基于 R-CNN 的物体目标检测

曾耀沛，钟帅，齐嘉程

摘要：随着机器学习的不断发展，计算机视觉所涉及到的范围也越来越广，如图片分类，图片生成，物体检测等，而物体检测又可以在很大分类中进行检测。本文旨在针对物体检测当中的一个方面进行一个基本的研究。对于物体检测来说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。本文旨在研究 Yolo 算法，学习它们的原理，最后尝试仅使用深度学习框架实现物体目标检测。

1 提出方法

1.1 卷积神经网络（CNN）基本原理

CNN，卷积神经网络依旧是层级网络，但是对层的功能和形式做了变化，CNN 由输入和输出层以及多个隐藏层组成。

1.1.2 CNN 的基本模块

数据输入层：该层要做的处理主要是对原始图像数据进行预处理，其中包括：去均值、归一化等等。

卷积计算层：这一层就是卷积神经网络最重要的一个层次，也是“卷积神经网络”的名字来源。层的参数由一组可学习的滤波器（filter）或内核（kernels）组成，它们具有小的 Receptive Field，延伸到输入容积的整个深度。

激励层：把卷积层输出结果做非线性映射。CNN 采用的激励函数一般为 ReLU(The

Rectified Linear Unit/修正线性单元)，它的特点是收敛快，求梯度简单。

池化层：池化层夹在连续的卷积层中间， 用于压缩数据和参数的量，减小过拟合。简而言之，如果输入是图像的话，那么池化层的最主要作用就是在保留有用信息的同时压缩图像。

全连接层：这个层就是一个常规的神经网络，它的作用是对经过多次卷积层和多次池化层所得出来的高级特征进行全连接（全连接就是常规神经网络的性质），算出最后的预测值。

1.1.2 CNN 卷积神经网络的特点

此单元主要讨论 CNN 相比与传统的神经网络的不同之处，CNN 主要有三大特色，分别是**局部感知**、**权重共享**和**多卷积核。**

1. 局部感知

局部感知就是上文所提及的感受野，一般认为人对外界的认知是从局部到全局的，而图像的空间联系也是局部的像素联系较为紧密，而距离较远的像素相关性则较弱。因而，每个神经元其实没有必要对全局图像进行感知，只需要对局部进行感知，然后在更高层将局部的信息综合起来就得到了全局的信息。

2. 权重共享

传统的神经网络的参数量是非常巨大的，比如 1000X1000 像素的图片，映射到和自己相同的大小，需要（1000X1000）的平方，也就是 10 的 12 次方，参数量太大了，而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. 多卷积核

一种卷积核代表的是一种特征，为获得更多不同的特征集合，卷积层会有多个卷积核，生成不同的特征，这也是为什么卷积后的图片的高，每一个图片代表不同的特征。

1.2 Yolo 算法的基本原理

Yolo（You Only Look Once）直接采用 regression（回归）的方法进行坐标框的检测以及分类，使用一个 end-to-end 的简单网络，直接实现坐标回归与分类。基础模型中共有 24 个卷积层，后面接 2 个全连接层，如上文所述，卷积层用来提取图像的特征，而全连接层根据特征来预测物体位置和物体类别。

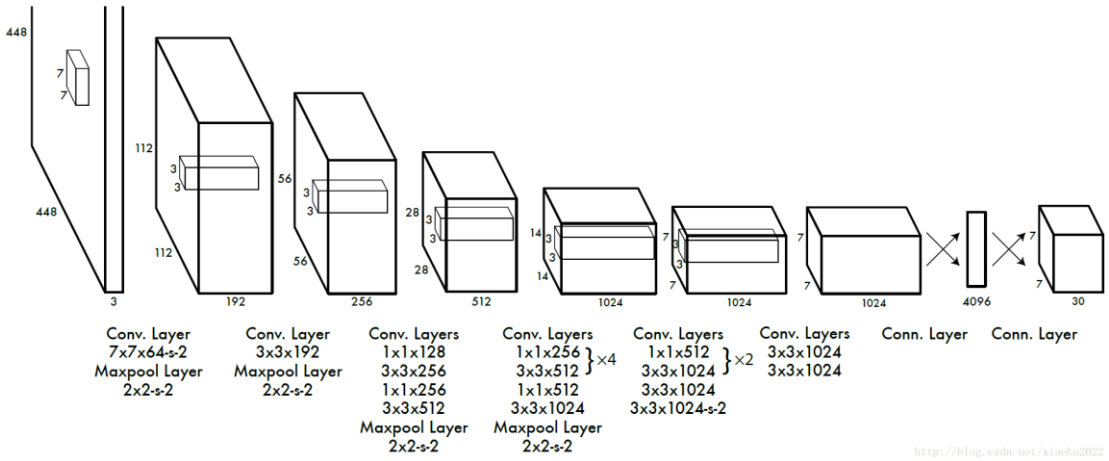

图 1 Yolo 网络结构

1.2.1 Yolo 算法设计思想

具体来说，Yolo 的 CNN 网络将输入的图片分割成 S×S 网格，使用每个单元格去检测那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）。置信度包含了两个方面，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$P_r(object)$，当该边界框是背景时（即不包含目标），此时$P_r(object) = 0$。而当该边界框包含目标时，$P_r(object) = 1$。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IOU_{pred}^{truth}。因此置信度可以定义为$P_r(object) * IOU_{pred}^{truth}$。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)(x,y,w,h)，其中(x,y)(x,y)是边界框的中心坐标，而 w 和 h 是边界框的宽与高。中心坐标的预测值(x,y)(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，与其成正比关系。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，在这样的条件下理论上 4 个元素的大小应该在[0,1][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)(x,y,w,h,c)，其中前 4 个表征边界框的大小与位置，而最后一个值就是置信度。

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，其实只预测一组类别概率值，这也是 Yolo 算法的一个缺点，因为效率不够高，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。在，我们可以计算出各个边界框类别置信度:

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

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。

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×S 网格，那么最终预测值为 S×S×(B*5+C)大小的张量。对于本次的课设提供的数据，其共有 5 个类别，假设使用 S=7,B=2，那么最终的预测结果就是 7×7×15 大小的张量。

1.2.2Yolo 算法性能分析

根据收集到的资料，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 算法可以在保证较高的 mAP（平均准确率）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。由此我们可以发现，Yolo 算法牺牲了一定的准确率来保证更快的训练速度和检测速度。

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

表 1 Yolo **在** PASCAL VOC 2007 **上与其他算法的**对比

下面将更直接地比较一下两种算法在检测物体上正确率的表现，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，说明 Yolo 算法的定位不是很准确。但是 Yolo 的 Background 误差很低，说明 Yolo 对背景的误判率较低。

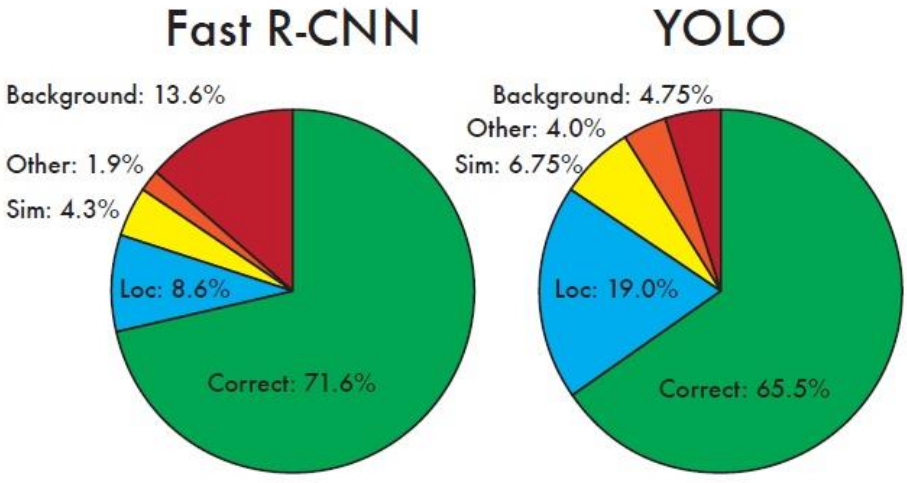

图 2 Yolo **与** Fast R-CNNde 误差对比分析

1.2.3 Yolo 的优势和缺点

综上，Yolo 在物体检测上具有如下的优势：

1. Yolo 仅采用一个 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 不同于 sliding window 以及 region proposal 方法在训练和检测的时候只能看到局部图像，而 Yolo 可以看到全局图像,(这也是图 2 中展示的 Fast R-CNN 的背景检测错误的概率要高于 Yolo 的原因)。Yolo 可以看到图像全局特征，得益于后面的两个全连接层，Yolo 通过 reshape 将全连接的结果对应回原图,使得其可以看到全局的特征
3. Yolo 的泛化能力强，在做迁移时，模型鲁棒性高。

而相比 R-CNN 系列物体检测算法，Yolo 则具有以下缺点：

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 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。
3. 召回率低。

2 具体实现

2.1 Yolo 框架的实现

本次实验我们选用的 Yolo 框架,在实现上我们首先使用官网上 yolov3-darknet 来处理，并在要求的数据集上得到了较为满意的结果，之后我们尝试使用 python 的第三方库来重现 yolo 框架。调查之后我们选择了 keras-yolov3 + pytorch-yolov3 的方式，但这种方式在实现的过程中遇到了一些问题，最终没能成功解决。

2.1.1 Yolo_v3 框架概述

Yolo_v3 作为 yolo 系列目前较新的算法，对之前的算法既有保留又有改进。先分析一下 Yolo_v3 上保留的东西：

1. Yolo_v3 算法依然通过划分单元格来做检测，只是划分的数量不一样。
2. 依旧采用"leaky ReLU"作为激活函数。
3. 端到端进行训练。一个 loss function 搞定训练，只需关注输入端和输出端。
4. 从 Yolo_v2 开始，Yolo 就用 batch normalization（批量标准化）作为正则化、加速收敛和避免过拟合的方法，把 BN 层和激活层接到每一层卷积层之后。
5. 多尺度训练。在速度和准确率之间做平衡。想速度快点，可以牺牲准确率；想准确率高点儿，可以牺牲一点速度。

而 Yolo_v3 这一代的提升很大一部分决定于 backbone 网络的提升，从 v2 的 darknet-19 到 v3 的 darknet-53。Yolo_v3 还提供替换 backbone——tiny darknet。这为使用者提供了很灵活的选择空间，想要检测结果更准确，backbone 可以用 Darknet-53，想要更快的检测速度，可以用 tiny-darknet。

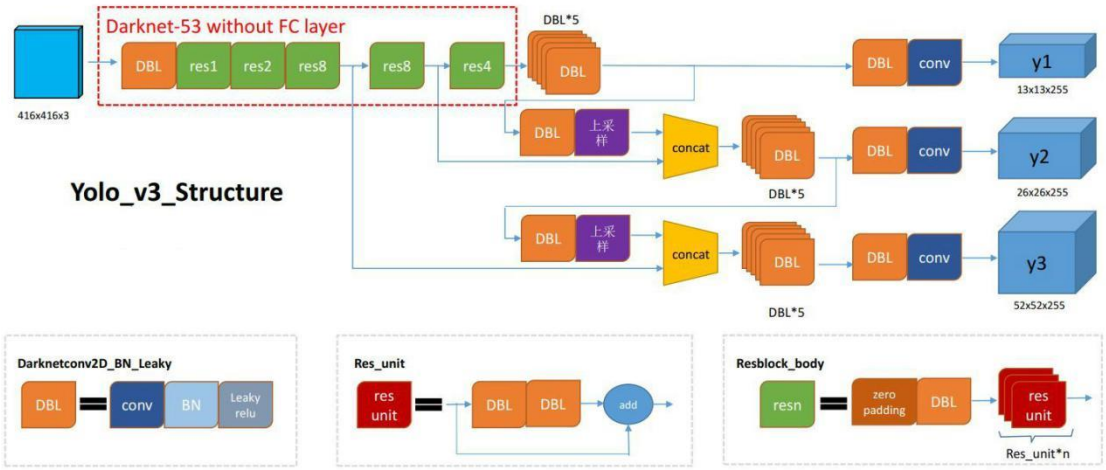

图 3 Yolo_v3 的结构图

2.1.2 使用 Yolo_v3:darknet 框架实现过程

1. 修改数据集格式

由于 Yolo_v3 所留出来的数据集接口适用于 voc 类型的数据集，直接使用原来的数据集是无法被读取的，故而要将原来的图片及其对应的 txt 文件进行一定的处理和修改。

首先了解一下 voc 数据集里面的内容，包括 **Annotations**（存对应图片的 xml 文件，主要是描述图片尺寸、物体类型、位置坐标等信息，是生成 labels 目录下 txt 文件的依据，只不过本次实验不需要），**ImageSets/Main**（存训练、测试相关的 txt 文件；txt 文件里内容是 JPEGImages 里图片的编号，本次实验也不需要），**JPEGImages**（存数据集的图片，训练用图片和测试用图片统一放在里面）和 **labels**（存 Yolo_v3 真正需要使用的 txt 文件。每一张图片都有一个 txt 文件与其对应，包含图片的所有物体信息，包括它的类型和定位）。

到这里我们可以知道我们的目标 txt 文件的格式为：

<object-class> <center-x> <center-y> <width> <height>

值得注意的是除了物体的类型，物体的中心点坐标，以及长宽都是百分比，即需要除以 128（图片的长或宽）。

在来分析一下我们目前所有的数据集的信息，目前的数据集中我们有图片和由这些图片的类型，物体的所在位置的坐标组成。为了得到 labels 里面的 txt 文件，我们只需要通过计算获得物体的中心点坐标和长宽就可以了，这也就是上文中所说的我们不需要由 xml 文件作为过渡获得 labels 中的 txt 文件的原因。

2. 修改基础配置文件

首先是 yolov3-voc.cfg 文件，里面指示了许多训练或测试的基础信息，例如 **batch**（一批训练样本的样本数量），**subdivisions**（batch/subdivisions 作为一次性送入训练器的样本数量），**width/height**（图片的宽高），**channels**（输入图像的通道数），**learning_rate**（学习率，决定权值更新的速度，设置得太大会使结果超过最优值，太小会使下降速度过慢），**max_batches**（训练的批次总数）等等这样的参数，这些参数的设置对训练结果的好坏有着很大的影响，例如将 batch 值设置的越大，训练的效果就越好，而参数 decay 的设置则能防止过拟合，使其对测试

集更友好，是需要反复调试的过程，只是这次的实验训练耗费时间太长，没能研究这些参数与准确率的关系。本次实验我们的配置如图 4 所示。

```
# Training
batch=64
subdivisions=16
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 = 1500
policy=steps
steps=40000,45000
scales=.1,.1
```

图 4 yolov3-voc.cfg 文件配置示意图

其次需要 voc.names 配置文件，里面存放的是本次实验所能识别的所有物体类型，注意类型的顺序编号，这对应着图片 txt 文件中的类别标志。本次实验的配置如图 5 所示。

```
bird
car
dog
lizard
turtle

```

图 5 voc.names 文件配置示意图

最后需要更改的是 voc.data 配置文件，该文件指示了训练集和测试集具体包含了哪些文件，这些信息分别在 train.txt 和 test.txt 里面指示，而 voc.data 则指示这两个文件的具体位置。本次实验的配置如图 6 所示。

```
classes= 5
train  = /root/zenghape/darknet/VOC-Own/train_docker.txt
valid  = /root/zenghape/darknet/VOC-Own/test_docker.txt
names = data/voc.names
backup = backup
```

图 6 voc.data 文件配置示意图

3. 开始训练

在配置完成后就可以开始训练了，这里在训练时我们给予了该模型一个初始权重（darknet53.conv.74），使用这个初始权重的好处是该权重是经过了预训练后得到的权重文件，在此基础上进行训练能获得更好的效果。

在训练完成后，我们对输出得到的数据进行一定的处理之后可以发现，loss 值一直在慢慢减少，如图 7 所示（值得注意的是本次训练过程中，由于在还没有迭代到既定次数，即 1500 次，的时候就与服务器断开了连接，故而能展示的只能是迭代了 1316 次的情况）

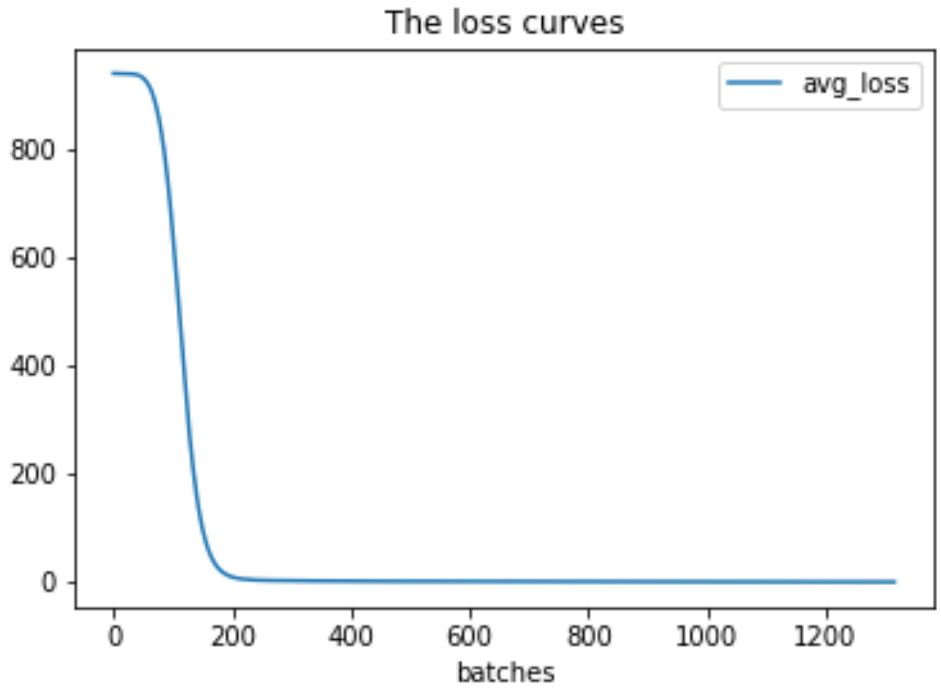

图 7 loss 值随迭代次数的变化曲线

而 IOU 的数值在训练中也有所体现，它与迭代次数的关系如图 8 所示，可以发现随之训练迭代次数的增多，模型对于物体位置的判断准确度也在增加。这里的横

坐标比我们的迭代次数大了不少，原因是我们每次迭代中又分了 16 次批量输入图片（见图 4），而每次输入图片会用三个不同的尺度进行预测（Region 82, Region 94, Region 106），但是有的尺度未必能识别到物体，故而每次能得到的 IOU 的数量在 1 到 3 之间，故而有了这样的差异。

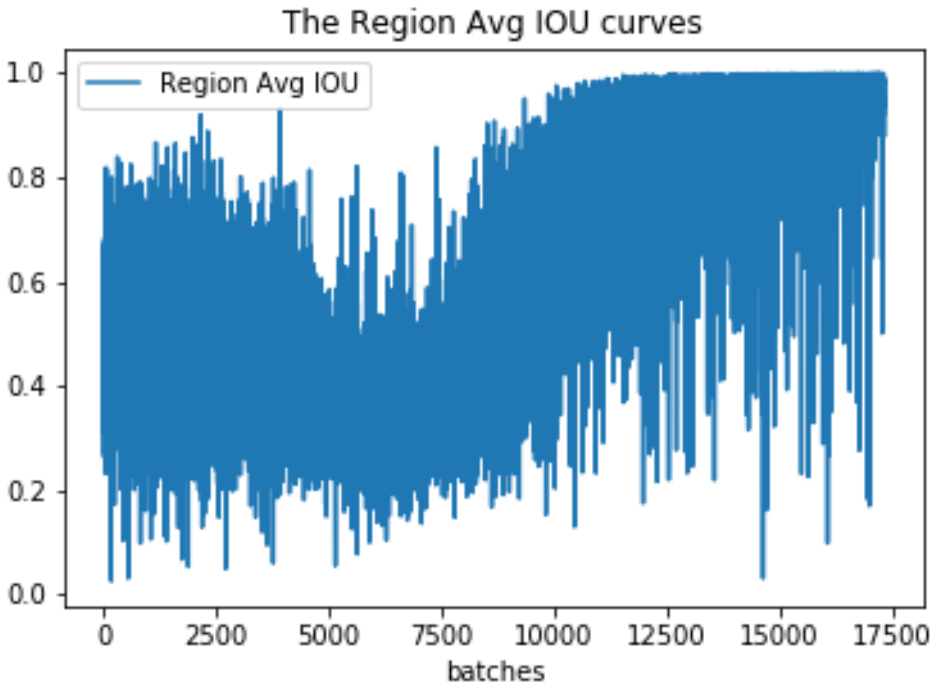

图 8 IOU 数值随迭代次数的变化曲线

4. 开始测试

在训练完成后，对 loss 值和 IOU 的变化进行了评估，确定了该训练模型的有效性，这时，使用测试集对该模型进行进一步的测试。

测试集为剩余的 90 张图片，测试结果部分展示如图 9 所示

```
Try Very Hard:/root/zenghape/darknet/VOC-Own/VOCdevkit/VOC2007/JPEGImages/000752.jpg: Predicted in 0.019970 seconds.
bird: 57%
```

```
Try Very Hard:/root/zenghape/darknet/VOC-Own/VOCdevkit/VOC2007/JPEGImages/000759.jpg: Predicted in 0.020376 seconds.
bird: 59%
```

```
Try Very Hard:/root/zenghape/darknet/VOC-Own/VOCdevkit/VOC2007/JPEGImages/000786.jpg: Predicted in 0.020260 seconds.
car: 58%
```

```
Try Very Hard:/root/zenghape/darknet/VOC-Own/VOCdevkit/VOC2007/JPEGImages/000787.jpg: Predicted in 0.020007 seconds.
car: 84%
```

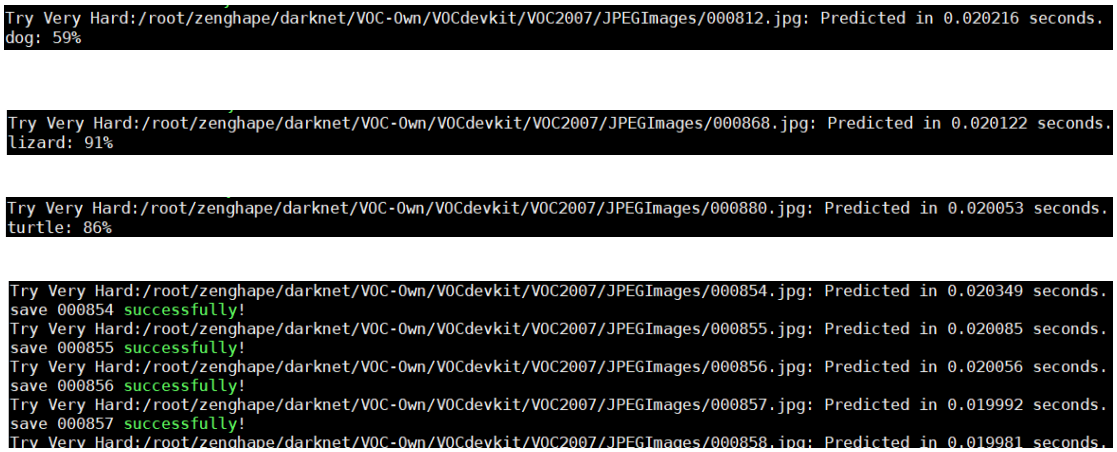

图 9 部分测试结果展示

由图 9 所示，可以发现对于部分图片的检测准确率已经达到了 50%左右，当然也出现了许多未能识别出物体的情况，这与我们的训练仅在 1316 次迭代就结束了有关，而测试用的又是迭代到 900 次的权重文件（因为代码的设计导致其在 1000 次以上的时候不保存权重文件），而这也是训练结果比较好但测试结果没那么好的原因，毕竟期间还差了两百多次训练。

2.1.3 使用 pytorch-yolov3 + keras-yolov3 尝试实现过程

我们调查到有两种比较主流的通过 python 的库函数实现的 YOLO 框架的方式，但这两种方式都不能很好的完成我们需要的从训练到预测的过程，我们首先使用 Pytorch-yolov3 方式，解决一些问题后，我们达到了能利用已经训练好得到的.weight 权重文件来进行预测的效果。

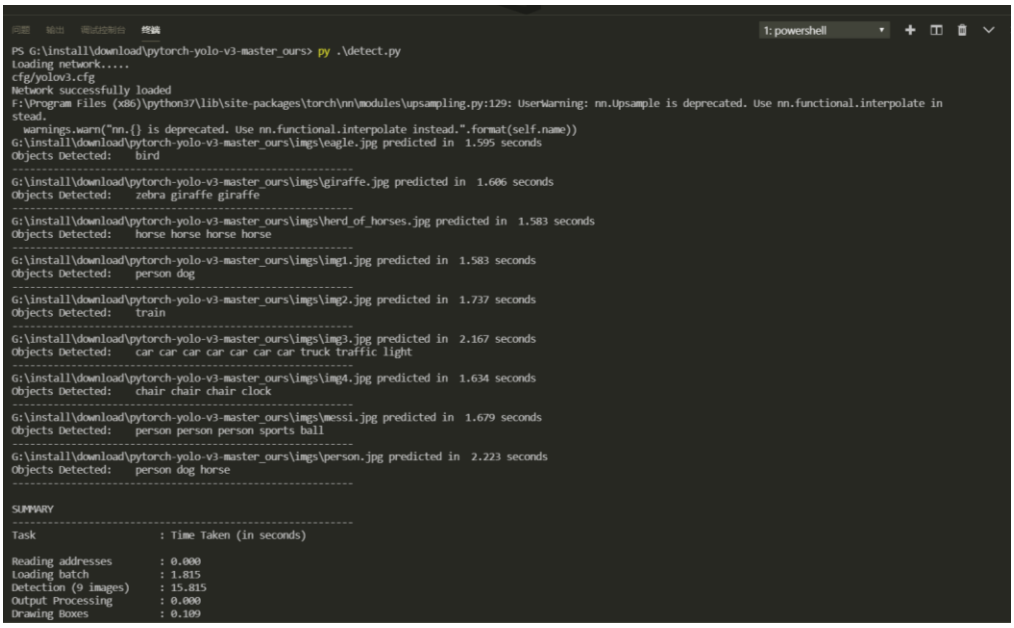

图 10 pytorch-yolov3 运行过程截图

图 11 pytorch-yolov3 运行结果样例

但由于不是通过要求的数据集训练得到的权重文件，所以我们无法在要求的测试集上输出结果，而 Pytorch-yolov3 这种方式又难以实现训练于自己的数据集。于是我们尝试运用 keras-yolov3 的方式，不过这种方式能够用要求的训练集来训练，并能输出权重文件，但其在检测上主要是用于视频实时检测，而图片检测模式下，我们只能使其输出识别到的物体类别而不能标注位置。

之后我们尝试在 keras-yolov3 上用要求使用的训练集训练得出权重文件，然后给 Pytorch-yolov3 来做检测，但最终并没有得到理想的结果。遇到的主要问题第一是在 keras-yolov3 方式中我们没能实现使用 gpu 来训练，使得训练时间非常长，一次 epoch 需要十五分钟左右，导致我们在有限的时间内难以调试并得出合适的权重文件；第二是在 Pytorch-yolov3 用要求的数据集做测试时，有部分图片会导致程序崩溃，我们猜测的原因是要求使

用的数据集有些图片质量较差，不能识别到任何物体而导致使程序崩溃，但我们并没有成功解决这个问题，导致最终这个方案难以得到结果。