新编译出 darknet 文件。但是由于 docker 服务器上的第一块显存一直处于占用状态，我们每次 train 的时候都会出现显存不足的问题。后经过修改 darknet 训练部分的函数将 gpu 指定为使用第二块，解决了这个问题。

2.1.3.1 输出日志分析

首先解释一下训练日志中各参数的意义。Region Avg IOU：平均 IOU，代表预测的 bounding box 和 ground truth 的交集与并集之比，期望该值趋近于 1；Class：标注物体的概率，期望该值趋近于 1；Obj：越接近 1 越好；No Obj：期望该值越来越小，但不为零；Avg Recall：当前模型在所有 subdivision 图片中检测出的正样本与实际的正样本的比值；

在我们的输出日志中出现了 nan，是因为我们的训练集中目标物体较大，而当 batch_size 较小的时候，栅格中检测不到物体，就会出现 nan。

将我们日志中的 loss 和 Avg IOU 进行可视化，观察分析曲线。

从损失变化曲线可以看出，模型在 2000 次迭代后损失下降速度非常慢，结合 log 文件发现，学习率在 2000 次迭代时会下降到非常小的程度，导致损失下降速度降低。

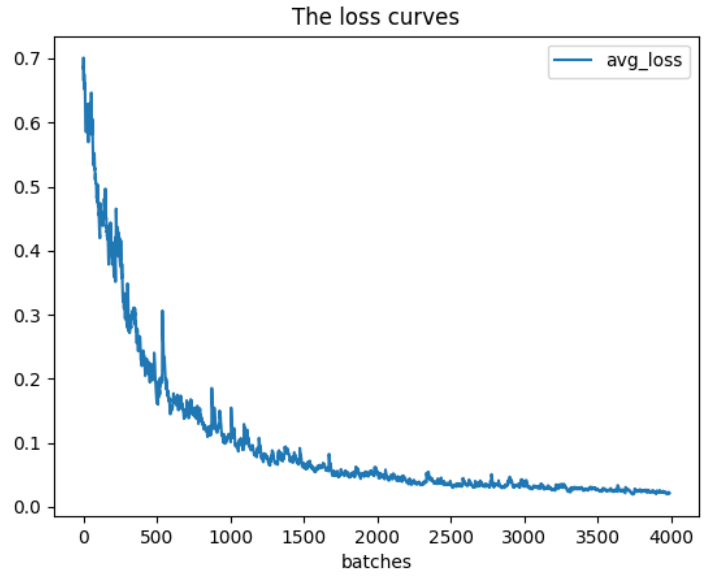

图 10. 损失变化曲线

由于我们设置了 5000 次迭代次数，一次迭代会分 8 次送入网络训练，每一次的训练会在三个维度上进行平均 IOU 的估算，但是由于我们的目标物体较大，几乎只能在 82 这个维度上检测到目标，以及极少数可以在 94 维度下检测出来，所以最终平均 IOU 个数应该是（8*5000 + 在 94 维度下检测出来的物体数量）> 40000，符合我们的输出曲线图。观察 IOU 变化趋势，可以看到 IOU 有一些抖动，但是大趋势是从一个较小的值不断增加到一个趋于 1 的值，最后我们的平均 IOU 维持在 0.88 左右的水平。通过训练结果可以看出，该模型可以说是非常不错的、在训练检测的过程中分类准确率已经达到趋近于 1 的水平，而 AvgIOU 也是远高于要求的 0.5。

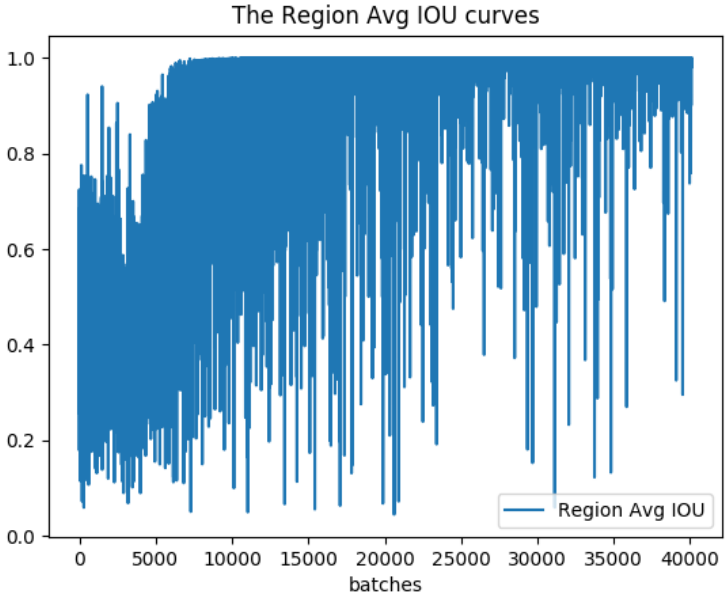

图 11. IOU 变化曲线

```
Region 82 Avg IOU: 0.880893, Class: 0.999031, Obj: 0.999764, No Obj: 0.006567, .5R
Region 94 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000001, .5R: -nan, .75R
Region 106 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000000, .5R: -nan, .75
Region 82 Avg IOU: 0.900560, Class: 0.999182, Obj: 0.999936, No Obj: 0.007261, .5R
Region 94 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000001, .5R: -nan, .75R
Region 106 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000000, .5R: -nan, .75
Region 82 Avg IOU: 0.886796, Class: 0.999043, Obj: 0.998048, No Obj: 0.006567, .5R
Region 94 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000010, .5R: -nan, .75R
Region 106 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000000, .5R: -nan, .75
```

图 12. 第 4989 次迭代时的输出日志

2.1.3.2 测试结果分析

利用训练出来的权重模型，我们可以对图片进行测试。

由于 darknet 只提供对单张图片检测的接口、我们需要修改 test 检测的函数来使它可以批量遍历我们的测试集。通过修改 darknet.c、detector.c 以及 image.c 文件的相关函数、我们可以批量对测试集中的图片进行测试、将分类和定位结果输出到一个 test.txt 文件中并将标注后的图片放入 testData/out 文件夹下。

之后通过编写脚本对 test.txt 文件数据进行分析、统计出总的分类准确率/IOU 和各个类的分类准确率/IOU。

以下为最终输出结果:

Name	Classification accuracy	Positioning accuracy
[Bird]	0.8	0.8616666666666667
[Car]	0.9666666666666667	0.9646666666666666
[Dog]	0.9333333333333333	0.8876666666666666
[Lizard]	1	0.8783333333333333
[Turtle]	1	1
[All]	0.94	0.9184666666666668

图 13. 测试集目标检测结果

通过结果可以看出。在分类方面，我们的模型测试结果达到了 94%，而在定位上面更是超过了 90%，远远高于 50%，可以说是一个非常不错的结果。在单类上，我们发现对于蜥蜴和海龟的分类异常准确，应该是这两类动物有较明显的特征，因此易于区分。

此外，通过对 txt 文件以及预测后输出图像的查看，发现有极少部分图片并没有被检测出物体。这些图片特征大致分为三种：一种是和环境融合的较好、一种是图片过于模糊(人眼几乎都无法识别)、还有就是信息不全(比如只拍到一个车轮和极少部分车身)。由于我们的训练集有限、训练迭代次数有限，故无法做到对这类图片的 object detection。此外，值得高兴的是，darknet 框架实现的 yolo 天生支持多物体检测、但是在我们的测试结果中并有对任何一张图检测出来有多个结果。

以下是三种无法识别类型的图片：

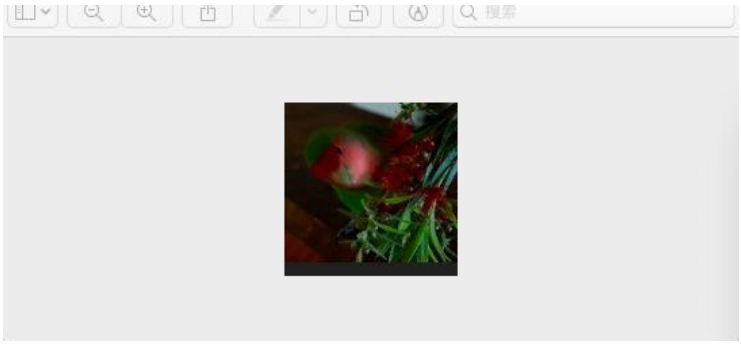

图 14. 和环境融合的鸟

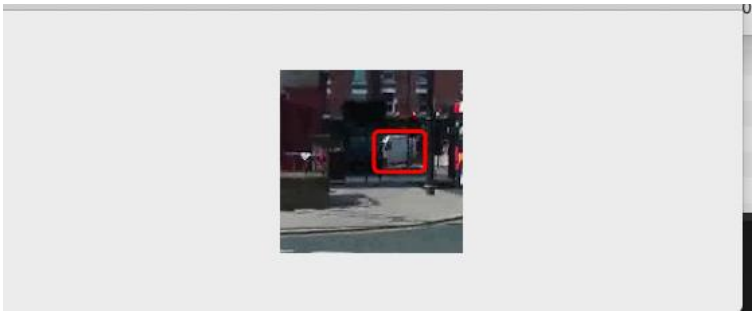

图 15. 过于模糊的车

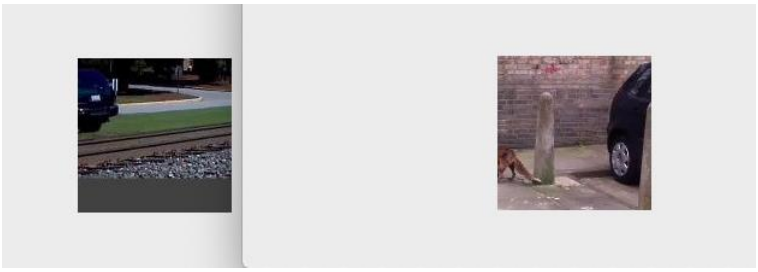

图 16. 信息不全的车和狗

2.2 利用 pytorch 重写 yolo_v2 进行训练并测试准确率

在完成了 yolo_v3 官方框架训练自己的测试集，观察日志并分析结果后，我们决定更深入地理解 yolo，于是自己动手用 PyTorch 参考网上的教程来实现 yolo。

本次实现环境基于 python2.7 和 PyTorch0.4.0，改写的是 yolo v2 版本。

2.2.1.1 创建网络层级

我们使用 PyTorch 的 nn.Module, nn.Sequential 和 nn.parameter 等来创建整个模型的基本构件块，和使用 yolo 的官方框架一样，我们还是使用 cfg 文件作为网络参数配置文件，自己编写 python 文件来解析 cfg 文件，将每一个块存储为一个词典，每个块中的属性和值以键值对的形式存储在词典中，最后将所有块存储在一个数组中，完成解析。

如前一个实验中提到的，cfg 文件中一共定义了 5 种类型的层——卷积层、shortcut 层、上采样层、路由层和 yolo 层，其中卷积层和上采样层可以通过 PyTorch 提供的预置层来创建，其

他的层级我们通过 nn.Module 类来自己编写。

对于路由层和 shortcut 层，我们提取关于层属性的值，将其表示为一个整数，然后实例化一个 EmptyLayer 层，作为初始化的 layers，然后将 forward 函数中的特征图拼接起来并向前传送，在路由层之后的卷积层会把它的卷积核应用到之前层的特征图上。对于 yolo 层，我们定义一个 DetectionLayer 来保存用于检测边界框的锚点。

2.2.1.2 实现网络的前向传播

为了计算输出以及尽早处理的方式转换输出检测特征图，我们需要实现 forward 函数。

由于路由层和捷径层需要之前层的输出特征图，因此我们需要缓存每个层的输出特征图，以层的索引和特征图作为键值对。

Yolo 层的输出是一个卷积特征图，包含特征图的边界框属性，边界框属性由彼此堆叠的单元格预测得出，并且检测是在三个维度上进行的，除了维度，其他的检测过程是一样的，因此我们编写了 predict_transform 函数进行复用。这个函数把检测特征图转换成二维张量，张量的每一行对应边界框的属性。根据 yolo 的原理，我们使用 sigmoid 函数进行重心坐标预测，并且用 object 分数表示目标在边界框内的概率。因此我们对(x, y)坐标和 objectness 分数执行 Sigmoid 函数操作，将网格偏移添加到中心坐标预测中，将锚点应用到边界框维度中，将 sigmoid 激活函数应用到类别分数中。

2.2.1.3 遇到的问题及解决方法

i. Python 环境问题

本实验需要的环境应该是 python2.7 和 pytorch0.4.0，但我们所使用的 docker 中已经安装了 pytorch1.0.1，这导致代码一开始无法跑通，因此我们利用 virtualenv 在文件路径下构造了 python2.7 和 pytorch0.4.0 的虚拟环境。

ii. GPU 使用问题

因为使用的 docker 为全班同学共用，我们通过 nvidia-smi 查看 GPU 占用率，发现 GPU0 一直处于占用状态，而 GPU1 为闲置状态，直接运行代码会默认使用第 0 块 GPU 导致资源不足的报错，因此我们通过指定使用 GPU1 解决了这个问题。

iii. 共享内存问题

报错提示 dataloader 进程被杀死，共享内存不足，通过 df -h 命令查看发现我们使用的 docker 共享内存只有 64M，在网上查找资料发现大家基本将共享内存设置为 7G 才能正常运行，但是 docker 上有许多同学正在跑的进程，不方便修改参数并且重启，因此我们直接将 num_worker 修改为 0，使程序单进程运行，成功解决了这个问题。

2.2.1.4 实验结果

训练过程中我们解决了在具体算 bbox region 等相关参数时，会有浮点数与 python 本身方法需要传递整数等冲突，对 PyTorch 支持广播的概念理解有误，以至于在实际进行测试时总是会出现维度不匹配的问题，后来索性用 reshape 方法解决，但最后由于无法解决显存不足的问题，这个实验的训练过程没有完成。

因此我们拿实验一训练出来的模型来测试 PyTorch 编写的 yolo v2 的测试部分。在解决掉资源不足等问题之后，代码顺利跑通，但是我们却发现测试图片上的目标完全不能被检测到，因此我们重新复查了一遍测试逻辑，并且打印了寻找目标前的图片来定位错误，发现图片在读入后的处理过程中出现了计算错误，导致所有值都是-inf，即负无穷，因此导致检测不到任何物体，猜测是参数设置不同以至于图片的基本处理过程有问题。

由于对框架的理解不够深入，确定 draw box region 和优化等处理的的时候对维度等细节问题不能有很明确的认知，导致在实际测试框架的时候，对于传参的理解等问题上，小组成员之间常常有较大分歧，并且在对中间结果的分析时经常走入误区，导致最终没有拿到成功的实验结果。